Tangram: Model-based integration and testing of complex high-tech systems

A collaborative research project on multidisciplinary integration and testing of embedded systems

Editor:

Jan Tretmans, Embedded Systems Institute

Publisher:

Embedded Systems Institute, Eindhoven, The Netherlands

Embedded Systems
INSTITUTE

Publisher:
Embedded Systems Institute
TU/e Campus, Den Dolech 2
P.O. Box 513, 5600 MB Eindhoven
Eindhoven, The Netherlands

Keywords:
system engineering; modeling; integration; testing; embedded systems; multidisciplinarity

ISBN: 978-90-78679-02-8

The Tangram project has been executed under the responsibility of the Embedded Systems Institute, and is partially supported by the Netherlands Ministry of Economic Affairs under the SenterNovem TS program (grant TSIT2026).

Foreword

With the ancient Chinese Tangram, creative people can make an unlimited number of designs from just seven pieces of wood. In an era of increasingly advanced technology, creativity is also what we need to put together ever-more sophisticated machines with limited resources. In short, to tackle growing complexity, we need new ideas to save time and money in integration and testing. These were the goals for the Tangram Project.

Why integration and testing? A lithography system is an extremely complex semiconductor manufacturing machine that transfers nanometer circuit patterns onto silicon wafers to make semiconductor chips. It comprises a multitude of subsystems, each of which is already an integrated conglomerate of optics, mechanics, mechatronics, electronics and software. Integrating such a system is about molding numerous subsystems into one coherent machine with extreme specifications. Put simply, integration and testing is the phase when all of these handling-, measuring-, positioning- and imaging subsystems first come together and should work together. It's the time when newly invented concepts first confront each other. It's the point when a small mistake can have big consequences. From a business viewpoint, it's where costs per hour are at their peak and long lead times hit the hardest.

Nearly seven years ago, ASML successfully introduced the first dual stage TWINSCAN^{TM} lithography system. It has been a huge success. Yet, this system first highlighted that new developments would require much more effort, have longer lead times and higher costs than ever before. We would need major breakthroughs in integration and testing for the challenges we face now and in the future. These challenges include:

- Growing complexity of new products - new technologies such as immersion lithography and Extreme Ultra Violet (EUV) require an even more sophisticated and smarter test approach
- Greater diversity of products for various nanotechnologies
- More modularity in designs and testing enabling:
  - shorter lead times and easier interactions with third parties (outsourcing)
  - further product diversification, customization and re-use offering more added value for customers and ultimately consumers
  - faster ramp-ups and fault diagnosis for customers
- More systems — integration and testing forms a substantial part of overall lead times and our capital expenditure
- Greater machine accuracy — not only important in R&D but also crucial in manufacturing
- Higher customer expectations — demanding shorter times-to-market and shorter learning curves for new products to meet their final specifications.

The Tangram project is helping us offer a significant reduction in lead times and in cost of integration and testing. It has helped us to achieve ‘faster, cheaper, better’ without compromising product quality or reliability. By providing accurate models of system components that are not yet physically realized, it truly reduces potential problems at the earliest stage.

In wanting to take integration and testing to the next level, ASML realized that this area of research would fit perfectly with the Embedded Systems Institute’s (ESI) mission to apply academic results to an industrial environment. This gave birth to the Tangram project in which universities, industrial partners, ESI and ASML undertook research that can benefit the semiconductor and many other industries. After four and a half years of work, the Tangram project contributors proudly present this book with their research findings. For ASML, the Tangram project has delivered many improvements to our approach to integration and testing, especially regarding strategies and sequences. The project proved that - supported by the proper infrastructure - model-based approaches are the key to successful test automation, early integration and diagnosis.

But most importantly, we have seen once more that supporting cooperative research, with partners having different yet relevant backgrounds, can really lead to breakthroughs in achieving our goals. ASML is delighted to have been the ‘carrying industrial partner’ in this project. On behalf of ASML, I look forward to seeing what this project, and others like it, can bring to ourselves and to the industry as a whole.

Ir. Harry Borggreve
Senior Vice President of Development and Engineering
ASML Netherlands B.V.
Veldhoven, The Netherlands
October 2007

Preface

This book is the second in a series of publications by the Embedded Systems Institute that report on the results of ESI research projects carried out in collaboration with its industrial and academic partners. This volume summarizes the key results of the Tangram project, which was devoted to model-based integration and testing of complex high-tech products and involves the collective work of researchers and engineers from ASML, together with research groups at Eindhoven University of Technology, Delft University of Technology, the University of Twente, Radboud University Nijmegen, TNO, Science and Technology and ESI. The topic of integration and testing is of central importance to the high-tech industry, and fits perfectly with the ESI Research Agenda. Like almost all ESI-coordinated projects Tangram was organized as an industry-as-laboratory project, in which real cases in an industrial setting provide the experimental platform to develop and validate new methods, techniques and tools. This approach proves very successful in producing substantial, industrially relevant results, whilst maintaining high scientific standards.

It is clear that an effort like Tangram can only achieve such results with a very strong commitment of all involved to contribute, not only in terms of one's own area of expertise, but also by taking responsibility for the success of the project as a whole. We have been most fortunate to have ASML as the carrying industrial partner of this project, giving the project access to the testing and integration problems of their lithography machines, which are indisputably among the most sophisticated high-tech machines one can find. The consortium of researchers in Tangram have really seized upon this opportunity to test and elaborate the viability and practicability of their ideas, leading to impressive improvements in the test and integration process, as well as the articulation of relevant new research questions.

I would like to thank all participants involved for their commitment and contributions that have secured the success of Tangram. The support of ASML and the Dutch Ministry of Economic Affairs, which provided the financial basis for carrying out the project, is gratefully acknowledged. The project now having brought substantial benefits for all involved, we hope to share some of the results and insights with our wider industrial and scientific environment through this book.

Prof. dr. Ed Brinksma
Scientific Director & Chair
Embedded Systems Institute
The Netherlands
August 2007

Contents

1	Tangram: an overview of the project and an introduction to the book	1
2	ASML: the carrying industrial partner	21
3	Integration and test planning patterns in different organizations	31
4	Integration and test planning	45
5	Test time reduction by optimal test sequencing	61
6	Optimal integration and test planning for lithographic systems	73
7	Model-based integration and testing in practice	85
8	Using models in the integration and testing process	101
9	Timed model-based testing	115
10	Model-based testing of hybrid systems	129
11	Test-based modeling	143
12	Model-based diagnosis	163
13	Costs and benefits of model-based diagnosis	179
14	A multidisciplinary integration and test infrastructure	187
15	The Tangram transfer projects: from research to practice	199
A	Tangram publications	209
B	List of authors	219

Chapter 1

Tangram: an overview of the project and an introduction to the book

Author: G.J. Tretmans

1.1 Introduction

Integration and testing of complex embedded systems, such as wafer scanners, medical magnetic-resonance-imaging (MRI) scanners, electron microscopes, and copiers, is expensive, time consuming, and often faults are found that could have been detected much earlier in the development trajectory. The quest for ever faster introduction of new products, preferably with less costs and with better quality, makes high demands on the development process of such systems. Whereas traditionally a lot of attention and effort has been devoted to the design and implementation phases of system development, we currently see an increasing awareness that the time-to-market pressure makes also high, and perhaps even higher demands on the integration and testing phases.

Consequently, it is of increasing importance to improve the integration and testing process of embedded systems, and to make this process faster, cheaper, and better. This is exactly the aim of the Tangram project: developing methods, techniques and tools to reduce lead time, improve quality, and decrease costs of integration and testing of high-tech embedded systems. Tangram is an applied research project, in which different universities and companies, coordinated by the Embedded Systems Institute, have collaborated on achieving these goals.

The goal of this book is to give an overview of the outcomes of the Tangram project.

The goal of this chapter is to introduce the project and the book, and to provide an overview and summary of it.

First, the problem scope of Tangram, i.e., the challenges of integration and testing of complex systems, is sketched in Section 1.2. Section 1.3 describes Tangram and some of its characteristics: its way of working in an *industry-as-laboratory* setting, its bias towards model-based approaches, and its true focus on integration and testing.

Section 1.4 introduces the five research areas: *integration and test planning*, *model-based integration*, *model-based testing*, *model-based diagnosis*, and *integration and test infrastructure*. The subsequent chapters are organized following these lines of research, so this section also serves as a reading guide for the rest of the book. Each research area is briefly introduced and positioned within the integration and testing domain, the main outcomes are mentioned, and pointers refer the reader to the chapters where the full details and results can be found.

Sections 1.5 and 1.6 address some non-technical issues, such as the transfer of project results to the industrial partner in the project, and some particularities of organizing such a collaborative project of industry and academia. Lastly, Section 1.7 summarizes the achievements, open issues, and perspectives.

This book intends to give an accessible overview of the Tangram results for anybody interested, and in particular for technical professionals working in the area of integration and testing, who may recognize the challenges that Tangram addresses. The scientific orientation of the different chapters varies, but none of them has the intention to give a full scientific treatment of the topic covered; where necessary references to other publications, such as journal and conference papers, are made. The different chapters are independent, so that sequential reading is not necessary.

1.2 Integration and testing

Complex systems are designed following a divide and conquer strategy. In the design phase the requirements that the system must satisfy, and the tasks that the system must perform, are decomposed and divided over different subsystems, modules, and components. For each of these subsystems, modules, and components, in turn, the requirements are determined, and further decomposed and divided over smaller subsystems and components, and so forth, until the components reach a level of detail so that they can directly be implemented, constructed, or bought; see Figure 1.1.

After the decomposition there is composition: when all components are ready they must be combined and integrated to form larger modules, which are again integrated, until the complete system has been obtained from the integration of all subsystems. Moreover, in between all these integration activities, the individual components, combined components, integrated modules and subsystems, and the whole system will be tested to check their quality and compliance with their requirements. This process continues until eventually the complete system has been integrated and tested. This integration and test process, which starts with the individual components and ends with the

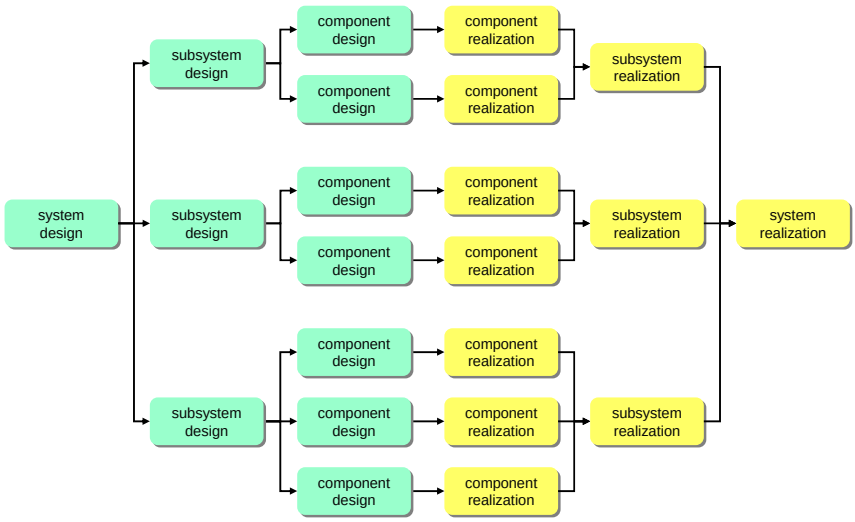


Figure 1.1: Decomposition and composition.

completely integrated and tested system, constitutes the scope of the Tangram project; see Figure 1.1.

Challenges

In the integration and test process, integrators and testers face different challenges. First, there is the ever growing complexity of the individual components, and the increase in the number of components. With the latter also the number of possible interactions between components increases and, consequently, the number of potential integration problems – on average the number of possible interactions increases quadratically with the number of components. Apart from designed and intended interactions between components, there are unintended and undesired interactions which often cause problems during integration, such as heat production or electromagnetic interference.

A second challenge is the market demand for ever faster introduction of new products, with more features, with lower cost, and with higher quality. The integration and testing phases, being closer to the new product's delivery date, will even more sense the time-to-market pressure than the specification and design phases. The latter phases, however, are likely to introduce more faults in the system, because of the quest for more new features, which again increases the pressure on testing.

Problems occurring during integration and testing can be process-oriented, as well as product-oriented. Integration can be difficult, or impossible, because components do

not fit together. Many integration problems are caused by imprecisely or ambiguously defined, or not correctly implemented interfaces. It may also occur that all components together, although correctly implemented, do not deliver the requested performance, functionality, or quality of service. A lot of these problems can actually be attributed to the specification or design phase, and are only detected much later during the integration and testing phase. Consequently, these errors are expensive in terms of repair and rework. Moreover, this late detection of such faults almost always leads to delays in the integration and testing trajectory, and thus in the time-to-market of the product.

Also without any problems being detected, the lead time of the integration and test trajectory is often an issue. With the classical way of integration, first the realizations of the components need to be available before integration and testing can start. This implies that there will be many dependencies and critical paths in such a process, with many integration and test activities waiting for others to be ready. If there is a disruption somewhere in this process, e.g., because a component is delivered late, this will immediately have big consequences for all subsequent activities in the process leading to an increase in the duration of the integration and test trajectory. Moreover, apart from the extra costs of delay, interest costs may play an important role if very expensive components have to be integrated early in this trajectory.

Another challenge is to specify and develop the test cases to test all components and subsystems thoroughly. Tests are needed that sufficiently cover all component functions and requirements. This should include criteria and methods to analyze the test results. Moreover, a technical test environment and infrastructure should be made available that facilitates easy and efficient execution of all these tests. As with integration, test effort tends to increase more than proportionally with the growing complexity and size of systems: if the number of components doubles, the number of possible test cases grows with the product of the sizes of their input spaces.

With an increasing number of components, timely diagnosis is also getting more and more problematic. If a failure occurs it must be identified, and the root cause of the failure must be localized for repair. Diagnosis is an issue during both development testing and operational use by the customer.

Last, but not least, these integration and testing challenges aggravate dramatically, if also external, third-party components are to be tested and integrated, and when the components to be integrated emanate from different engineering disciplines, such as mechanical, optical, mechatronic, electronic, and software engineering. And especially in the area of complex embedded systems third-party components and multidisciplinarity are very common.

All these challenges make that the integration and testing process is an interesting and stimulating area for research and development. New methods, techniques, and tools are demanded, so that integration and testing are able to keep pace with the increasing complexity of embedded systems, and with the growing demands on time-to-market, cost, and quality. During four years, Tangram has worked on these challenges.

1.3 The Tangram project

The Tangram project is an industrial-academic research and development project managed by the Embedded Systems Institute. The goal of Tangram is to develop methods and techniques to reduce lead time, improve quality, and decrease costs of integration and testing of high-tech multidisciplinary systems, i.e., to contribute to solutions for the integration and testing challenges described in Section 1.2.

In Tangram, researchers and engineers from ASML, TNO, and Science & Technology, have worked closely together with researchers of Delft University of Technology, Eindhoven University of Technology, the University of Twente, Radboud University Nijmegen, and the Embedded Systems Institute. The project started in March 2003 and lasts until December 2007, and it is financially supported by the Netherlands Ministry of Economic Affairs.

In tackling the challenges of integration and testing, Tangram started from three principles. First, to stimulate the applicability and transfer of results, the project has worked in a setting which is referred to as *industry-as-laboratory*. Second, the new methods and techniques are based on *model-based* approaches. Third, the project shall explicitly focus on the integration and testing phases of system development.

Industry-as-laboratory

The academic-industrial cooperation in Tangram took place in a setting referred to as *industry-as-laboratory* [103]. This means that the actual industrial setting is used as a laboratory, akin to a physical or chemical laboratory, where new theories, ideas, and hypotheses, mostly coming from the academic partners in the project, are tested, evaluated, and further developed. This setting provides a realistic environment for experimenting with ideas and theories, but, of course, care should be taken that the normal industrial processes are not disrupted. Moreover, the industry-as-laboratory setting facilitates the transfer of knowledge from academia to industry, and it provides direct feedback about the applicability and usefulness of newly developed academic theories, which may again lead to new academic research questions.

For Tangram, the laboratory has been provided by ASML. ASML is the leading global company for lithography systems for the semiconductor industry. Its wafer scanner machines involve highly complex configurations of embedded systems with extreme requirements regarding performance and precision. During the development of these machines, ASML experiences many of the integration and testing challenges sketched in Section 1.2. Consequently, ASML provides an ideal, stimulating and demanding laboratory environment for Tangram. Chapter 2 elaborates on ASML and its role in Tangram.

Models

The basic approach for tackling the integration and testing challenges is the use of *models*. A model is an abstract view of reality, in which essential properties are recorded,

and other properties and details considered not important for the problem at hand, are removed.

An every-day example of a model is a road map: a road map only contains lines representing roads, and circles representing cities. The map abstracts from many other details of reality, such as buildings, forests, railways, mountains, the width and the kind of pavement of roads, et cetera. Such a map, i.e., a model, may very well help with planning your trip by car from Eindhoven to Amsterdam, because all relevant details for such a trip are there. For planning a railway trip, however, or for calculating the altitude difference between Eindhoven and Amsterdam, such a road map is useless. Another map, i.e., another model with other abstractions, is needed such as a railway map, or a geographic map, respectively.

Models can be analyzed, they can be the basis for calculations, they can help in understanding a problem, they can form the basis for constructing a system, for testing it, and for diagnosing it, and when they are expressed in an executable language, they can be simulated (simulation models). Models can be made of systems, subsystems, or components, but also the development process, or part of it such as the integration and test process, can be modeled. Models can be made *a priori* to guide and analyze the design, or *a posteriori* to analyze, test, or diagnose an existing system. Different models can be made of the same system, each focusing on a different kind of properties, e.g., a functional model, a performance model, or a reliability model.

In Tangram different types of models are used to tackle different kinds of integration and testing challenges. Process models are used to analyze and optimize the integration and testing process, fault models are used to model potential system faults and their implications, and models of component behavior are used for simulation, model-based integration, model-based testing, and model-based diagnosis.

Focus on integration and testing

Tangram deals with integration and testing, and only with integration and testing, that is, with the right-hand side of Figure 1.1. As noted above in Section 1.2, many integration and test problems can be attributed to errors made during the specification or design phase, but it is not an option for Tangram to propose, or work on improvements or changes in the specification or design methodology. The new methods and techniques should be able to cope with the current reality of imperfect specifications and designs.

1.4 The research areas

Different research directions have been pursued in Tangram, addressing many of the integration and testing challenges discussed in Section 1.2, and covering different parts of the right-hand side of Figure 1.1. This section briefly introduces these research areas, it describes their main results, and it positions each area in the integration and testing domain as depicted in Figure 1.2. This figure extends Figure 1.1 with explicit

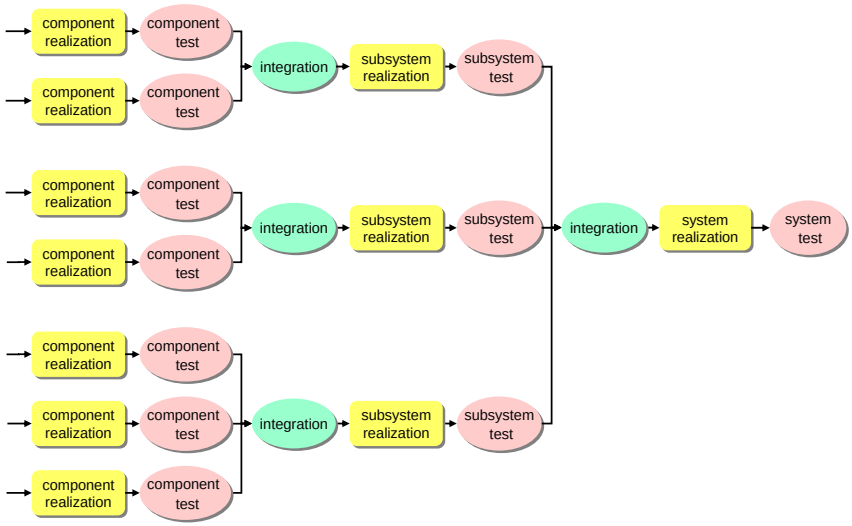


Figure 1.2: Integration and testing activities.

integration and testing activities. The other chapters of this book are organized following these lines of research, so this section also serves as an introduction and reading guide for the rest of the book. For that purpose, this section contains ample references to the other chapters. Tangram has the following research areas:

- *Integration and test planning* uses models to plan and schedule the different activities in the integration and test process, and to optimize this process with respect to time, cost, or quality – Chapters 3, 4, 5, and 6.
- *Model-based integration* introduces component models in parallel to, and as replacement for component realizations, so that many integration and test activities can be performed on these models, e.g., through simulation, long before the real realizations are available – Chapters 7 and 8.
- *Model-based testing* is concerned with comparing models with realizations using automatically generated and executed test cases – Chapters 9, 10, and 11.
- *Model-based diagnosis* localizes a faulty component in a (sub)system when a failure has been detected – Chapters 12 and 13.
- *Integration and test infrastructure* provides a generic infrastructure for the execution of tests, model-based integration, and model-based testing – Chapter 14.

Integration and test planning

The area of integration and test planning is a process-oriented line of research, aiming at optimizing the planning of integration and test activities in the integration and test trajectory. It may apply to the whole integration and testing phase when all integration and testing activities are optimally scheduled, or to a single testing activity when an optimized test sequence is determined for the test cases within such a test activity; see Figure 1.3.

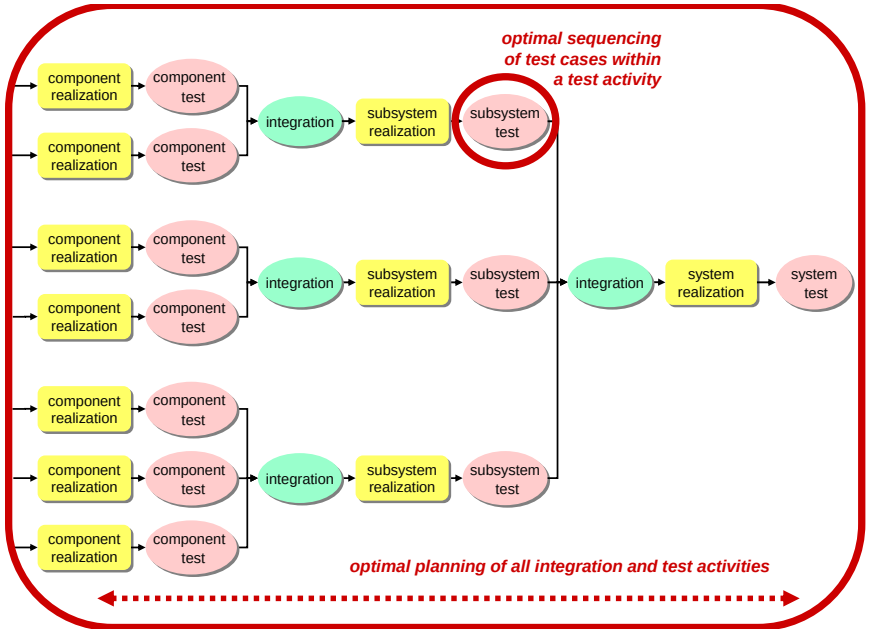


Figure 1.3: Integration and test planning.

Given a model of the planning problem consisting of the possible faults that a system may have together with the probabilities of their occurrence, a set of tests together with the probabilities with which they may detect these faults, possible integration steps, the cost of tests and integration steps, and the impact of remaining non-detected faults, an optimum test and/or integration sequence is calculated. Optimization can be along different parameters, such as minimal cost, minimal remaining risk, or minimal duration of integration and test time.

With such planning techniques it turns out to be possible to reduce the integration and test time of ASML machines with 10-20% compared with the currently used manual, expert-based planning approach. Moreover, these techniques are especially useful when something changes during the process, e.g., when a component is delivered later than expected. An updated planning is then easily generated by adapting some pa-

rameters and recomputing the optimum integration and test sequence. The method and techniques are effective during the integration and testing trajectory of the development of a new machine or prototype, as well as during the integration and testing phase of the manufacturing of a machine of an existing product line. In both cases, of course, it is important that a reasonably complete list of possible faults is available, and that valid estimations for the parameters in the model can be made, such as the cost of test cases and the impact of faults. Moreover, the method does not say anything about how to specify the test cases which should detect the possible faults.

Integration and test planning is described in four chapters. First, Chapter 3 investigates the kinds and patterns of integration and test plans of different organizations with differing business drivers, such as time, cost, and quality. The question is whether the plan of a time-to-market driven company like ASML is structurally different from, e.g., a quality (safety) driven company in the aircraft industry. Then, Chapter 4 presents a step-wise approach for making an integration and test plan. The steps in this approach are modeling the system for integration and test planning, making an integration sequence, positioning of test phases in the integration sequence, planning the individual test phases, and optimizing the integration and test plan. Chapter 5 gives more technical details on the planning of an individual test phase, including the kind of models used and two case studies performed at ASML showing the benefits of this test sequencing method. Finally, Chapter 6 discusses the techniques for integration sequencing. Here, test phases are positioned as soon as they can be performed. This chapter also presents the results of two ASML cases to illustrate the benefits of the method.

Because of the successes in terms of reductions of integration and test time demonstrated in the case studies, these methods and the corresponding tool, called LONETTE, have been transferred in a special transfer project, and they are currently used within ASML. Transfer projects are discussed in Chapter 15.

Model-based integration

Whereas the Tangram activity of integration and test planning is process-oriented, i.e., uses models for the integration and test *process*, all other Tangram activities are more *product*-oriented, i.e., models are made of the behavior of the system under development, its subsystems, and its components.

Model-based integration aims at reducing the critical path of integration and testing by trying to detect as many faults as possible on *models*. Currently, most methods require completed realizations of components to start integration and testing. By making models of components, many properties can already be analyzed on these models, without the need for waiting for realizations to become available. Models of single components, and combinations of models can be analyzed, simulated, model-checked, and tested. If the models are formal, i.e., is expressed in a formal language with mathematically and precisely defined semantics, then properties of the model can even be proved with mathematical precision. As an example, a model-checker can formally prove that the behavior of a model is deadlock-free. Chapter 7 gives examples of

model-checking for an ASML component.

Even more potential problems can be detected early when models are combined with already available realizations of components; see Figure 1.4. (cf. hardware-in-the-loop simulation and testing, or the use of stubs in software testing). This might lead to an incremental integration and test approach starting with a system of component models, where then gradually models are replaced by component realizations.

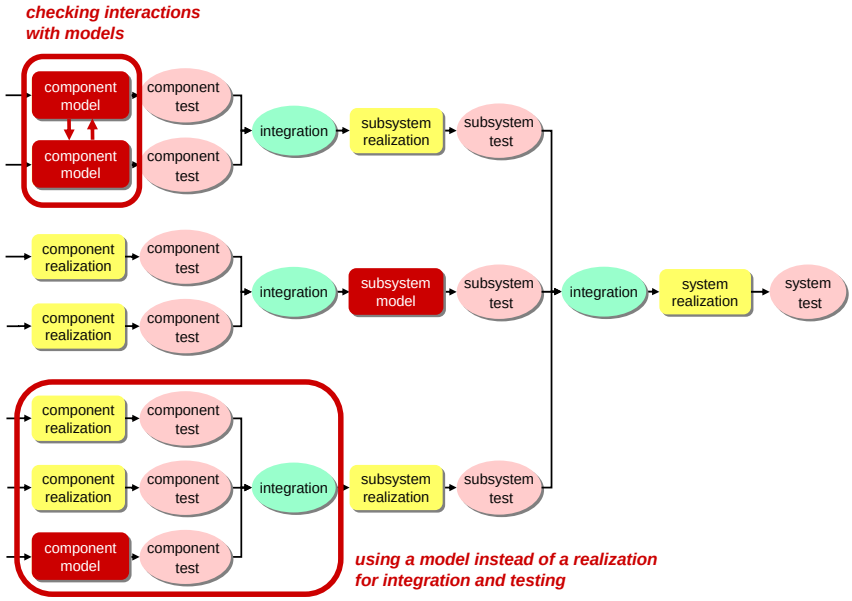


Figure 1.4: Model-based integration.

Chapter 7 describes the method, and its successful application to modeling, integrating, and testing some interacting components of the ASML wafer scanner. Chapter 8 embeds the method in an industrial integration and test process, where both software and hardware components are upgraded and integrated.

The feasibility and usefulness of model-based integration and testing have been shown in a number of case studies. The advantages are that a lot of potential problems can be detected earlier, and thus cheaper, often before the realization of the component has been built. In particular, interfaces, which are the source of many integration problems, can be analyzed and model-checked before any interface is built, allowing the early detection of interface design errors. Secondly, the method makes it possible to increase the concurrency and to decrease the critical dependencies in the integration and test process, since for many tests there is no need to wait for realizations. Thirdly, many tests can be performed much faster in a simulation environment than in a real environment. Finally, if models are used for testing, there is less need for expensive

equipment, which holds for both test equipment and expensive parts of the system under test.

These advantages, of course, should outweigh the extra effort of making models. Not in all cases it is profitable to make this extra investment. In Chapter 8 it is shown how the integration and test planning method of Chapter 6 can be easily extended to plan the making of models. This is achieved by including a model as an extra, alternative component in the planning model, with the model's development cost, development time, and benefit expressed as possibly earlier detected faults. The planning and optimization techniques can then be used to predict whether making a model of a particular component can be expected to be advantageous or not.

Model-based testing

Model-based testing involves checking whether a system or component realization behaves in accordance with its model. This means that the model is taken as the component specification to which the realization must conform. Checking this conformance is done by means of testing. Tests are algorithmically generated from the model, executed on the component under test, and the test results are automatically analyzed for compliance with the model. Conformance of realizations to models is important to guarantee that properties verified, simulated, or tested on models (e.g., during model-based integration) also hold for the realizations.

Potential advantages of model-based testing are that large quantities of test cases can be completely automatically generated from the model, while also test results can be automatically checked with respect to the model. Moreover, if the model is valid then all these tests can be proved to be valid, too. There is quite a lot of research and development going on at the moment in the area of model-based testing, both industrially and academically; see, e.g., [84] for an overview of the activities of Microsoft Research in this area.

In Tangram, extensions of the state of the art in model-based testing, as well as applications of model-based testing were developed. The first extension involves testing of real-time systems. Whereas the original model-based test theory is restricted to the testing of ordered events without a notion of real-time, Chapter 9 presents a theory, an algorithm, and a tool for testing systems where the time of occurrence of the events does matter. The next extension takes real-time a bit further by considering hybrid systems. These are systems in which not only discrete events must occur at specified moments in time, but also continuous variables must change according to specified trajectories, typically described by means of differential equations. Theoretical research, as well as tool development and an initial case study were performed for testing of hybrid systems; these are described in Chapter 10. Finally, Tangram was partly involved in extensions for symbolic testing, which considers the case when the tested events are parameterized with complex data structures and values. These results were published elsewhere [50, 51].

The principle of model-based testing has been successfully applied in the ASML

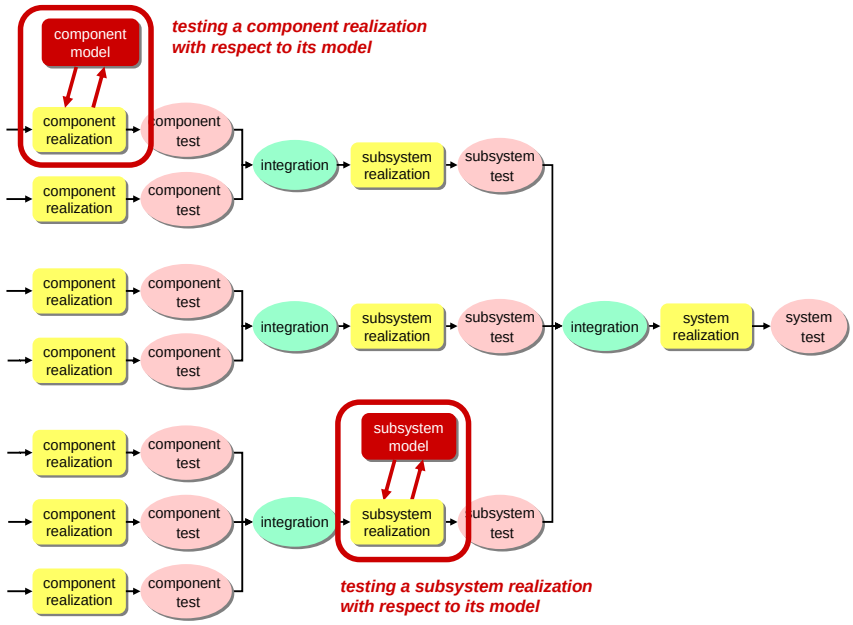


Figure 1.5: Model-based testing.

context. Some very small software components were modeled, tests were automatically generated from these models and executed on the realizations of these components, and some discrepancies between model and realization were detected. For larger-scale applications, however, models with sufficient detail must be available, or it must be possible to develop them from available documentation or from other information sources such as expert interviews. This turned out to be difficult in the ASML context. It looks like the effort necessary for developing models for systems with the complexity and size of the ASML wafer scanners is currently still too large. Moreover, the constant time-to-market pressure makes it difficult for engineers to spend much time on developing, or supporting the development of such detailed models.

Unfortunately, the models which were used for analysis during model-based integration could not easily be reused for model-based testing. First, for analysis the level of abstraction of the model is chosen by the modeler, who can abstract from difficult or unimportant details. In particular, analysis can also be performed on behavior scenarios while abstracting from alternative behaviors. For model-based testing, the level of abstraction is dictated by the details of the component realization under test. All details of behavior that the realization may exhibit, should be included in the model. This requires more detailed models. Secondly, model-based testing mainly focuses on testing of software, whereas the experiments done with model-based integration concentrated

more on hardware and physical components, in particular, in the experiments where models were combined with realizations of components.

The difficulty of obtaining models for model-based testing triggered new research questions focusing on alternative ways of obtaining models. One new line of research investigates the possibility of deriving a model from the observations made during executions of an existing component realization, using a kind of reverse engineering method. Such a model could then later be used, e.g., for model-based regression testing of a new version of that component. This approach was baptized *test-based modeling*, and is described in Chapter 11. This research looks promising, yet, is in a very initial phase.

Model-based testing is an area of research where the original Tangram goals were not achieved, but new research questions were triggered, which are now pursued. Consequently, the book chapters about model-based testing, Chapters 9, 10, and 11, have a more theoretical focus. It should be noted that model-based testing has successfully been applied in other application domains, in particular, there where better specification documents are available as a starting point for modeling, e.g., internationally standardized specifications, and where the systems are smaller, e.g., smart-card applications [122, 123].

Model-based diagnosis

Fault diagnosis is about the localization of faults. Given a system and a failure of that system, i.e., an observation that shows that the system does not comply with its requirements or expectations, fault diagnosis tries to point to the root cause of the failure, i.e., the component or part that should be replaced or repaired; see Figure 1.6. The failure may be observed during normal operation of the system, or it may concern a failure detected during testing.

In model-based diagnosis the localization is done using a model of the system behavior expressed as a composition of subsystems or components. When failure behavior is observed, a kind of backward reasoning on the model is used to infer which component(s) could have caused the failure. Probability annotations can be used to infer a probability for each potentially failing component.

In Tangram, LYDIA, a diagnosis method, modeling language, and tool, has been applied to components of the ASML wafer scanner. It turned out to be very successful for the diagnosis of electronic components during operational use when these components may break. In these cases reductions of diagnosis time were achieved from a couple of days with the currently used techniques, to only minutes when LYDIA is used. An investment, of course, is needed for making a diagnostic model, but the reduced diagnosis times more than compensate for these investments. Moreover, models, which are expressed in LYDIA as Boolean equation systems, can sometimes be partly automatically generated from the VHDL descriptions of the electronic component designs. For diagnosis of faults detected during development testing, and for diagnosis of software components, the model-based LYDIA approach turned out to be less successful. Diag-

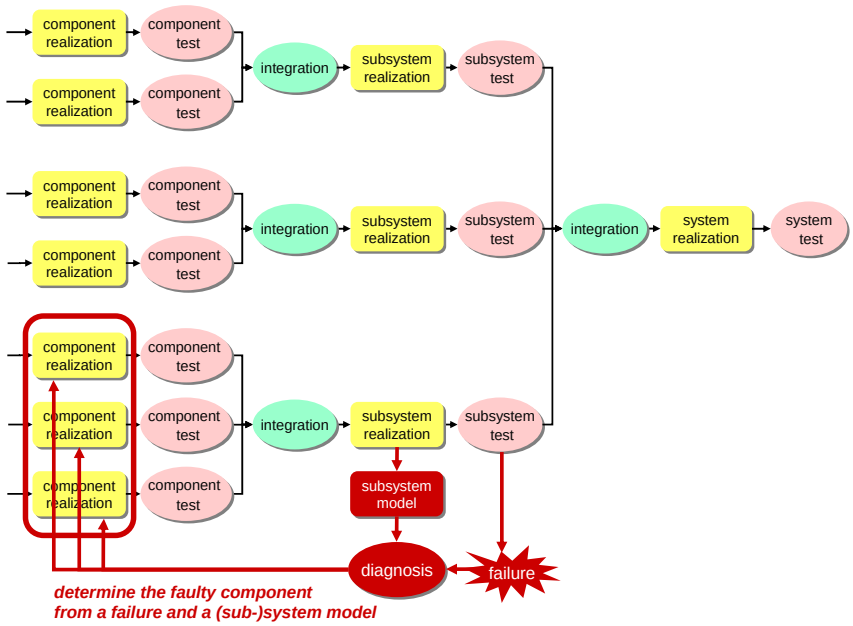


Figure 1.6: Model-based diagnosis.

nosis of software requires other approaches, such as *spectrum-based diagnosis*. This is investigated in the Trader project of the Embedded Systems Institute [40].

The principles of model-based diagnosis with LYDIA, and the extensions developed during Tangram, are described in Chapter 12. Chapter 13 gives some case studies performed at ASML, with emphasis on the modeling experiences, and, based on these, summarizes the benefits and costs of model-based diagnosis. The successes of model-based diagnosis have been transferred in a transfer project, and LYDIA is currently used at ASML; see Chapter 15.

Integration and test infrastructure

For testing, whether model-based or not, a test environment, also called test infrastructure, is necessary. Such a test infrastructure should facilitate that tests can be executed on the system under test, – components of the ASML wafer scanner –, that the interfaces of the system under test are easily and uniformly accessible, and that observations of test results can be made. Moreover, if executable models of components are combined with realizations during model-based integration, an infrastructure is needed that allows easy connection of realizations with these models written in, e.g., Simulink, Labview, or the process modeling language χ (Chi).

An integration and test infrastructure for ASML wafer scanners was developed, which acts as a generic and uniform way to control and observe the interfaces of the wafer scanner components. It can concurrently support functions of traditional test stubs and test drivers, it can be used for manual testing, model-based testing, and model-based integration, and it exposes both physical hardware and software interfaces as much as possible in a uniform way. The infrastructure was implemented using standardized technology based on middle-ware such as CORBA and DDS (Data Distribution Service, i.e., publish-subscribe, or message-broker middle-ware). The integration and test infrastructure, with its requirements and some implementation choices, is described in Chapter 14.

The integration and test infrastructure, often referred to in Dutch as the 'tang' because it surrounds the system under test as a pair of tongs, is partly ASML specific. It is currently being transferred to ASML, see Chapter 15, where it is successfully used as a test environment for manual testing.

1.5 Industrial transfer

An important goal of Tangram is to validate techniques, methods, and tools, developed at academia, in the industrial ASML context. Moreover, in case of success, actions should be initiated to transfer these to ASML. The different research topics within Tangram have shown different results with respect to applicability and transfer. Some methods have shown good results, feasibility, and applicability, but need further research and development before they can be really deployed on a daily basis within a company like ASML. Other techniques and methods turned out to be so mature and useful that their advantages are evident, and can be quantified. For the latter areas special transfer projects were initiated. These projects should transfer the methodology, techniques, and tools to the industrial partner ASML, so that they can be consolidated and institutionalized. In these transfer projects Tangram researchers have worked together with ASML engineers. The transfer projects were separately organized, and Chapter 15 describes their rationale and organization.

Considering the successes of integration and test planning, model-based diagnosis, and the integration and test infrastructure, transfer projects were organized for these three areas. Model-based integration has shown its feasibility and advantages, but in particular the support of the method is considered not sufficiently mature enough, yet, to initiate industrial transfer. As explained above, model-based testing has had some successful experiments, mainly on small systems, which show that the principles work well, but its main result is the identification of the need for methods to obtain models for systems of the complexity and size of the ASML wafer scanners. Test-based modeling was identified as one of the candidates for this. This is an example where the industry-as-laboratory approach has triggered new research questions.

1.6 Some organizational issues

Tangram is a cooperation between academia and industry. The way the project was initiated and organized was different from other research projects. In this section we briefly address some of these organizational, non-technical issues.

The beginning

Tangram is different. It is not an academic research project such as those sponsored by the Netherlands Organization for Scientific Research (NWO) or the Netherlands Technology Foundation (STW). It is larger, it is more application oriented with the goal to show a realistic, industrial proof-of-concept of new technologies and methods, and the participants come from more diverse organizations than in a traditional research project. It is also not a typical industrial R&D project. The nature of the research is more exploratory, and apart from business goals, the aim of the project is to deliver theses and scientific publications.

Consequently, the project started with a lot of learning. The participants had to learn about each other's interests and aims, about the ways of working and communicating with each other, and about the problems of the industrial partner. The initial problem description statement was rather abstract compared with those found in academic research proposals. It was more a collection of symptoms stated from an industrial, even business perspective, and not a well-defined scientific problem statement. This implied that the first task of the project was to clarify and concretize the problem statement of the industrial partner. On the one hand, this statement should be recognizable for the industrial partner and provide sufficient confidence for possible solutions. On the other hand, it should allow the academic partners to transform it into concrete research topics for the individual PhD. candidates linked up with their academic research agendas. Preferably, such research topic descriptions are clear and concrete from the beginning, i.e., before the PhD. candidates start, but this creates a dilemma: in order to define the project problem statement, domain knowledge from the industrial partner is required, and this domain knowledge can only be obtained with sufficient depth and detail during the project. This is rather different from traditional academic research projects where new problems and research areas build on existing results with which the researchers are completely familiar. This period of learning, and of problem extraction and definition, is, of course, very instructive in itself, but it does not immediately lead to publishable results, and it thus influences the lead time for the production of dissertations.

Industrial–academic cooperation

Tangram has a goal, and academic and industrial people collaborate to achieve that goal. But apart from this common goal, academia and industry have their own goals, and also their own ways of working. For academia, originality and specialized knowledge are important to distinguish themselves, and to survive in the world of academic

writing and publishing. For the industry it is not originality or intellectual challenge that counts, but the benefits for their business and intellectual property, where most of the problems have a multi-disciplinary nature, and the time horizon and cycles are shorter than those of academia. This tension between academic depth and industrial breadth must be carefully balanced, and the project participants should, of course, respect each others goals. Yet, interesting results can be achieved, as Tangram shows, with on the one hand a number of dissertations and scientific publications, and on the other hand the successful transfer of some methods, techniques and tools to the daily business of ASML. One of the working practices in Tangram was to work in short, incremental cycles, but with long term goals, so that industry can assess the progress of each increment, as well as academia can see, and focus on the global aims after four years. It is natural that the resulting dissertations have their major value in making theoretical results applicable, and not in the scientific depth of the results.

The industry-as-laboratory setting turned out to be fruitful, in the sense that there was a better focus of research on industrial needs and on results with industrial feasibility, and it facilitated acquiring the necessary domain knowledge for doing the industrial case studies. Moreover, it allowed the industrial participants to have a closer look at, and to perform some trials with some academic methods and techniques, and it triggered academics with new industrially relevant research directions.

Dissemination

Apart from the direct transfer to the industrial partner, various other ways of dissemination of the Tangram results were organized. First, there are the scientific results, including five dissertations to be defended at the partner universities, articles published in international journals, and papers presented at conferences and workshops. Also in professional journals, several publications appeared, and presentations were held at professional conferences, symposiums, and seminars; see Appendix A for a list of Tangram publications.

Several students participated in the project to do a trainee-ship or to perform their BSc. or MSc. project. Three symposiums were organized by Tangram itself, and a follow-up project at the Embedded Systems Institute is in the phase of being started. Dissemination also takes place via a new course on integration and testing provided by the Embedded Systems Institute, via internal courses at ASML, e.g., those in the context of the transfer projects, and via the academic partners who use the acquired knowledge in their lectures on testing techniques and system engineering. And, last but not least, there is this book.

1.7 Summary of results, open issues, and perspectives

For four years, Tangram has worked on methods and techniques for reducing lead time, improving quality, and decreasing costs of integration and testing, and on the application of these methods and techniques to the ASML wafer scanners. In the different

research areas, several results have been obtained in this period:

- Methods for *integration and test planning* were developed, implemented, and successfully transferred. These methods have shown that automatic planning of the integration and test trajectory may reduce its lead time with 10-20%.
- With *model-based diagnosis* a reduction of diagnosis duration from days to seconds was achieved for some specific diagnosis areas. Also this method and the corresponding tool were transferred.
- An *integration and test infrastructure* was developed and transferred. It showed that standard middle-ware solutions can be used to provide a generic infrastructure for the execution of tests.
- *Model-based integration* showed the benefits of using models to replace realizations for early integration and testing. Many integration issues could be detected several months before the realizations were available.
- With *model-based testing* it is possible to automatically generate test cases, but the dynamic and complex ASML environment makes it difficult to obtain or develop the necessary models. This resulted in new research questions addressing alternative ways of deriving or developing models, such as test-based modeling. Theoretical results and prototype academic tools were obtained for real-time, data-intensive, and hybrid extensions of model-based testing.

Apart from these specific results in the research areas, there are more abstract, global achievements. First, Tangram helped to increase the awareness of the growing importance of the integration and testing phases in system development, whereas traditionally system design and construction get more attention. Secondly, it demonstrated that improvements in the integration and testing process are possible by adopting structured, scientifically underpinned, and tool supported methods and techniques. Thirdly, it stressed the role of models and model-based thinking in integration and testing. Finally, a lot of feedback about academic methods and techniques was generated, and new research items and opportunities for future developments were triggered. We mention some of these:

- An important question for *integration and test planning* is how to obtain realistic estimates for the data and parameters in the planning models, e.g., the probability that particular faults occur, or an estimate for the impact or risk of a not detected fault. On the scientific side, there is the challenge of developing better algorithms and heuristics for the calculation of optimal integration and test plans. The complexity of these algorithms may hamper the scalability of the method.
- In the area of *model-based integration* more work is needed to make the method easily applicable in an industrial environment. On the one hand, this involves a stronger link to scientific methods of modeling and model-checking. On the other hand, the connection to industrially used simulation environments, e.g.,

Simulink, should be strengthened such that seamless integration of modeling, model-checking, simulation, and testing in a common infrastructure will be possible.

- The important question for *model-based testing* is how to obtain the models from which the test cases can be generated. New lines of research focusing on model generation have started, e.g., using model learning and test-based modeling, and others are possible, e.g., code abstraction. Case studies for model-based testing are currently better performed in an environment where stable and clear specifications, e.g., standardized specifications, are available, and not in the hectic and time-driven ASML context. Moreover, the theoretical work on real-time, data-intensive, and hybrid extensions should be continued, in particular, with more application oriented investigations.

- Also in the area of *model-based diagnosis* one of the major challenges is to cope with the computational complexity of the diagnostic algorithms in order to deal with scalability. On the practical side, a challenge is the further exploration of the methodology for other disciplines than the ones where it was successful within Tangram.

- The *integration and test infrastructure* should be further developed to a generic and standardized infrastructure for the ASML machines, so that not only during testing but also during operation easy and uniform access is obtained to system components for testing and monitoring. This involves the addition of extra functionality, but also the improvement of other, non-functional quality attributes such as reliability, configurability, maintainability, efficiency, and portability.

In addition to these area-specific issues, there is the question of integration of, and synergy between the areas. A strong point of Tangram is that it achieved a couple of interesting results in different areas as listed above, but at the same time this is also a weak point, in the sense that the different techniques are not strongly integrated and linked up with each other in a coherent methodology, yet. Different modeling languages are used in the different areas. Some of them are at least based on the same principle – automata, or state machines, for model-based integration and for model-based testing –, but model-based diagnosis uses a different paradigm of modeling, viz. propositional logic. Also the level of abstraction of the different techniques differs. Industry cannot be expected to adopt a model-based approach for integration and testing, if this requires them to make different models for model-based integration, model-based testing, and model-based diagnosis, and to use completely different tools for these activities. Ideally, it should be possible to reuse models developed for model-based integration for model-based testing, when the realization of such a model becomes available, and for model-based diagnosis, when failures are observed during testing or operational use. Future research and development into a more integrated and coherent methodology for integration, model-checking, testing, and diagnosis is desirable. For integration and test planning the situation is slightly different, because these models are more process-

oriented, whereas the other areas use product models. Here the connection consists of using product models as items in a planning model, e.g., the development and the analysis of a product model are activities that can be planned (see also Chapter 8), potential faulty behavior in a diagnosis model can also constitute a possible fault in a test planning model, and the execution of a test suite generated with model-based testing can be optimally sequenced. The combined use of the same models for model-based testing, model-based diagnosis, model-based monitoring, and model-learning will be one of the topics of the follow-up project, which is initiated by the Embedded Systems Institute.

A second point for a more integrated and coherent methodology concerns the system specification, design, and construction phases. Tangram strongly focused on the integration and testing trajectory, i.e., the right-hand side of Figure 1.1, see also Section 1.3. As already explained in Section 1.2, many integration and testing problems originate from specification, design, or construction faults. This means that sooner or later these phases will have to be considered, too, when studying integration and test problems. Moreover, also in these phases the development and analysis of models may help to solve problems, as the Boderc project of the Embedded Systems Institute demonstrated [58]. This implies that modeling does not start in the integration and testing phase, but that models are developed throughout the system development cycle. Consequently, also modeling during specification and design must be considered when investigating an integrated and coherent, model-based development methodology, such that, ideally, models made during system specification and design can be reused for model-based integration, model-based testing, and model-based diagnosis. (Actually, model-based diagnosis already reuses, to some extent, VHDL design models; see Chapter 12). This coherence should also apply to planning models, where all development activities from requirements capturing until final system testing could be planned.

Equally important when considering a coherent, model-based methodology is the feedback from the integration and testing phases to specification, design, and construction. Errors detected during testing or operational use should be traced back so that design and construction processes can be improved. Moreover, the system design and architecture must take the integration and testing phases into account, and they shall facilitate easy integration and testing, i.e., integratability and testability are system properties which should be designed into the system. This would allow to fundamentally improve the integration and testing process. This future of a coherent, model-based methodology from initial requirements until final system acceptance and further to operational diagnosis, while incorporating both process and product models, requires a lot of further research and development, and some follow-up projects. Tangram has worked on some of the issues related to this, and has made a couple of contributions in this direction. You are invited to read about these contributions in the next chapters of this book.

Chapter 2

ASML: the carrying industrial partner

Author: L. Engels, T. Brugman

2.1 Introduction

The Tangram project aims at a significant reduction of lead time and cost in the integration and test phase of complex high-tech products while maintaining or even improving the product quality. Since ASML goes through intense integration and test phases when bringing their products to the market, the company was happy to be the carrying industrial partner in this 'industry-as-laboratory' project.

2.2 Industry-as-laboratory

Many academic methods and techniques never reach industrial application. The focus of universities is to continuously take the research itself one step further. Professors and Ph.D. students are stimulated for delivering new scientific results rather than preparing their results for industrial application. As a consequence of striving to extend present scientific frontiers, the academic results are often confined to an isolated, restricted domain. Though this approach is legitimate for practicing science, it does not hold a directed way for bringing academic results towards industrial application.

While the industry is capable of organizing the research required for doing their core business, they often fail to do this for adjacent domains. Being confronted with their day to day operations, they hardly find the time to even investigate which academic results – especially non core – might very well fit with their business. Though this way of working might very well result in prosperous business results, it lacks the

guided route for incorporating academic methods and techniques in industrial application.

The ‘industry-as-laboratory’ setting brings down these walls when academia and industry cooperate in bringing scientific results to industrial maturity. The academic partners find their challenge in maturing their methods and techniques for industrial applicability while focusing on real life problems in real life industrial environments. In Tangram this real life environment is delivered by ASML, the ‘carrying industrial partner’ (CIP).

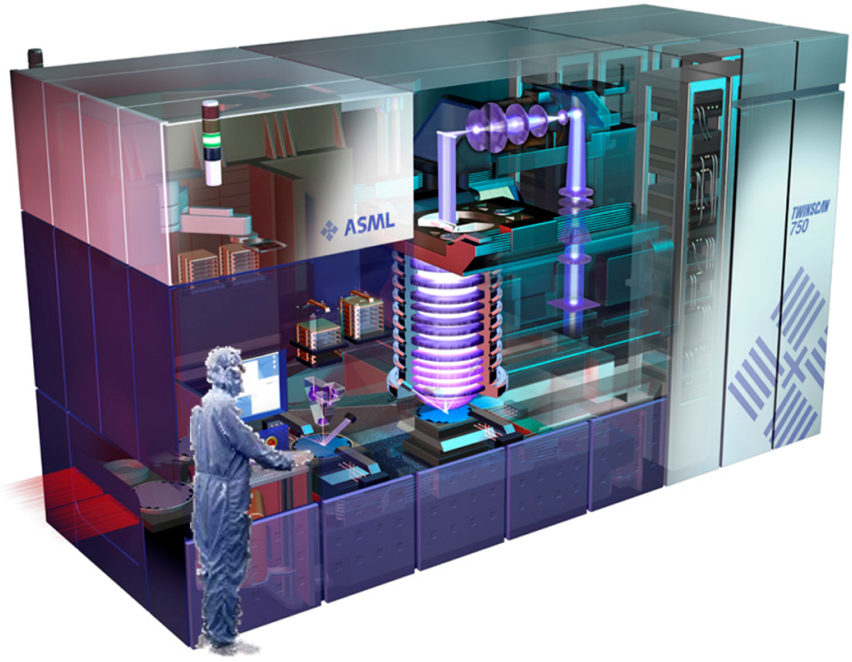


Figure 2.1: A wafer scanner of ASML.

2.3 About ASML

ASML is the world’s leading provider of lithography systems for the semiconductor industry, manufacturing complex machines that are critical to the production of integrated circuits or chips.

ASML technology transfers circuit patterns onto silicon wafers to make every kind of chip used today, as well as those for tomorrow. The technology needed to make

chips advances as digital products become more pervasive - such as mobile phones, consumer electronics, PCs, communications, and information technology equipment.

With each new generation of chips, personal and business products become smaller, lighter, faster, more powerful, more precise, more reliable, and easier to use. In parallel, the global semiconductor industry is pursuing its long-term road-map for imaging ever-finer circuit lines on silicon wafers.

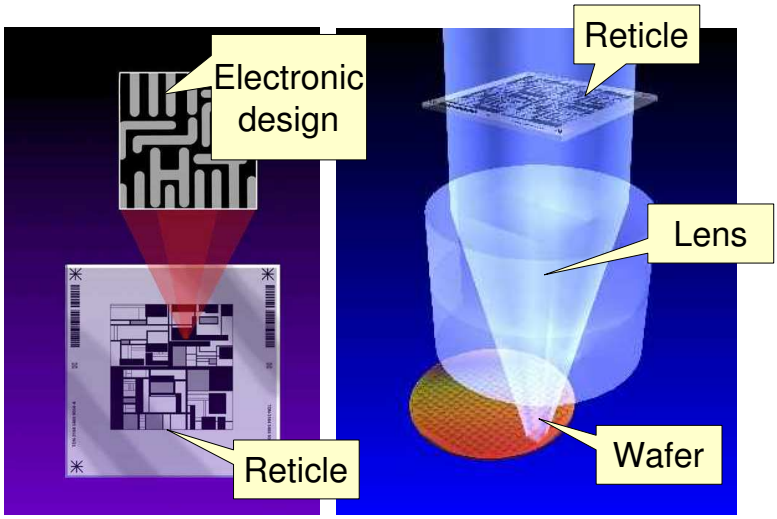


Figure 2.2: From circuit pattern to chips on the wafer.

Core business: lithography

The technology behind ASML's business is known as lithography. ASML has always been at the leading edge of the industry. ASML systems – called steppers and step-and-scan tools – use a photographic process to image circuit patterns onto silicon wafers, much like a camera prints an image on film (see Figure 2.1).

Light generated by a source, such as a laser, is transmitted through a pattern known as a reticle and then through a lens (see Figure 2.2). This process projects an image of the pattern onto the wafer, which has been coated with a light-sensitive material called photo-resist. The wafer is then developed and one layer of the circuit pattern appears. Other chip making steps follow as you can see in Figure 2.3. Repeated a number of times, the process results in a wafer full of completed integrated circuits.

Eventually, these integrated circuits are packaged and used in all kinds of industries to make the products that people use every day at home, at work and on the move.

Figure 2.4 depicts the main subsystems in the ASML system. The Reticle Handler is the robot for loading and unloading reticles. The Wafer Handler takes care of the

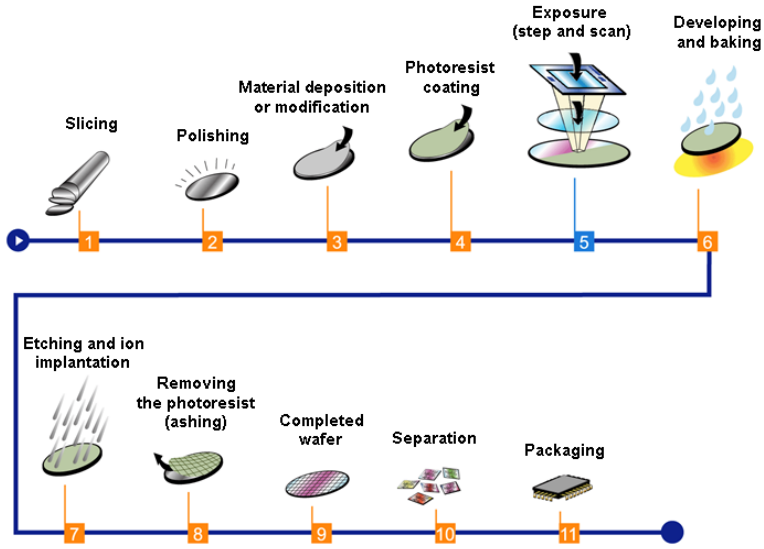


Figure 2.3: From silicon to an integrated circuit: the steps of making a chip. Step 5 is the lithography step. Steps 4 to 8 are repeated several times to create overlays, depending on the complexity of the chip up to 30 times.

loading and unloading of wafers. The Illuminator directs the light such that it passes correctly through the reticle and through the lens onto the wafer. The Reticle Stage controls the movement of the reticle during the expose step. The Wafer Stage controls the movement of the wafer on the measurement chuck as well as the motion on the expose chuck.

Commitment to technology leadership

ASML's largest business focuses on lithography systems for 200 and 300 millimeter diameter wafer manufacturing. The ASML TWINSCAN™ lithography system is the industry's only dual-stage system that allows exposure of one wafer while simultaneously measuring another wafer. Another example is their new immersion lithography system. It replaces the air over the wafer with fluid to enhance focus and shrink circuit dimensions.

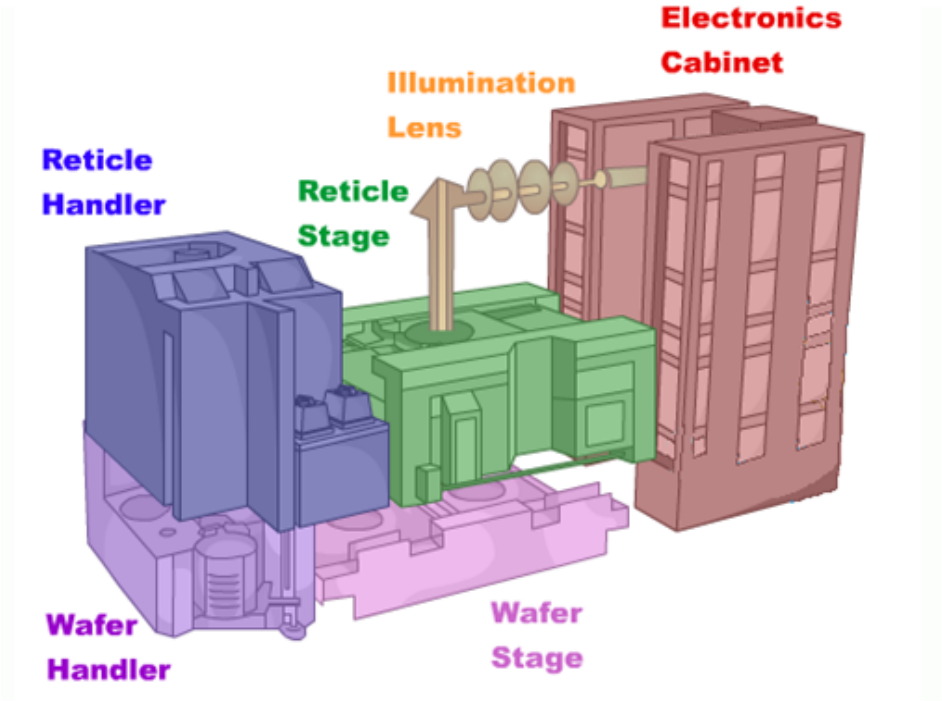


Figure 2.4: Main subsystems of a wafer scanner.

Commitment to customers

ASML researches, develops, designs, manufactures, markets, and services technology systems used by the semiconductor industry to fabricate state-of-the-art chips.

Lithography, or imaging, is the critical technology that shrinks the width of circuit lines, allowing chip makers to continuously design and produce more chips per wafer, more powerful chips or both. Finer line widths (some less than 1,000 atoms across) allow electricity to flow around the chip faster, boosting its performance and improving its functionality. For chip makers, such technological advancements mean increased manufacturing productivity and improved profitability.

ASML is committed to providing her customers with the right technology that is production-ready at the right time. Doing so enables ASML's customers and their customers to sustain their competitive edge in the marketplace.

2.4 ASML's problem statement for Tangram

As stated earlier, the Tangram project aims at a significant reduction of lead time and cost in the integration and test phase of complex high-tech products while maintaining or even improving the product quality. Since it is extremely important to ASML to provide customers with the right technology that is production-ready at the right time, integration and test activities already start during the development phase (see Figure 2.5).

Lead time for shipment t1

It is important for ASML to reduce lead time to shipment t1. The earlier ASML customers get to use the ASML system, the earlier these customers are able to do their part of their process development, i.e., tuning the lithography step with other chip making manufacturing equipment.

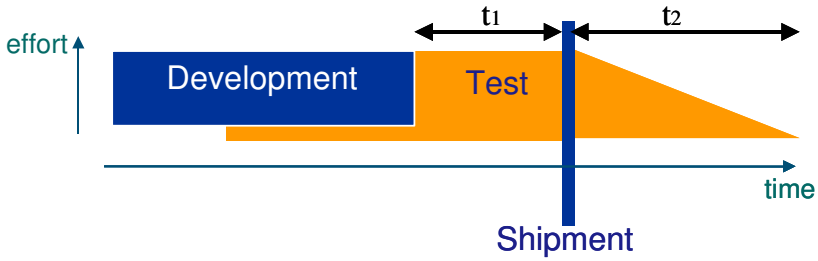


Figure 2.5: The two lead times t1 and t2.

For an ASML system this integration and test involves the confrontation of 10-15 handling-, measuring-, positioning- and imaging subsystems with each other, which all incorporate their part of optics, mechanics, mechatronics, electronics and software. Since ASML has to continuously introduce new - sometimes yet to be invented - concepts (like dual stage or immersion) to meet Moore's law, integration sometimes results in concluding that the different subsystems do not behave as expected. The combination of integrating multi-subsystems and multi-disciplines in yet to be invented domains makes predicting, not to mention reducing, lead time t1 extremely difficult.

Lead time after shipment t2

As soon as the ASML system has been shipped and ASML customers start doing their part of process development, issues in terms of quality and performance will be reported. Even though both customer and ASML perfectly understand that this is inherent to working with state of the art equipment, the ASML customer wants a swift solution to the problem. One could say that in this respect the integration and test phase, now

at ASML customer's site, still continues. The problem with reducing lead time t2 is, compared to reducing lead time t1, more complicated now that the specifics of ASML's customer process have to be incorporated as well. Since ASML has to deal with a variety of customers, predicting and reducing lead time t2 is a difficult multidimensional problem.

Integration and test is at the right hand side of the V-model

ASML's development is organized according to the V-model (see Figure 2.5). While being convinced that there will always be improvements to be implemented on the left hand side side of the V-model, there is an equally strong conviction that these improvements at design- and implementation time never will have an impact such that integration at the right hand side will be completely free of flaws. For that very reason Tangram is to focus on identifying improvements on the right hand side of the V-model only (and leave candidate improvements on the left hand side to others).

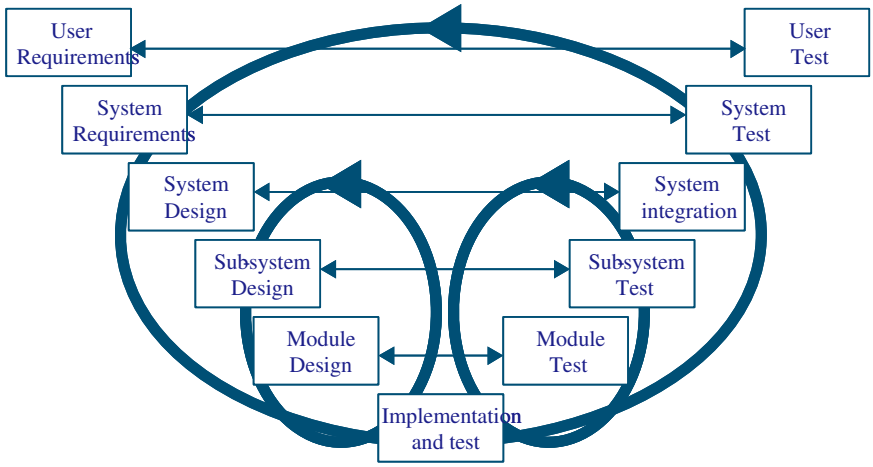


Figure 2.6: The V-model for system development and testing.

A dominant part of the challenge to identify improvements is that there is hardly any formalism or mechanism to reason about integration and test. Where there has hardly been any research on this topic, Tangram is to provide results in this area.

ASML problem statement

To summarize the Tangram problem statement: ASML integration and test must be done faster, cheaper and better. This challenge is never more present than when subsystems from different projects, from different disciplines, and using state of the art

technology, have to be integrated and tested. Where semiconductor business dictates a break-through in this respect, Tangram is to provide an answer for:

- controlling and improving product quality
  - make it specific, measurable, attainable, realistic, timely (SMART) both before and after shipment
- formalizing approaches to deliberate, predict and control lead time t1 and t2
  - make the integration and test plan more predictable
  - moving integration and test activities from the critical path as much as possible
- bringing down overall integration, test and diagnostic costs

2.5 Project results

As will be described in the remainder of this book, significant results have been achieved from an ASML point of view. To summarize these results:

- Tangram demonstrated the capability to reduce lead times with 10-20% by applying model-based test and integration strategies.
- Model-based integration has demonstrated the ability to detect integration flaws several months before real life subsystems come to place.
- Model-based diagnosis has proven the ability to reduce diagnosis time from days to seconds.
- Model-based testing has demonstrated to be too immature to handle real-life, industrial scale applications in the ASML environment. From this observation the concept of Test-Based Modeling has originated; see Chapter 11.
- The test and integration infrastructure has proven to be that effective that ASML applies this infrastructure already in integrating and testing her latest platform.

2.6 Lessons learned

For ASML, some measures have proven to be successful and others turned out to be candidates for improvement.

Benefit in working with half year increments

Working with half year increments has for sure contributed to the Tangram success. Rather than believing in very convincing reasoning and hoping for the best at the end of the project, Tangram decided to focus on tangible results every six months. Especially the first half year increment where every individual partner presented their

interpretation of the problem statement to fine tune their potential improvement, has resulted in experiencing the full extent of the ASML problem and bridged the confusion of tongues amongst the partners.

Benefit in striving for ready-to-use solutions

ASML, being the carrying industrial partner for Tangram, has persisted in ready-to-use solutions at the end of every half year period. ASML would not accept methods and techniques that would fit on paper or would fit in the future; only the proof of concept to a real life problem would satisfy. Even though this attitude has not always been perceived as an easy going approach, it for sure helped in achieving the ultimate result: applying academic results in industrial environments.

Benefit in working on site at ASML

After the first half year increment, Tangram decided to execute the project on the ASML site. Considering that Tangram wanted to bring solutions to real industrial problems, being amongst the industrial people, experiencing their real life problems and getting their immediate feed back has contributed a great deal.

Concern in selecting partners

Tangram has demonstrated that it takes at least six months for all partners involved to experience the full extent of the industrial problem. Having said this, the projects that apply the 'industry-as-laboratory' approach are confronted with a bootstrap problem. To select the partners that fit the problem statement, one should know the problem statement. To get to know all dimensions of the CIP-problem, one should challenge academic partners to present their credentials on the underlying issues. Furthermore it turned out to be very challenging to select research partners that are both competent in the problem domain and willing to invest their key resources to a research project. There is a key role for Embedded Systems Institute to build the expertise required to make the perfect match between industrial problem and both academic and industrial partners.

Concern in recruiting Ph.D. students

Having only the rough contours of a problem statement and not knowing exactly where the scientific contribution will be in solving these problems, it has proven to take a long time to recruit Ph.D. students that match the required profile. It took Tangram 13 months before all Ph.D. students were on board.

Concern in transferring results

Tangram has demonstrated that delivering proof of concepts is, though academically considered very valuable, too poor for providing a basis for roll out in an industrial

organization. After recognizing that the proof of concept – with all its benefits being crystal clear – did not result in a self propelling initiative from ASML, so called transfer projects were initiated (see Chapter 15). The transfer projects delivered ready to use methods, techniques, tooling and training. And although these transfer projects lived outside Tangram, they were an essential element in deploying Tangram results.

Chapter 3

Integration and test planning patterns in different organizations

Authors: I.S.M. de Jong, R. Boumen, J.M. van de Mortel-Fronczak, J.E. Rooda

3.1 Introduction

Planning an integration and test phase is often done by experts. These experts have a thorough knowledge about the system, integration and testing, and the business drivers of an organization. An integration and test plan developed for an airplane is different from the integration and test plan for a wafer scanner. Safety (quality) is most important for an airplane, while time-to-market is most important for a wafer scanner. These important aspects are reflected in the integration and test plan. To investigate the influence of the business drivers on the resulting integration and test plans a number of companies has been visited.

An integration and test plan describes the tasks that have to be performed to integrate individual components into a system. Test tasks are performed in between the integration steps. Note that integration is sometimes called assembly. Integration and testing is performed in early development phases and also in a manufacturing environment.

Business drivers describe what the most important drivers for an organization are. Business drivers are defined in terms of time, cost or quality. The hypothesis is that the order in which business drivers are perceived in an organization determine the way of working and therefore also the integration and test plan.

The goal of the investigation into different integration and test plans at different organizations is to determine what the common elements of such an integration and test plan are. Next to that, the differences are investigated. A number of aspects of an organization next to the business drivers are recorded, like: company size, product volume, number of components in the system, technology used and the sub-contractor model. Note that much of the data is obtained directly from the organization or from publicly available resources. A best guess is made by the authors based on the visits to fill in the gaps.

This chapter, which is based on [66] presented at the 2007 Conference on Systems Engineering Research, is structured as follows. First, the elements of an integration and test plan are introduced. Next, the business drivers and organizational aspects which we consider to be of influence are discussed. Then, the different organizations, business drivers, organizational aspects and integration and test plans are discussed in detail followed by a summary and conclusions.

3.2 Integration and test plans

The *integration and test phase* is the phase in product development or product manufacturing, where components are tested and integrated (assembled) into systems. Components can be tested when component development or manufacturing is finished. Furthermore, components can be tested after each integration phase. An *integration and test plan* determines the order of integration and where testing takes place, that is test phases are positioned in between integration steps.

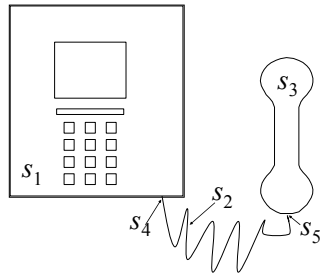


Figure 3.1: Example telephone system.

An integration and test plan is developed before the integration and test phase is started and is often updated when the integration and test phase is executed.

An integration and test plan consists of a sequence of integration and test phases, e.g., the order in which developed components are integrated (assembled) and tested. A test strategy is chosen for each test phase in the integration sequence resulting in a test plan (sequence of test cases) for each test phase.

The elements in an integration and test plan are: Develop *DEV*, Assemble *ASM*, Test *TST*, Disassemble *DAS* and Copy *CPY*. These elements, except Copy, are illustrated using an example system: the telephone system depicted in Figure 3.1. *CPY* is illustrated in Section 3.5.8, when two typical software integration plans are discussed. The three modules in the telephone system, horn (*H*), cable (*C*) and device (*D*), can be integrated using the integration sequence depicted in Figure 3.2.

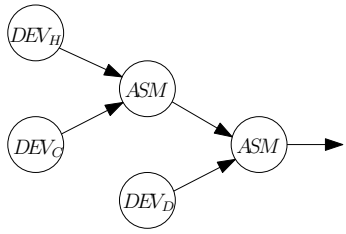


Figure 3.2: Example integration sequence.

The required test phases can now be positioned in between the development (DEV_H, DEV_C, DEV_D) and assembly (*ASM*) phases. In this quality driven strategy[1], a test phase is planned for each developed component and after each assembled component. Figure 3.3 shows the resulting integration and test sequence. Note that a time-to-market driven strategy[2] may result in an integration and test sequence where some of the test phases are skipped (not depicted in Figure 3.3).

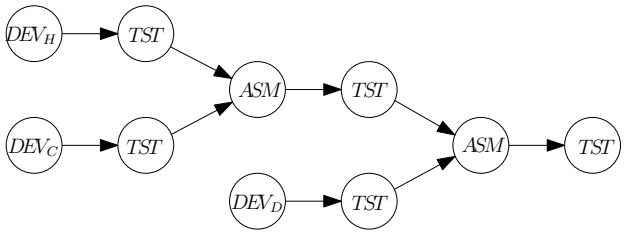


Figure 3.3: Example integration and test sequence.

A replacement of the horn can now be modeled by a disassembly of horn 1 (DAS_{H_1}) and an assembly of horn 2 (ASM_{H_2}). Figure 3.4 illustrates the disassembly of Horn 1, followed by the assembly of Horn 2. For this, two horns are developed in DEV_{H_1} and DEV_{H_2}.

[1] A quality driven strategy is a strategy that reduces the risk after every development or assembly step as much as possible by performing test phases. The probability that the final quality is less than expected is minimal by removing risk as early as possible.

[2] The goal of a time-to-market-driven strategy is to integrate and test the product as fast as possible with as less quality loss as possible. Some testing is performed in between assembly steps, but not everything is tested. Priority lies with the progress of integration.

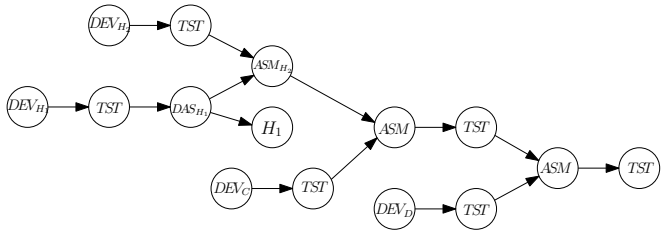


Figure 3.4: Example integration and test sequence with disassembly and assembly of the horn.

The test strategy for each of the test phases can now be chosen, resulting in a test plan for each test phase T. A single test strategy for all test phases could also be chosen.

A quality driven integration and test plan requires that all test phases are executed with a quality driven strategy. The selected elements of the strategy are:

1. Test sequence: execute all test cases, fix the detected faults and re-execute the test cases;
2. Stop criterion: all risks must be removed in each test phase;
3. Test process configuration: execute test cases first, followed by diagnosis and fixing the detected faults.

Many different integration and test plans can be obtained for a single system by varying integration sequences, test strategies and test phase positioning strategies. Different organizations often use a specific integration and test planning *method* resulting in similar integration and test plans for similar products.

3.3 Business drivers

Business drivers are the requirements that describe the goal of an organization. The business drivers *Time*, *cost*, and *product quality* are known from manufacturing management [76, 97]. We will use these business drivers to characterize the investigated organizations.

An organization with time as the key business driver is focused on delivering products as quickly as possible to the market. An organization with cost as key business driver is focused on delivering products as cheaply as possible to the market. Finally, an organization with product quality as key business driver is focused on delivering products to the market which satisfy the customer as much as possible.

The order of importance determines the way of working in the organization. For example, an organization with T-C-Q (Time first, cost second and quality least important) as business drivers delivers products of different quality and production cost than an organization operating with T-Q-C as business drivers. Both deliver products as

quickly as possible to the market. The first organization develops, manufactures and services these products as cheaply as possible. Product quality is least important. The focus of the second organization is on product quality (next to fast delivery). Cost is least important.

3.4 Organizational aspects

The integration and test plans of very different organizations were investigated. Sometimes a specific department was visited. The observed integration and test plan was probably only one of the forms in that organization, while the business drivers are for the entire organization. Therefore, additional aspects of the organization are also recorded to determine the possible effect of these aspects. A number of organizational aspects that are recorded are:

1. The number of products shipped per year or number of end-users influences the required product quality and maintenance cost.
2. More complex products result in more complex integration and test plans. Complexity can be the result of many components, resulting in many integrations and possible test phases. Complexity can also be the result of the use of complex technology resulting in complex test cases.
3. Using many different sub-contractors for the development of components could result in many additional test phases to qualify the delivered components. Next to that, political aspects could result in additional test phases. For instance, sub-contractor test cases could be repeated to accept the delivered products, resulting in additional test phases. The other way around is also possible.

3.5 Investigated organizations

A number of different organizations have been visited to investigate the influence of business drivers on integration and test plans.

A summary is given for each of the investigated organizations. The order of business drivers indicates the relative importance of the business driver, i.e., T-Q-C means that time-to-market is most important followed by quality and least important is cost. T-Q/C means that quality and cost are equally important. The order of the business drivers is determined by the authors after the visit or investigation. Next to that, relevant information like company size, product volume, number of components, technology used and the number of sub-contractors was recorded.

3.5.1 Semi-conductor (ASML and others)

A typical semi-conductor equipment integration and test sequence (Figure 3.5) consists of development phases (*DEV*) executed at suppliers, followed by a supplier qualifica-

Company size	Medium, 5000 employees
Product volume	200-300 systems/year
Business drivers	T-Q-C
Number of components	large / very large
Technology used	New technology
Sub-contractors	Many, cooperating

Table 3.1: Semi-conductor equipment manufacturer characteristics.

tion test and a system assembly phase (ASM). The assembly phase of each system is followed by two test phases: the calibration test TST_C[3] and acceptance test TST_A[4]. Chuma [38] investigated the duration of the assembly phase (ASM) and the durations of TST_C and TST_A for lithographic equipment manufactured at ASML, Canon and Nikon[5]. The average duration of the assembly phase is 9.8 days while the average duration of the calibration and acceptance tests in 2005 are 34.5 and 32.5 days, respectively, according to the report.

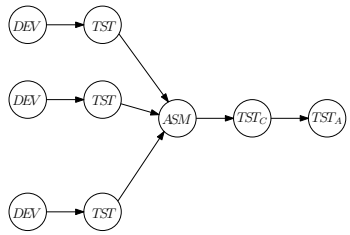


Figure 3.5: Typical semi-conductor manufacturing integration and test plan.

ASML develops semi-conductor equipment using platforms. The integration and test plan of a new system-type of a new platform is developed specifically for this system (See product development later). Subsequent system types in a new platform are integrated and tested based on a previous system type. First, a previous system type is manufactured as in Figure 3.5. New subsystems are developed. The old sub-systems are replaced by the new versions. Figure 3.6 depicts this integration and test plan. The previous system type is assembled after the first assembly step. Modules, like M_1, are disassembled (and re-used) and a newly developed module M_1' is assembled. Module M_2 is replaced similarly by M_2'.
A typical aspect in this time-to-market driven organization is that the newly developed sub-systems M_1' and M_2' are not tested thoroughly. Integration progress is more impor-

[3] A calibration test phase is a test phase where test cases and calibration tasks are interchanged. Test cases are executed on the system to determine the performance of the system. If the system is 'out of specification', calibrations are performed and testing continues.

[4] An acceptance test is the test executed to determine if the customer accepts the system.

[5] ASML, Canon and Nikon are the main suppliers of lithographic equipment to the semi-conductor market.

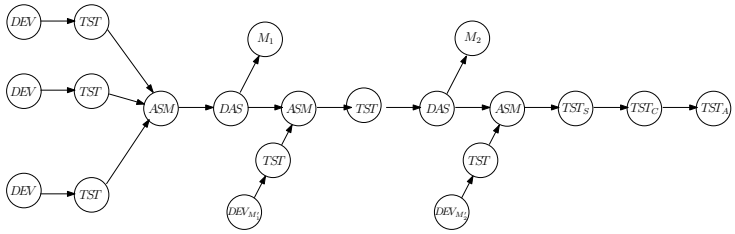


Figure 3.6: Semi-conductor development integration and test plan.

tant than the qualification of sub-systems. Remaining risk in the system is covered in higher level (later) test phases. The final acceptance test is a combination of a thorough system level design qualification TST_S and the normal final calibration and acceptance test phases TST_C and TST_A. The test cases in the final test phases TST_S, TST_C, and TST_A are often mixed such that a faster test sequence is obtained.

3.5.2 Automotive

Company size	large, 30000 employees
Product volume	100000 systems/year
Business drivers	C-T/Q
Number of components	Medium
Technology used	Proven technology
Sub-contractors	Many, cooperating

Table 3.2: Automotive manufacturer characteristics.

A typical assembly line (Figure 3.7) for cars consists of a number of assembly steps (A) followed by a short final acceptance test phase T_A. Suppliers develop (manufacture) and test the parts which are assembled into a car. Testing is standardized and focused on quality (for instance measurement techniques for electrical systems are described in IEC 61508 Part 7 [64]).

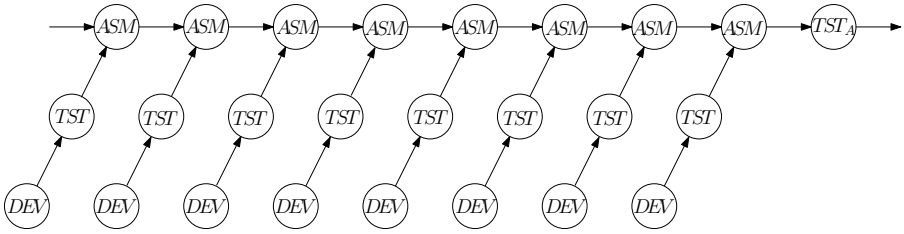


Figure 3.7: A typical 'assembly-line' for cars.

3.5.3 Communication

Company size	large, 30000 employees
Product volume	120000000 systems/year
Business drivers	Q-C/T
Number of components	Small
Technology used	Proven technology and new software
Sub-contractor	Few/none

Table 3.3: Communication equipment manufacturer characteristics.

A mobile phone communicates with other mobile phones via the (GSM/GPRS/3G) network. An estimated 120000000 mobile phones have been shipped in the USA only in the year 2005 [63]. The estimated number of shipped units in 2011 is 1.25 billion worldwide. The communication protocol between a mobile phone and the infrastructure is standardized [43]. A single test phase of a few weeks qualifies if a mobile phone operates according to the standard. The visited organization developed such a standard test set, which is used by different mobile phone developers. This test phase is repeated if problems are found and fixed until the phone operates according to the standard. A specific example of this re-test phase with three test phases and two diagnose and fix phases (DF) is depicted in Figure 3.8.

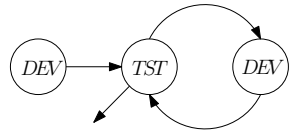


Figure 3.8: Specific example of a mobile phone test phase.

3.5.4 Avionics/DoD

Company size	large, 30000 employees
Product volume	300 systems/year
Business drivers	Q-C-T
Number of components	High
Technology used	Proven technology
Sub-contractors	Many, regulated

Table 3.4: Avionics/DoD manufacturer characteristics.

Airplanes and systems developed for the department of defense (DoD) are integrated and tested using a strict process, like for example the integration and test process for the

777 flight controls [35]. All sub-systems are tested in the supply chain to ensure a short final test phase. To accommodate this, interfaces between sub-systems are thoroughly described and do not introduce new problems. An integration and test plan for an airplane or DoD system is similar to the plan depicted in Figure 3.5. Sub-systems are tested completely before integration. The duration of the final calibration test phase T_C for an airplane, like an Airbus A320, is only a few days, including a test flight. Assemblies are performed in between the final calibration test phase and acceptance test phase. For instance, the engine of an airplane is assembled when all other parts have been assembled and calibrated. The reason for this is safety and cost. Assembling an engine is done in a special area and the engine is costly, so it is assembled as late as possible.

3.5.5 Space (satellites)

Company size	medium, 5000 employees
Product volume	10 systems/year
Business drivers	Q-C-T
Number of components	Medium
Technology used	Proven technology
Sub-contractors	Few, cooperating

Table 3.5: Space/satellite manufacturer characteristics.

Development of a satellite or another space vehicle results in a single system which is delivered to the customer. Each system is unique. The integration and test plan is very similar to an integration and test plan of a newly developed system. The assembly phases are executed as concurrently as possible. Test phases are planned after each development and each assembly phase such that the risk in the system is minimal at all times. An overview of international verification and validation standards for space vehicles, including the main differences between standards, is described in [52]. A planning and scheduling method for space craft assembly, integration and verification (AIV) is described in [3].

3.5.6 Machine builders

A number of machine building organizations has been visited. The developed systems varied from manufacturing equipment to large office equipment. A variety of integration and test plans has been observed in the different organizations. Most of the organizations use an integration plan which is similar to the plan used in the automotive industry. Some use a fix-rate assembly, e.g., each assembly step is performed in a fixed 20 minute time slot by a single operator. Some calibration tests are performed in between assembly steps. Configuring the system for a customer is done just before the

Company size	Medium, 5000 employees
Product volume	1000 systems/year
Business drivers	C-Q-T
Number of components	Medium
Technology used	Proven technology
Sub-contractors	Many, cooperating

Table 3.6: Machine manufacturer characteristics.

acceptance test T_A. An example of a sequence with a customer specific configuration in the last assembly step is depicted in Figure 3.9.

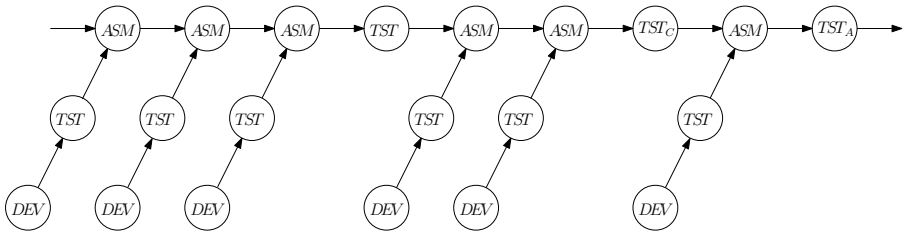


Figure 3.9: Example manufacturing sequence for machine builders.

3.5.7 Drug industry

Company size	large, 20000 employees
Product volume	Millions of tablets/year
Business drivers	Q-C-T
Number of components	Small
Technology used	New technology
Sub-contractors	None

Table 3.7: Drug developing company characteristics.

The drug testing industry is discussed based on [105, 106]. The type of products in this industry is different compared to the technical products as discussed before. Testing of medical drugs is also quite different. Figure 3.10 depicts an integration and test plan for medical drugs.

The development of a potential new drug is a combination of chemical design and a structured search. The integration and test plan starts if a new chemical entity (NCE) is discovered. A screening test (TST_S) is performed to test the potential of the new chemical. The new chemical is then 'integrated' into tablets (DEV_T) or dissolved in liquid (not depicted). What follows next are four test phases in which the new drug is

tested (TST_A, TST_I, $TSTII$, TST_{III}). The average total duration of the entire plan is 14 years. Test phase TST is performed on animals to test for toxicity and long term safety. Test phase TST_I is performed mainly on healthy volunteers to determine the dose level, drug metabolism and bio-availability[6]. Test phase TST_{II} is a test phase on a few hundred patients to test the efficacy of the dose and the absence of side effects. Test phase TST_{III} is performed to test efficacy and safety on thousands of patients. Test phase TST_{IV} is performed after the new drug has received a product license to test for rare adverse events and to gain experience with untested groups of patients.

The conclusion of every test phase can be that testing will not be continued. The new drug will not be further developed and released, in contrary to the (technical) products of the other organizations which can be fixed.

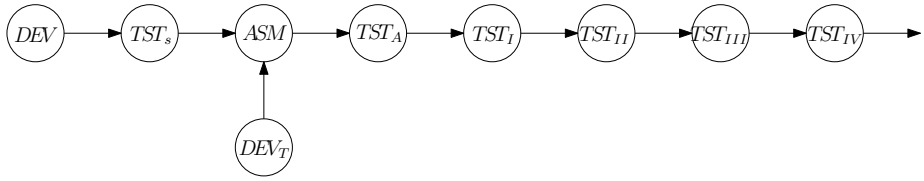


Figure 3.10: Integration and test plan for medical drugs.

3.5.8 Integration and testing of software baselines

A special case of an integration and test plan for product development is an integration and test plan for software developments which are delivered into a single code base. All code ends up in a configuration management system. Testing is done on the code before delivery and on the 'release', a specific baseline in the configuration management system. Two example integration and test plans are discussed. These types of integration and test plans have been encountered at several visited companies, including ASML. Next to that, Cusumano describes a similar integration and test plan as used by Microsoft [39]. The first example plan, depicted in Figure 3.11, contains a periodic test phase. Integration continues when the test phase passes.

The second example plan, depicted in Figure 3.12, contains a periodic test in parallel with integrations of new code. A copy (CPY) of the software is made and used to test the (copied) software.

The test phase in the periodic case is on the critical path, while the test phase in the parallel case is not. On the other hand, problems found in the periodic case are solved before new integrations are performed. Problem solving in the parallel case is more complex, because two baselines are to be maintained at any point in time. This is depicted in Figure 3.12 with an explicit 'self-loop' on the test process and an explicit assembly of solutions into the baseline.

[6]How (and how fast) is the product entered in the body, bloodstream and excreted from the body.

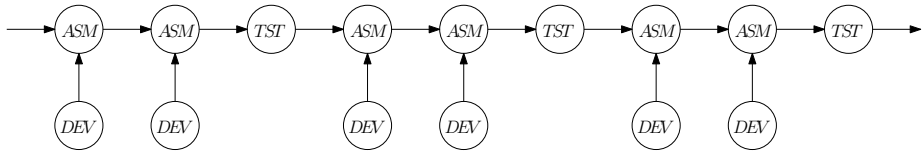


Figure 3.11: Software integration with periodic test phases.

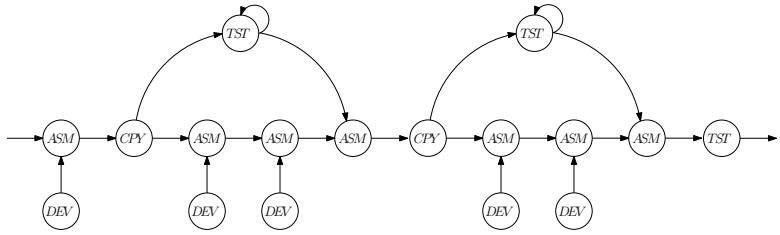


Figure 3.12: Software integration with parallel test phases.

3.6 System complexity versus planning approach

An overview of the organizational types and their influence on an integration and test plan is depicted in Figure 3.13. The organizational types can be found in Table 3.8. Each circle indicates an organization which has been visited or otherwise investigated. The size of the circle indicates the size of the organization (large circles correspond with large organizations). The gray tone of the circle indicates the number of delivered end-products. A darker circle indicates more shipments. Each circle contains the key business drivers (in order) for the visited organization. The organizations are placed in the graph in Figure 3.13 according to the integration and test planning approach on the x-axis (regulated or flexible) and the system complexity on the y-axis. The complexity is a combination of number of components and technology used. The type of organization is described in the bottom half of the circle. In some cases, multiple organizations of the same organizational type have been investigated. All investigated organizations are depicted in Figure 3.13.

A distinction is made between a regulated approach and a flexible approach. The strategy of a *regulated approach* is focused on removing all risk as soon as possible. Consequently, test phases are planned after each development and assembly action. The focus of each test phase is on removing all possible risk. The *flexible approach*, on the other hand, is focused on maximal integration progress. Test phases are planned after some of the development and assembly actions. These test phases are partially executed and the remaining risk is covered by a later test phase.

The flexible approach allows the optimization of test phases by moving test cases from one phase to another phase. The regulated approach prescribes that specific test

cases need to be performed in a specific test phase. Optimization of a test phase can only be done within the test phase itself. The organizations which are visited are grouped according to the complexity of the product and the use of a regulated or flexible test approach.

Semi	Semi-conductor equipment
Avionics	Airplanes
Space	Satellites
DoD	Department of defense systems
Drugs	Medical drugs
Comm	Communication equipment
Machines	Machine equipment

Table 3.8: Legend of organizational types.

3.7 Conclusions and discussion

Different organizations use different integration and test plans to develop or manufacture their products. The elements of an integration and test plan are the same for all investigated organizations. The key business drivers of an organization can be characterized by Time, Cost and Quality. An integration and test plan is specific to an organization, the product and the business drivers.

As a result, it can be concluded that a strategy to obtain an integration and test plan for a specific organization cannot be copied to another organization just like that. The business drivers of both organizations should match.

Two types of test approaches are distinguished: regulated and flexible plans. Flexible integration and test plans are used in time-to-market driven organizations, whereas regulated integration and test plans are used for other organizations. The main differences between a regulated and flexible integration and test plan are 1) the positioning of test phases and 2) the type of test strategy which is used for each of the test phases.

Optimizing an integration and test plan could be beneficial in terms of time, cost and quality. A flexible integration and test plan allows many optimization opportunities. Among these are the selection of an integration sequence, a test sequence, the selection of a test positioning strategy, and the selection of a test strategy per test phase.

A regulated (fixed) integration and test plan consists of a regulated integration sequence. Selecting a different (better) sequence is difficult. The cost of changing the regulations should be taken into account. This is also the case for the test positioning strategy and the chosen strategies for specific test phases.

The benefit of a regulated integration and test plan is that these plans are easier to plan and control. All parties involved know from the start what to expect and what to do. The test content is known in advance for all test phases. The benefit of a flexible integration and test plan is that the plan allows for more optimization techniques to

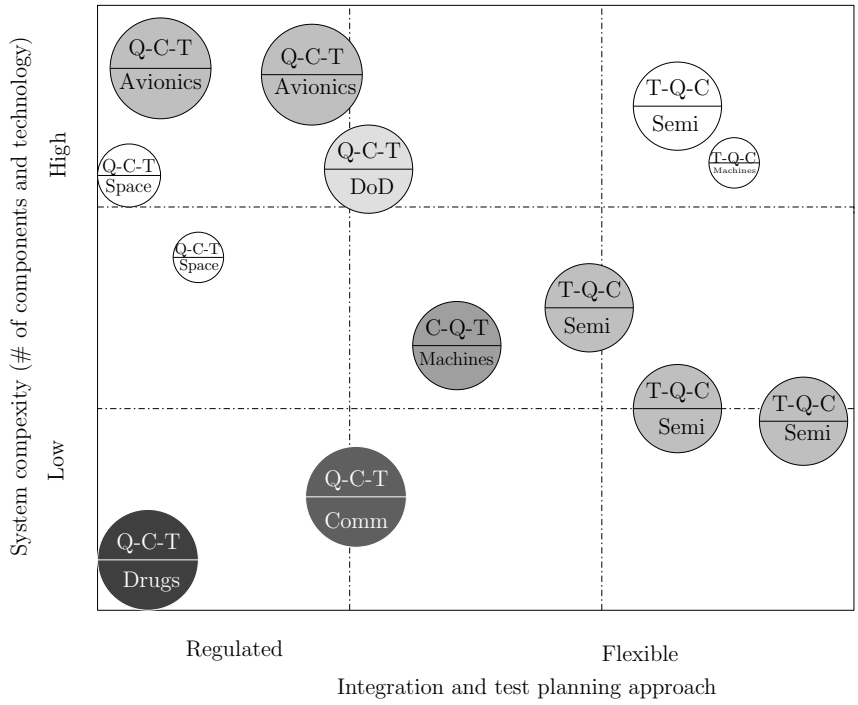


Figure 3.13: Overview of the visited organizations by system complexity and test strategy.

obtain a better plan. The cost of this flexibility is the organizational effort which is involved in the optimization cycle.

A combination of a regulated integration and test plan with known 'control' points in the plan and flexibility in the intermediate phases could be a good solution for organizations that either try to increase the quality levels and maintain the short time-to-market or organizations that try to reduce the time-to-market while maintaining the product quality.

Chapter 4

Integration and test planning

Authors: I.S.M. de Jong, R. Boumen, J.M. van de Mortel-Fronczak, J.E. Rooda

4.1 Introduction

The integration and test phase of a newly developed system is often one of the most hectic phases in system development. The initial integration and test plan changes on a daily basis. Newly developed components are delivered late, problems are found during testing and components in the system break down causing delays and additional cost. The integration and test planning experts update and change the plans to accommodate for these changes, such that the project target is met. The quality of the integration and test plan is highly dependent on the capabilities and system knowledge of the integration and test planning experts. Clear methods to develop an integration and test plan are lacking. Moreover, the company visits, described in Chapter 3, indicated that integration and test planning is performed differently at each organization visited. Not only the knowledge of the integration and test planner is relevant, also the type of organization and business drivers are of interest. Industrial efforts to overcome integration and test problems often have a limited scope. Only a part of the integration and test plan is improved. Theoretic approaches to solve integration and test problems have an even more limited scope. The limited scope of the industrial and academic approaches results in solutions with limited impact on the total integration and test duration, cost and remaining risk. The remaining risk in the system is our measure for product quality.

The work presented here describes an integration and test planning method. Three strategies are used in this method to create an integration and test plan: the integration strategy, the test positioning strategy and the test strategy. Several feedback loops are added in this planning method that describe how an integration and test plan can be improved if the required duration, cost or remaining risk is not reached. The method

presented is applicable for systems in several disciplines and also for multi-disciplinary systems, since the method does not apply specific techniques from a single discipline that are not applicable in another discipline. The presented integration and test plan-

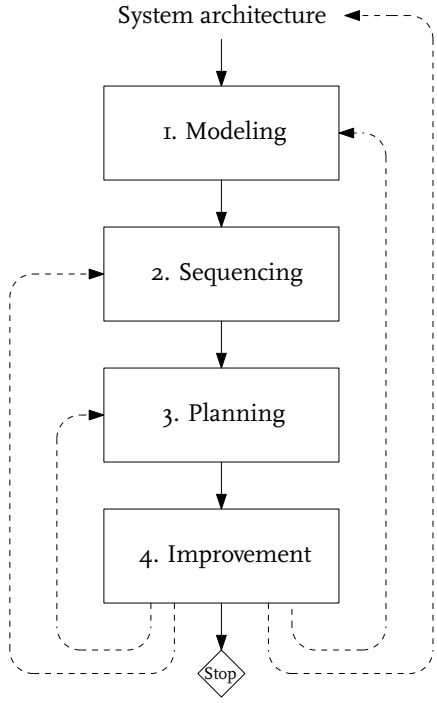


Figure 4.1: Overview of the integration and test planning method.

ning and improvement method, depicted in Figure 4.1, consists of the following four elements: modeling the system architecture, sequencing, planning and improvement. Modeling is described in Section 4.2. Section 4.3 describes techniques for the sequencing task. Section 4.4 describes techniques for test planning. And, improvement, is described by Section 4.5. This chapter ends with conclusions.

The figures that describe the tasks in the integration and test planning method indicate each task with a number, a period, and a number or character. The first number indicates that the task is part of the higher level modeling, sequencing, planning or improvement step: 1 through 4, respectively. The number or character indicates a unique task in the method. Numbers are used for sub-tasks and characters are used for inputs and results. A box is drawn around the sub-tasks that are part of the step in the method that is described in that section. The sub-tasks that are not part of the box are part of another step and are described in another section. Feedback loops (improvement tasks) are depicted with a dashed line. These improvement results are not mandatory to start a task. A solid line indicates a mandatory input to start the task.

4.2 Modeling the system architecture

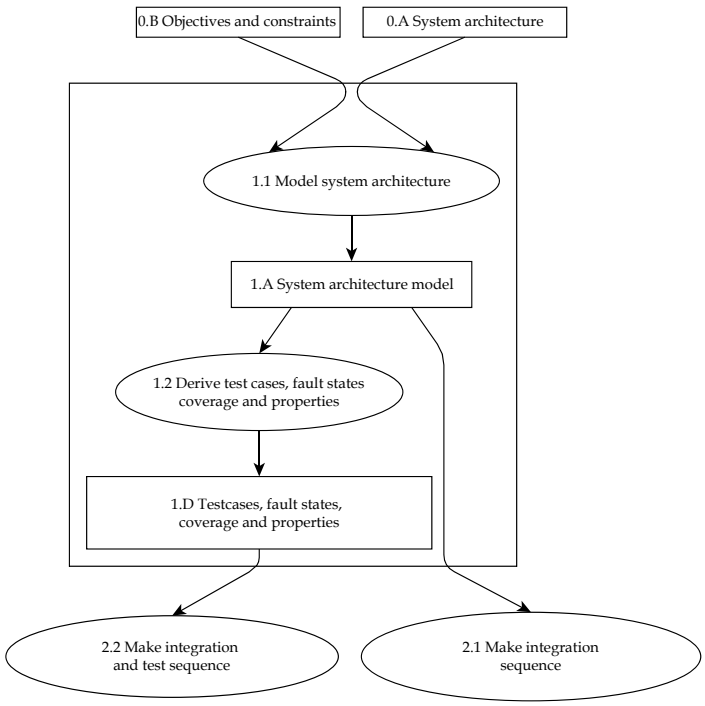


Figure 4.2: Overview of the modeling step.

The first step in the integration and test planning method is the modeling step. In this modeling step, the *system architecture* is modeled for integration and test planning. Inputs to this step are the system architecture (0.A) and the initial (or system wide) objectives and constraints (0.B). The *system architecture model* (1.A) consists of *components* C, *interfaces* X_F, a *layering*, $\mathcal{L}$, the *delivery timing* of the components $timing$, and the *objectives* Obj: $A = (C, X_F, \mathcal{L}, timing, Obj)$. The components are connected with each other through their interfaces. A broad definition of interfaces is considered. For example, physical interfaces and bolts and screws determine if two components can be connected. The same holds for software interfaces over a Corba [92] or a publish-subscribe [90, 13, 95] network. In general, all interfaces which possibly influence the integration and test plan need to be taken into account. For instance, a wafer scanner images a pattern of a part of an integrated circuit on a silicon wafer. This is done by projecting laser light with a specific dose and a specific uniformity on a wafer. The path that the light follows through the wafer scanner is considered an interface between the laser source and the silicon wafer.

The *layering* groups components. The integration and test planning process is sim-

plified by this layering, because first the components within a layer are assembled and then the layers are assembled. This reduces the number of integration plans that can be made, since not every assembly task is possible anymore.

The *delivery timing* of each component and interface determines when integration and testing can start. The *objectives* of the integration and test plan are expressed in terms of maximal duration, maximal cost and minimal product quality. These objectives are needed for the evaluation of possible integration and test plans. The impact of the objectives on an integration and test plan is illustrated in the Chapter 3.

Test cases, fault states and coverage

The goal of *integration* is to assemble a system from the required components and interfaces. An *integration plan* therefore consists of a sequence of assembly tasks that connect components using their interfaces. The remaining risk after integration is the sum of the component risk and the interface risk, when no testing is performed in between the assembly tasks. Executing test cases, diagnosing problems and fixing faults reduces the risk in the system and by this the product risk is decreased. The elements in the system architecture model, 1.A, are used to create an integration plan. The reduction of risk (by testing) requires two additional models: the *system test model* and a model that relates the *system test model* with the *system architecture model*. Both models are introduced briefly.

The *system test model* describes the set of available test cases, T, the possible faults in the system and the coverage of the test case on these faults. The possible faults in the system are modeled as a set of *fault states*: S. The coverage of a test case on a fault state, modeled as R_{ts}(t,s), describes either what the coverage is of test case t on fault state s, or the probability that fault state s is discovered by test case t. It depends on the application how the coverage is modeled. The *system test model* D is defined as : $D = (S, T, R_{ts})$. Additional properties per test case and fault state describe for instance the duration of a test case or the failure probability and impact of a fault state. The failure probability and impact are used to determine the risk of a fault state and the risk covered by a test case. More details about the system test model can be found in Chapter 6. The modeling process for a newly developed system starts with a failure mode effect analysis (FMEA, FMECA) [25, 44] to determine the possible failure modes in the system. This FMEA can be according to a strict FMEA-process or a brief investigation of failure modes. These failure modes are the fault states in the test model. The failure modes in an FMEA are normally followed up with *counter measures*. These counter measures are the test cases in the test model, if the counter measure is a test[1]. The coverage of a test case on all fault states (components and interfaces) is estimated as well as the properties of test cases and fault states.

A *system test model* is used to describe the details of the components and interfaces in the system. A component is seen as a set of fault states. This is also the case for

[1]Some counter measures in an FMEA analysis are design changes or organizational measures instead of tests. Only test cases are relevant to the integration and test model.

an interface. The components and interfaces are related with the system test model by two models: $R_{DC}$and R_{D,X_F}. This relation is used to determine which test cases can be performed when a component becomes available and also to determine the risk of a component or interface.

As an example, a test model is defined for a common telephone. The telephone consists of a handset, a cable and a device. A graphical view of the telephone is given in Figure 4.3.

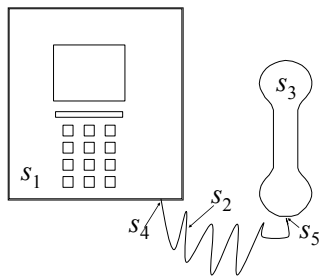


Figure 4.3: Telephone system.

The example telephone system consists of the following fault states S:

1. s_1 the device can be unreliable
2. s_2 the cable can be unreliable
3. s_3 the handset can be unreliable
4. s_4 the interface between the cable and the device can be unreliable
5. s_5 the interface between the handset and the cable can be unreliable

The test set, T, consists of the following test cases:

1. t_0 tests the complete phone system
2. t_1 tests the device
3. t_2 tests the cable
4. t_3 tests the handset
5. t_4 tests the device and the cable
6. t_5 tests the handset and the cable

S/T	t_0	t_1	t_2	t_3	t_4	t_5	P
s_1	0.01	0.02	0	0	0.015	0	0.6
s_2	0.01	0	0.02	0	0.02	0.015	0.1
s_3	0.01	0	0	0.03	0	0.02	0.2
s_4	0.01	0	0	0	0.015	0	0.4
s_5	0.01	0	0	0	0	0.015	0.3
C_T	1	1	1	1	1	1	

Table 4.1: A reliability test model for the telephone example.

A simple matrix representation of the *system test model*, including the properties per fault state and test, is given in Table 4.1. The column indicated with P describes the failure probability of the test case and the row indicating C_T describes the test duration.

A screen dump of the system test model of the telephone from the integration and test planning and improvement tool set LONETTE is given in Figure 4.4.

	Faultstates	0	1	2	3	4	5					
Tests		t_0	t_1	t_2	t_3	t_4	t_5	Probability (%)	Impact	Fix cost [euro]	Fix time [h]	Fault state risk
1	s_1	0.01	0.02			0.015		60	1	0	0	0.6
2	s_2	0.01		0.02		0.02	0.015	10	1	0	0	0.1
3	s_3	0.01			0.02		0.02	20	1	0	0	0.2
4	s_4	0.01				0.015		40	1	0	0	0.4
5	s_5	0.01					0.015	30	1	0	0	0.3
	Test Time [h]	3	1	1	1	2	2					
	Test Cost [euro]	3	1	1	1	2	2					
	Diagnose Time [h]	5	2	2	2	4	4					
	Diagnose Cost [euro]	5	2	2	2	4	4					
	Repeatable	Yes	Yes	Yes	Yes	Yes	Yes					
	Test Risk	0.02	0.01	0	0	0.02	0.01					
	Fail Probability (%)	2	1	0	0	2	1					
	Information Gain	0.1178	0.0938	0.0208	0.0376	0.1238	0.0806					

Figure 4.4: Screendump of the telephone model.

The *system test model* can be used to calculate the risk per test case, the risk after assembly and the remaining risk after testing. The same system test model can be used for the definition of test sequencing techniques and the comparison of test sequencing techniques. Moreover, the system test model is used for improving the integration and test plan in Section 4.5.

The model that describes the relations between the system test model and the sys-

tem architecture model consists of two elements: a relation between components in the system architecture model and the fault states in a system test model and a relation between interfaces in the system architecture model and the fault states in a system test model. The set of fault states can be obtained for each component and interface in the system architecture model this way.

4.3 Integration sequencing

The goal of integration sequencing is to form a sequence of integration tasks, which results in a complete system. Furthermore, test tasks are positioned in between the integration tasks. The start moment and duration of these test tasks is determined by the position in the integration sequence. The components which are integrated at a certain moment in the sequence determine which faults can be present in the system. The performance of the resulting integration sequence is measured in terms of time (duration), cost and remaining risk.

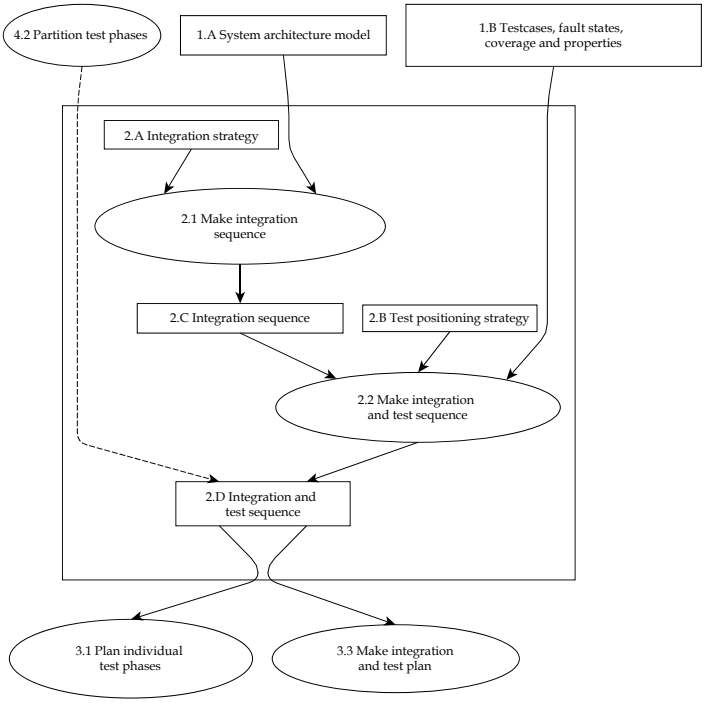


Figure 4.5: Overview of the sequencing step.

An overview of the sequencing step is given in Figure 4.5. An integration sequence is created in 2.1 using the system architecture model 1.A. Then, test tasks are positioned

in this integration sequence in step 2.2. The integration strategy and test positioning strategy are inputs for the integration and test sequencing step. Both are explained below.

An *integration strategy*, 2.A, is a specific approach which is used to create an integration sequence. Common examples of an integration strategy are known as *bottom-up integration*, *top-down integration* and *big-bang integration*. Other integration strategies originate from concurrent engineering and baseline development. Concurrent engineering and concurrent development results in an integration strategy (and plan) that is executed as concurrently as possible. Baseline development integrates components in a baseline system. A complete baseline system exists before integration of new components can be started. An older version component is replaced with a newly developed component. A baseline development integration strategy is often seen in software development, where the entire code base is available from a previous release in a configuration management system and new code is integrated into it. This strategy is also followed for the development of a new type of wafer scanner which is based on a previous system type. The previous system type is built first. Then, new components replace existing components in the system and parts of the performance test cases are re-executed. This last strategy is a strategy enforced by a standard, framework[2] or policy. An overview of this type of integration standards, frameworks and policies can be found in [113].

A *test positioning strategy* is the second strategy used for integration sequencing and positions test tasks in between development and assembly activities. This strategy determines how risk is reduced by executing test cases. Test cases that *pass* reduce the failure probability of the covered fault states. Test cases that *fail* require a diagnosis and eventually a fix of the fault state causing the fail. Fixing a fault state also reduces the failure probability. A reduction of the failure probability reduces the risk in the system. Whereas the integration strategy determines how components are assembled and, consequently, how risk is built up, the test positioning strategy determines when risk is reduced. Four test positioning strategies are described below.

The ‘test all’ test positioning strategy

The ‘test all’ test positioning strategy places a test task after each develop, assembly and copy task. The goal of each test task is to reduce the risk in the system at that moment in the sequence as much as possible, if not completely.

The ‘minimal’ test positioning strategy

The ‘minimal’ test positioning strategy only places test tasks if time is available in the integration plan. Time can be available if for instance two components need to be integrated and component 1 is ready before component 2. Component 1 is tested in

[2] An integration and test strategy enforced by the department of defense framework results in an integration and test plan with a standardized form.

this ‘minimal’ strategy, while component 2 is not tested. The duration of the test task (start and stop moment) is derived from the integration plan.

The ‘FS once’ test positioning strategy

The ‘FS once’ test positioning strategy plans a test as soon as a fault state can no more be introduced in the remainder of the integration plan, i.e., each fault state is tested once. Knowledge about the introduction of fault states is required for this test positioning strategy. This strategy only reduces the risk introduced by these fault states when the fault state is not introduced anymore in the sequence. The disadvantage of this method is that the overall risk in the system can increase rapidly when many re-occurring fault states exist, because risk reduction is done late in the process.

The ‘risk-profile’ test positioning strategy

The ‘risk-profile’ test positioning strategy plans a test task if the risk in the system has reached a certain risk upper limit. Risk is reduced, by testing, until a certain lower risk limit is reached. A more advanced strategy uses a ‘risk-profile’ where the upper and lower risk limits are a function of time. The upper risk limit in the beginning of the integration plan is set to another (higher) level than the risk limit at the end of the plan. The same approach is followed for the lower limits. An integration strategy and test positioning strategy are used to create an integration and test sequence in steps 2.1 and 2.2. A case where test positioning is applied at an ASML software release is described in Chapter 6

4.3.1 Integration sequencing

An integration sequence is created in this step, using an integration strategy (2.A) and a *system architecture model* (1.A). Integration sequences can be created by hand or using integration sequencing algorithms like the algorithm proposed in [24]. Chapter 6 describes the results of a case using these algorithms. Other cases have shown that integration sequences that are created using a mathematical algorithm perform better then manually created integration sequences [20]. The main reason for this is that the number of *possible* integration sequences is so large for reasonably sized systems that it is highly unlikely that the optimal integration sequence is chosen by hand. Next to that, changes are bound to happen and result in small changes to the chosen sequence. This causes that the integration sequence gradually becomes less optimal. An integration sequence is described by a graph $G^{IT} = (V, E)$, where the vertices describe the integration tasks and the edges describe the relation between the tasks; see also Chapter 3. The following tasks are used to describe an integration sequence: develop (dev), assembly, (asm) disassembly (das) and copy (cpy). The dev-task is used to describe the development of a single component. The development of a component is described by its development duration. The asm-task combines two components and their interface into a combined component. The das-task removes a component from a module.

With this task a replacement of a component by another component is modeled (disassembly followed by assembly). The cpy-task copies a component such that a second component is created with the same fault states, test cases and properties.

4.3.2 Integration and test sequencing

The next task adds test tasks (tst) to the integration sequence, such that an integration and test sequence is formed. A test positioning strategy, as discussed earlier, is used to position the test tasks in the integration sequence. An example integration and test sequence for the telephone example, introduced in Figure 4.3, can be found in Figures 3.2, 3.3, and 3.4 in Chapter 3. The resulting integration and test sequence is used for integration and test planning.

4.4 Test planning

The previous sequencing step resulted in an integration and test sequence without timing information. The timing and duration of the development and assembly tasks is known from the model or can be derived from the integration sequence. This way, the start and stop moments of each test task can be derived from the integration and test sequence.

A detailed test plan is created in this test planning step for each test task using a *test strategy*. The duration, cost and remaining risk of the resulting test plan are analyzed. Another test strategy is selected if the analysis results do not satisfy the constraints. An overview of the planning step is given in Figure 4.6. A test plan for an individual test task is created using a test strategy. A test strategy consists of a *test sequence*, a *test stop criterion* and a *test process configuration* [70].

4.4.1 Test planning and analysis

A test plan is developed that should meet the objectives, in terms of start and stop moment, maximal cost and target remaining risk. A test strategy is selected for this purpose. This means that a test sequencing technique is chosen, as well as a test process configuration and the test stop criteria follow from the objectives.

Two types of *test sequencing techniques* are currently defined: *off-line* test sequencing techniques and *on-line*, or adaptive, test sequencing techniques. The off-line test sequencing techniques determine a test sequence before test execution is started, while on-line test sequencing techniques define the next test case to be executed during test execution and based on the results of the previous test cases.

Both the on-line and off-line method select the next test case in the sequence using an objective function. The objective functions that are currently defined are: risk based, risk/cost based, inverse risk based, information gain based, random, ordered and probabilistic. A detailed definition of these objective functions using the *system test model* can be found in [70], as well as a comparison of test strategies for three cases.

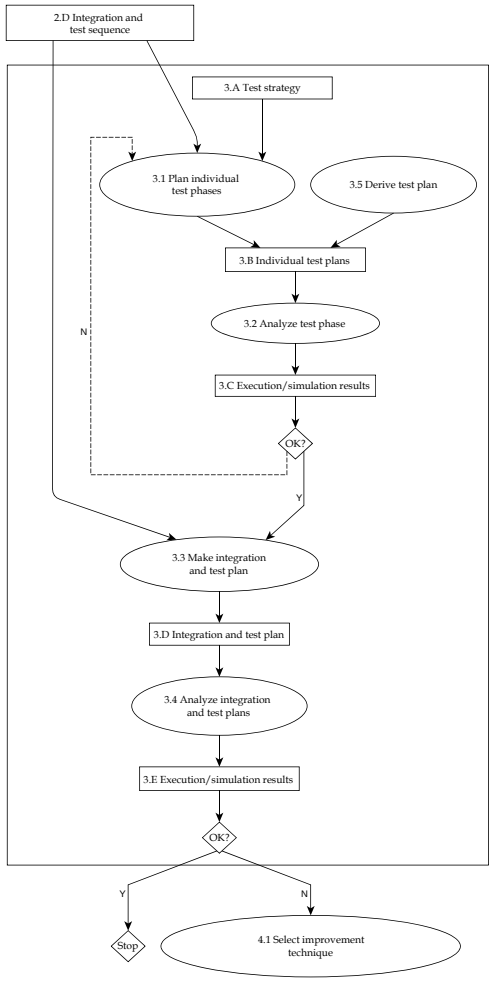


Figure 4.6: Overview of the planning step.

A *test process configuration* determines how test execution, diagnosis and fixing is performed. The number of test, diagnosis and fix resources is taken into account as well as when diagnosis and fix tasks are performed after a failing test case. Diagnosis and fixing can be started immediately after the test case has failed or after all test cases have been performed. When diagnosis and fixing start immediately, test execution can be postponed until the fixes become available, or testing can continue in parallel. These three test process configurations in combination with the available resources can influence the duration, cost and remaining risk of a test task and therefore should be taken into account in the planning process [70].

The *test stop criteria* determine when testing needs to be stopped. Test stop criteria can be defined in terms of a fixed duration, cost limit or remaining risk limit. The trade-off between stopping early with much remaining risk in the system or stopping when the remaining risk is low (or zero) and a very high test duration is reflected in the *test stop criterion*. The effect of this trade-off and the results of test planning after analysis need to be analyzed on the integration and test plan level in step (3.4), because it depends on the remainder integration and test plan if spending more test time or leaving more remaining risk then planned influences the duration, cost and remaining risk of the overall integration and test plan.

A *performance analysis* of each test plan is performed in (3.2). It is checked if the duration, cost and remaining risk of the test plan are according to the objective of the test task. A process simulation based analysis technique [70] has been developed for this purpose. The simulated execution of testing many faulty systems results in test duration, cost and remaining risk distributions and their expected value. Based on these simulation results, it can be decided to change the test strategy or to continue with the next step in the process: integration and test planning.

4.4.2 Integration and test planning and analysis

The integration and test sequence (2.D) and the test planning information (3.C) are combined into an *integration and test plan*. The planning information (start, stop and duration) is available for the dev, asm, das, and cpy tasks in the system architecture model and the system test model. The estimated test duration, cost and remaining risk of the individual tst tasks is added to this plan, resulting in an integration and test plan (3.D). Note that it is possible that the timing of an individual test task does not have to correspond with the original available time for this test task. The duration of a test task could be shorter than the planned duration, because a more optimal test strategy is chosen. The test duration of a test task could also be longer than planned, because the test stop criterion could not be reached earlier. The difference in requested test task objectives and the test task objectives after planning are analyzed on integration and test planning level.

The performance of an integration and test plan is analyzed in (3.4) to determine if the plan performs according to the constraints and objectives as modeled in the system architecture model (1.A). A large number of systems needs to be integrated and tested to determine the average duration, cost and remaining risk for such a plan. An integration and test plan is executed many times in a manufacturing environment, while the execution of an integration and test plan for a newly developed system is only done one or a few times. A simulation process has been developed which is able to simulate the development, assembly and testing of large number of simulated systems. Developing a component results in a set of fault states, which is present in this component. Assembling two components is the combination of two sets of fault states. The risk in the system can be determined using the failure probabilities of the fault states in the system and the impact of these fault states, since fault probability $\times$ impact = risk.

The risk as function of time, a *risk-profile*, is used for detailed analysis. High risk peaks and flat risk areas indicate possible areas in an integration and test plan which can be optimized. High risk peaks can be optimized by developing and assembling components incrementally. The system architecture might need to change for this and also the integration sequence. Flat risk areas can be optimized by planning additional test cases in the flat risk areas to reduce additional risk. Newly developed test cases with additional coverage could be needed for this. Executing test cases from later tasks could also reduce risk. Simulated components must be available for the execution of test cases from later tasks. This way, system level test cases can be executed earlier as described in [28, 27] and in Chapters 7 and 8.

4.4.3 Derivation of integration and test plans

Analyzing existing test plans could be beneficial if these plans are executed many times (manufacturing environment) or if it expected that the executed plan differs from the planned plan. Techniques like business process modeling, can be used to derive the integration and test sequence from logging gathered during the execution of a number of integration and test plans. Test start en stop logging is used, together with additional information, to construct the integration and test plan. This constructed integration and test plan is then compared with the expected integration and test plan. Nonconformances are identified and can be resolved. An experiment with business process mining techniques in the ASML manufacturing department is described in [109].

4.5 Integration and test plan improvement

An overview of the improvement step of the method is given in Figure 4.7. The improvement step currently consists of four techniques: test partitioning, development of new test cases, updating the constraints and objectives of the integration and test plan and redesigning the system. More techniques could be beneficial. A selection task is performed first in (4.1). Each of the techniques is briefly introduced below.

4.5.1 Partitioning test tasks

The main goal of partitioning a test task (4.2) is the reduction of the test duration by executing the two resulting test tasks in parallel. The partitioned test tasks change the integration sequence (2.D). The integration and test planning process is continued with step (3.1), planning of individual test tasks.

An adapted hyper-graph partitioning algorithm is used to partition a *system test model* into two system test models that can be executed in parallel. This local search algorithm is based on [37, 121] and adapted such that the test sequence, test process configuration and test stop criteria are taken into account. This local search algorithm is able to determine optimal solutions, by repeating the algorithm a number of times. The result of a case performed with this improvement technique is a 30% reduction

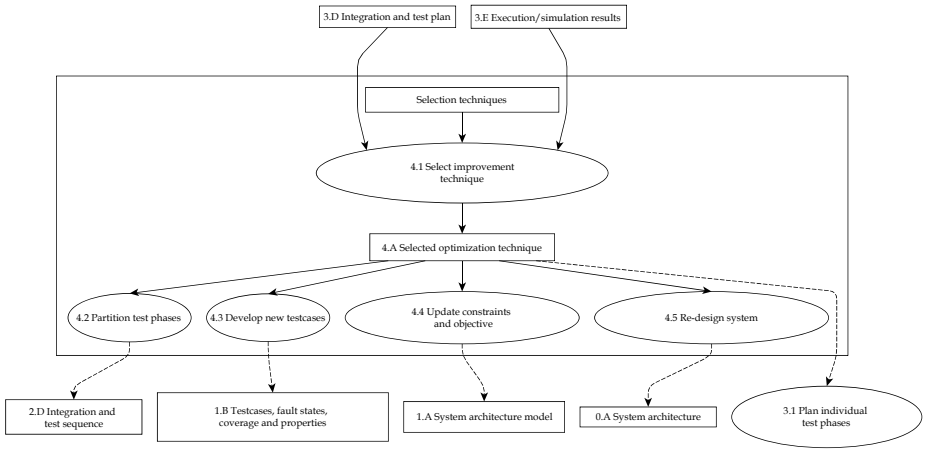


Figure 4.7: Overview of the improvement step.

of the test duration with 30% additional test cost because of parallel execution. More details on this method can be found in [67].

4.5.2 Developing new test cases

The coverage of the individual test cases in a *system test model* determines how many risk *can be* reduced in a test task. The combination of the test cases that are executed and the sequence of execution determines *how* the risk is reduced. In practice, test cases are added because new components were developed or problems were found, not because a new test case that is a combination of test cases with a specific coverage reduces the test duration or cost. The next-best-test-case algorithm has another approach. The *system test model* is analyzed and test cases are proposed with the highest information gain for that *system test model*. This means that a set of covered fault states (a new test case) is proposed, taking into account the coverage of the current test set and the failure probabilities of the fault states.

The next-best-test-case algorithm should determine the information gain for each *combination* of fault states and choose the combination of fault states with the highest information gain. This is computationally intensive for models with $|S| > 17$. Therefore an efficient clustering algorithm has been developed that first clusters fault states until $|S| < maxS \wedge maxS < 17$ and then determines the combination of fault states with the highest information gain. This algorithm and the results of a case at ASML have been described in [69].

4.5.3 Updating objectives or constraints and system redesign

Updating the objectives and constraints (4.4) is not really improving the integration and test plan. Updating the objectives and constraints could be the last resort if no feasible integration and test plan can be determined. Integration and test planning starts all over with updated objectives and constraints, because this update could result in a new integration sequence.

The last improvement technique discussed here is redesigning the system, such that a better integration and test plan could be made. This improvement technique should be selected if the current architecture of the system is not optimal for integration and testing. Examples of an architecture that is not optimal for integration and testing are:

- architectures that contain components of very different granularity (very large components in combination with small components, components with many interfaces in combination with components with little interfaces, et cetera),
- architectures that contain many components that are connected with many other components,
- architectures with a *layering*, groups of components, that have many interfaces between these groups if components.

Redesigning, rearranging the components, or choosing a different *layering* in the system could be beneficial in these cases. The effect of the number of components, number of interfaces and the layering in the architecture is discussed in detail in [68].

4.6 Conclusions

This chapter presents an integration and test planning method, which utilizes three types of strategies: an integration strategy, a test positioning strategy and a test strategy. The combination of strategies and system models results in an integration and test plan.

A system architecture model, consisting of components, interfaces, a layering, objectives and timing, is used to create an integration sequence using an integration strategy. Possible test tasks are positioned in between these integration steps according to a test positioning strategy. The individual test tasks are planned in detail using a test strategy and a system test model that describes the coverage of the available test cases on the possible fault states in the system. Properties for test cases and fault states like durations of test cases, diagnosis and fixing, failure probability and impact is used to determine a test plan for the test task. The performance of the test tasks is analyzed with respect to duration, cost and remaining risk using test process simulation.

The individual test tasks are combined into an integration sequence to form an integration and test plan. The performance of this integration and test plan is analyzed using the same process simulation that is used for the analysis of test phases. An appropriate optimization technique is selected if the performance is not according to

the initial objectives and constraints. Parts of the process are repeated depending on the selected optimization technique.

This general method describes the process which integration and test planners could use to create and improve integration and test plans. This method can be applied in industry by applying best practices for each step in the method. The detailed sequencing and improvement algorithms that have been developed can also be used in combination with best practices. The algorithms use models to determine test and integration plans. The LONETTE tool set has been developed to support the modeling of the system architecture and the test model. The algorithms that have been developed are also available via this tool set. Furthermore, this method can be used to guide research into more formal techniques, tools and methods that can be used within this framework.

Chapter 5

Test time reduction by optimal test sequencing

Authors: R. Boumen, I.S.M. de Jong, J.M. van de Mortel-Fronczak, J.E. Rooda

5.1 Introduction

Testing complex manufacturing systems is expensive both in terms of time and money, as shown by [39] and [41]. To reduce time-to-market of a new system or to reduce lead time during the manufacturing of these systems, it is crucial to reduce the test time. Reducing test time can be done by: 1) making testing faster, for example by automation of test cases, 2) making testing easier, for example by changing the system and 3) doing testing smarter, for example by choosing wisely which test cases to perform and in what sequence. Much research has been done on the first two aspects, less research is being done on the third aspect. In this chapter we show that test time can be reduced by optimizing test sequences which allows testing to be performed more efficiently. Deciding which tests are performed and which are not is important. Not performing a test case may leave crucial faults in the system, while performing a test case can lead to an increase of lead time or time-to-market. Deciding what to test is especially important in the time-to-market driven semiconductor industry and for companies providing manufacturing systems to this industry such as ASML. This is caused by the time-to-market pressure of delivering such machines to customers and the high costs associated with solving defects during system operation. For the optimization of test sequences three parameters are of interest: the test time, the test cost and the quality of the system after testing. In papers [23, 22], a test sequencing method has been developed that optimizes the selection and sequencing of tests. The main focus of these papers is on the mathematical models and algorithms. In this

chapter, we explain how the test sequencing method can be used to reduce test time. Additionally, we describe two case studies in which test time has been reduced. The first case study is related to the manufacturing test phase of a lithographic machine. The second case study is related to the reliability test phase of a new lithographic system.

The structure of the chapter is as follows. Section 5.2 briefly explains the test sequencing method. Sections 5.3 and 5.4 each describe a case study. The last section gives the conclusions of our work. This chapter is based on the paper titled 'Test time reduction by optimal test sequencing' [19] presented at the International Council of Systems Engineering (INCOSE) 2006 Symposium.

5.2 Test sequencing method

The test sequencing method has originally been used to solve diagnosis problems as described in [96] and [110]. In [23] the method has been adapted to solve system test sequencing problems. The method consists of four steps: definition of the test model, definition of the optimization objective and test stop criterion, calculation of the test sequence and, finally, testing the system. In Figure 5.1, an overview of the steps that comprise the test sequencing method is shown. In the sequel, we describe each step in more detail.

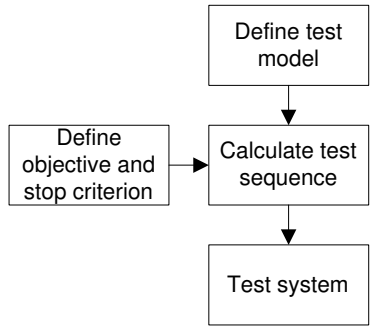


Figure 5.1: Test sequencing method.

Define the test model

To calculate an optimal test sequence for a certain test sequencing problem, the problem needs to be defined in a mathematical way as a, so-called, test model. This test model consists of:

- a set T of available tests,
- for each test, the test cost C in cost units and in time units to perform that test,

- a set S of possible fault states which are independent of each other,
- for each fault state, the probability P that a fault state may be present,
- for each fault state, the impact I in cost units or in time units if that fault remains and causes a system failure during operation,
- for each test, the fault states that are covered by that test, which is represented by a matrix A. A test covers a fault state completely if the test always fails when the fault state is present; then A_{ij} is 1. If A_{ij} is between 0 and 1, then test i covers fault state j with the probability A_{ij}, meaning that a test fails with probability A_{ij} if the fault state is present. If $A_{ij} = 0$, then test i does not cover fault state j and will pass when the fault state is present.

The set of available tests is often known for a certain problem. The cost of performing a test does not include the development cost of a test but only the execution cost. A fault state can be of many failure types. During manufacturing a fault state is often a failing component or an integration fault (e.g., loose cable). During development, a fault state can be a performance criterion that is not reached or a specification that is not met. The initial fault state set during manufacturing can be a list of components and interfaces, while during development the initial list can be the result of an Failure Mode and Effect Analysis (FMEA) or the list of system requirements. The probability that a fault state is present can be derived from historical data for manufacturing problems. For development problems, this probability can be derived from the risk analysis prediction within an FMEA analysis. If this data is not present even tentative assumptions (0%, 20%, 50%, 80%) on fault probabilities may lead to good test sequences, as we have seen in practise. The impact of a fault state is the cost or time related to the repair of the system and the cost of the caused damage if the system fails during operation because that fault state is present. The impact can be different for each industry and system. In the Department of Defence (D.o.D), space and aviation industries, the impact of a failing system can be extremely high since the cost of an airplane or shuttle crash is huge. In the manufacturing machine and semiconductor industry, the impact of a fault is much lower. A fault may result in a few hours machine downtime, but does not destroy anything. Which tests cover certain fault states is often known by the developers of the test cases or by test experts. The test coverage is more difficult to identify. Normally, this test coverage is 1, but in some cases this coverage is lower than 1. For reliability testing the coverage of 1 hour testing to show for example 100 hours of Mean-Time-Between-Failure (MTBF) can be approximated by 1hour/100hours= 0.01. For some tests, historical data may indicate how often a certain fault state is identified by a test.

We illustrate the modeling of a test problem with two small examples. The first example is shown in Table 5.1. The model represents a manufacturing test phase of a simple telephone, which consists of 3 modules: the receiver, the cable and the device. Additionally, two interfaces are distinguished: between the device and the cable (D2C) and between the receiver and the cable (R2C). We model this system with 5 fault states:

$S \setminus Test$	1	2	3	4	5	6	P	I(hour)	I(€)
Device	1	1	0	0	1	0	10 %	20	200
Cable	1	0	1	0	1	1	10 %	20	200
Receiver	1	0	0	1	0	1	10 %	20	200
D2C	1	0	0	0	1	0	10 %	20	200
R2C	1	0	0	0	0	1	10 %	20	200
C(hour)	3	1	1	1	2	1			
C(€)	50	20	20	20	30	30			

Table 5.1: Example manufacturing test model of a telephone.

S / T	Test 1	P	I(hour)	I(€)
MTBF	1/10	100 %	50	10000
C(hour)	1			
C(€)	500			

Table 5.2: Example reliability test model of a system.

one for each module and one for each interface. Six tests are available to test this system. A duration in hours and a cost in € is associated with each test. The impact of each fault state is 20 hours and €200, and the probability of each fault state is 10%. The second example shown in Table 5.2 is a model of a reliability test phase. The system is required to run, on average, at least 10 hours before a failure occurs, which means a MTBF of at least 10 hours. This requirement is modeled with one fault state called MTBF. The impact of this fault state is 50 hours and €10000. We have one test to show that this requirement is reached. Executing this test costs 1 hour and €500. The coverage of this test is 1hour/10hours = 0.1.

Define the objective and stop criterion

To decide which solution is optimal, the optimization objective and the test stop criterion need to be defined. What is considered optimal depends on what is important for the industry. We introduce three parameters that can be used to define what is important: the test time, the test cost and the quality of the system after testing, cf. the business drivers Time–Quality–Cost in Chapter 3. The time and cost of testing are defined in the model. The quality of the system is represented by the risk, defined in time and cost, present in the system. A high risk in the system means a low system quality. The system risk is the sum of the risk per fault state, which is defined as the impact of a fault state times the probability that the fault state is present, or:

$$R_{system} = \sum_{s \in S} I(s)P(s) \tag{5.1}$$

Applying a test to a system has two effects: the total test cost and test time increase and the risk in the system decreases either by tests that pass, or by repairing identified faults. We now abstract from the repair cost and time of faults during the test phase. These effects are also shown in Figure 5.2, where the increase of test cost or time and the decrease of risk is shown for a fictitious illustration. In practice, the increase of test cost may be nonlinear because of a difference in test costs. The third line in this figure shows the total cost or time, which is the sum of the test and risk time or cost. A test sequence can be optimized towards: A) minimal test time or B) minimal test cost. The test stop criterion decides when to stop testing. There are at least three test stop criteria: 1) stop when a certain system quality defined in risk is reached, 2) stop when the maximal test time or cost is reached, or 3) stop when the total cost is the lowest. Each test stop criterion is shown in Figure 5.2.

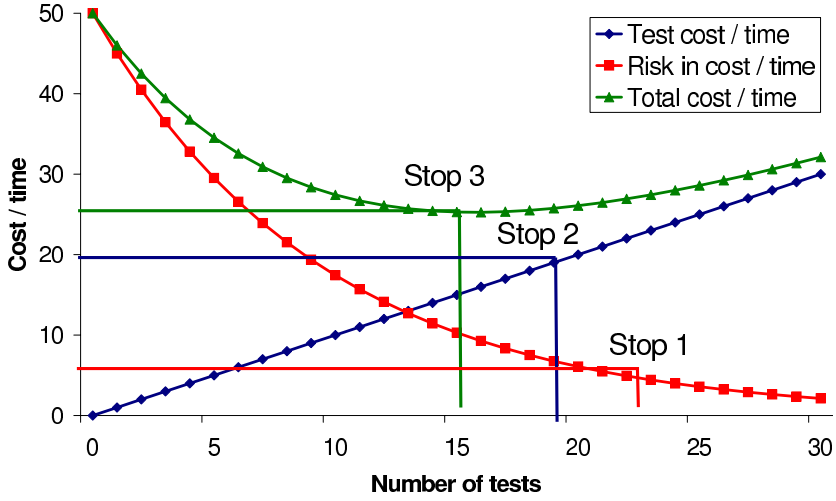


Figure 5.2: The effect of testing.

Calculate a test sequence

Now that we defined a model, an objective and a test stop criterion, the optimal solution can be calculated with the algorithm described in [22]. The solution is represented in a graph where the nodes are either test (circle), fix (square), diagnose (square) actions, or stop (triangle) actions, as shown in Figure 5.3(a) and 5.3(b). The outgoing arrows of a test denote the possible outcomes of a test: either pass or fail. Depending on the outcomes of individual test cases, a path in the graph is taken, which results in

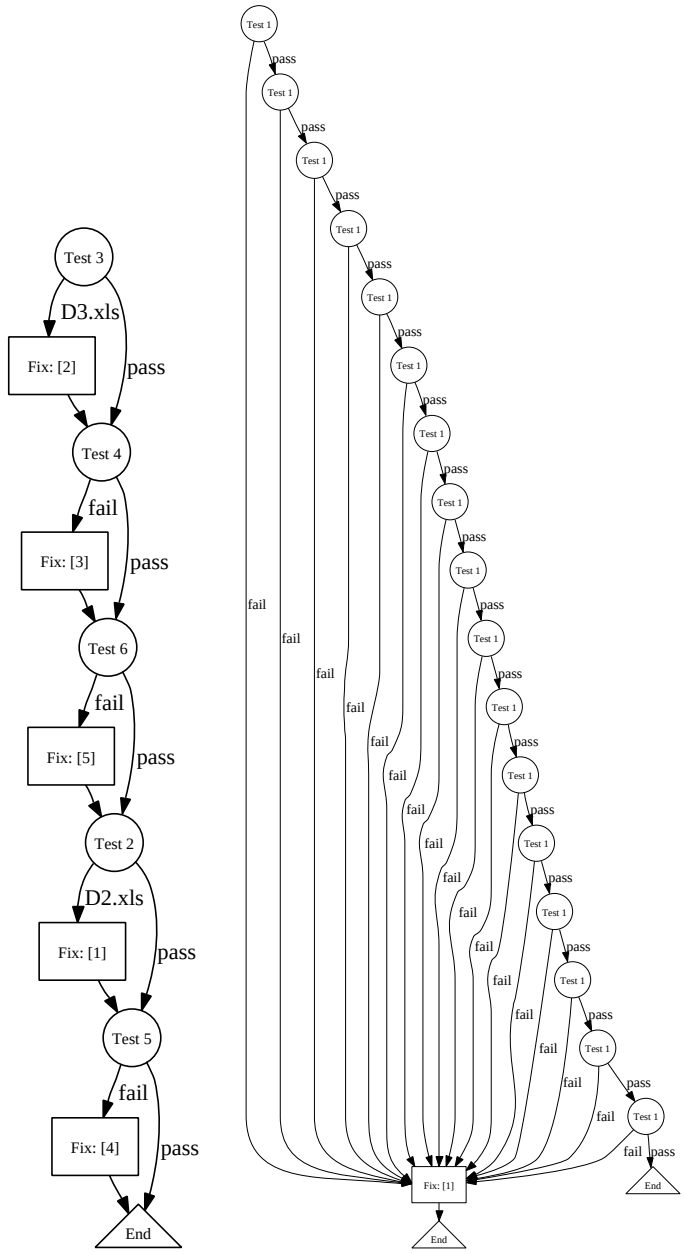
a test sequence. A fix action of a fault state denotes that the fault state is present and is fixed. A diagnosis action of a set of fault states denotes that at least one of these fault states is present, but no tests are available to distinguish the present fault state. Therefore, these fault states are diagnosed and the present fault states are fixed. The solution test graph of example 1 is shown in Figure 5.3(a). This test sequence is optimized towards minimal test time, while testing stops when the remaining risk is 0 (the highest quality). The average test time and test cost are 5,24 hours and €82.40 while the maximal (=longest path in graph) test time and cost are 16 hours and €260. The solution graph of example 2 is shown in Figure 5.3(b). This test sequence is also optimized towards minimal test time, while testing stops when the total cost is the lowest (stop criterion 3). The average test time and cost are 10,36 hours and €5181. The maximal test time and cost are 17 hours and €8500. If all tests are executed successfully, the remaining risk in time and cost is 8,33 hours and €1666, which is a risk reduction of 83%.

Testing the system

Now that a solution is obtained for a test sequencing problem, the system can be tested. The solution graph of example 1 can be automated, such that a next test is chosen based on the test graph and the outcome of previous tests. The dynamic behavior of the test graph (one does not know upfront which tests will be performed) results in an unknown end time of the test process. The solution graph of example 2 states how many hours of reliability testing are needed to satisfy the reliability requirement. This can also be explained by analyzing Figure 5.4, showing how the time of testing, the risk in [hours], and the total cost (sum of the testing time and risk) are changing with the number of tests applied. Here it is shown that when performing 17 tests, the total cost is the least. This corresponds to the optimal test sequence shown in Figure 5.3(b) which also consists of 17 tests.

5.3 Manufacturing case study

In this section, we describe a case study that is related to a test phase during the manufacturing of a lithographic machine. During the manufacturing of a lithographic machine, two large test phases are performed. Test phase 1 is performed at ASML. After successful accomplishment of this phase, the system is disassembled, transported to the customer, and reassembled again. After the system has been installed at the customer's site, test phase 2 is performed. The two test phases are almost identical, and both include multiple job-steps. Each job-step consists of multiple tests executed in a sequence. Currently, the sequence does not depend on the outcome of a test: if a test fails, the problem is diagnosed until the cause of the failure has been found and repaired. In these job-steps, faults introduced during manufacturing, assembly and transport must be discovered and system parameters need to be calibrated. The case study is performed for three job-steps in both test phases. Test models have been made by



(a) Solution of example 1

(b) Solution of example 2

Figure 5.3: Solutions for examples.

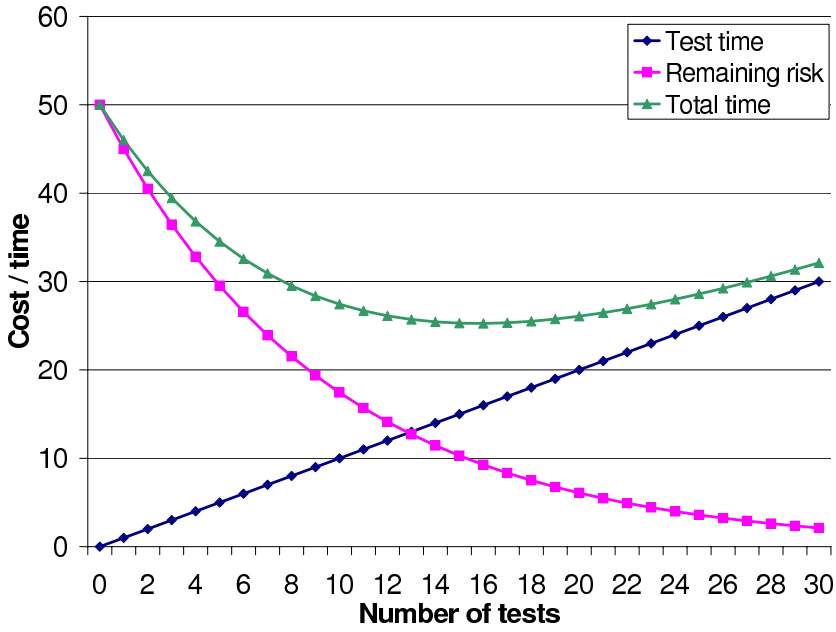


Figure 5.4: Time of testing for example 2.

ASML engineers and the test sequences have been optimized towards test time, while testing stops when the risk is 0 hours (stop criterion 1). The parameters of the three models related to three job-steps are shown in Table 5.3. Since the test problems in test phase 1 and 2 are identical almost the same models can be used: only the probabilities differ between the two test phases, as also shown in Table 5.3.

In general, the fault probabilities are lower in the second test phase, because the system was already performing according to its specifications after the first test phase. The faults found during the second test phase are merely transportation and assembly faults. No data is available for job-step 3 in test phase 2. The results of this case study are shown in Table 5.4, presenting the currently average test time (determined from

Job-step	Number of tests	Number of fault states	Average fault probability test phase 1	Average fault probability test phase 2
1	39	73	30 %	16%
2	15	15	86 %	71%
3	33	60	46 %	No data

Table 5.3: Model properties of the manufacturing case study.

	Test phase 1			Test phase 2		
Job-step	Current time [hour]	New time [hour]	Reduction	Current time [hour]	New time [hour]	Reduction
1	12.2	10.2	17%	12.2	8.3	32%
2	13.6	13.1	4%	13.6	11.5	16%
3	33.0	27.0	18%	No data	No data	No data

Table 5.4: Results of the manufacturing case study.

$S \setminus Test$	1	2	3	4	5	6	P	I(hour)	I(€)
Module 1	1/50	6/50	3/50	2/50	2/50	0	90 %	100	10000
Module 2	1/50	0	10/50	0	4/50	0	80 %	100	10000
Module 3	1/50	0	3/50	0	2/50	0	80 %	100	10000
Module 4	1/50	0	0	2/50	4/50	4/50	80 %	100	10000
Module 5	1/50	0	0	1/50	4/50	10/50	90 %	100	10000
System	1/50	0	0	0	0	0	50 %	100	10000
C(hour)	1	1	1	1	1	1			
C(€)	500	500	500	500	500	500			

Table 5.5: Model of reliability case study.

historical data), the new average test time and the reduction in test time. Due to the currently used fixed test sequence, the test times in both test phases are equal. However, when using the presented method we notice that more test time can be reduced during the second test phase than during the first test phase because fault probabilities are in general lower. Due to these lower fault probabilities, we can start with test cases that cover more fault states which, in most cases, reduces test time.

5.4 Reliability case study

In this section, we describe a case study that is related to a reliability test phase during the development of a new type of lithographic machine. For a new type of a lithographic machine, a test phase needs to be performed to show that the first of a kind machine has a reliability of at least 50 hours MTBF. For the most critical modules (the modules that have been changed) in the system, separate tests have been developed that test a particular module faster than a total system test (normal production) would do. For example, during normal production a robot accomplishes 1 cycle per minute; in a specific test, the robot is tested such that it accomplishes 10 cycles per minute, without testing the rest of the system. In the current situation, the specific module tests are not used to show the total system reliability. Only the system test is used, because current methods, for example SEMI standards, only consider this type of test when testing the

Test	Hours
Test 1	26
Test 2	0
Test 3	7
Test 4	0
Test 5	37
Test 7	0
Total	70

Table 5.6: Results of the reliability case study.

system reliability. The model of this problem is shown in Table 5.5. There are five critical modules, and a fault state (s1-s5) is associated with each module. Additional fault state (s6) is associated with the rest of the system. The normal production is the system test (t1) that covers each fault state with a probability of 1/50. The other five tests (t2-t6) provide higher coverage for certain modules, because these tests make more cycles per hour for these modules than normal production would make.

The solution is optimized towards test time, while testing stops when the total cost is minimal (stop criterion 3). The optimal solution has a maximal test time of 70 hours, which is shown in Table 5.6. This means that 70 hours of testing is needed to show the 50 hours MTBF. During this test phase, none of these tests may fail. Otherwise, the MTBF requirement is not fulfilled.

In Figure 5.5, the risk in the system denoted in time units is shown for two situations. The first situation (Test 1 only) shows the risk decrease when only Test 1 is used. This situation is the current situation at ASML. Situation 2 (optimal sequence) shows the risk decrease of the optimal test phase, calculated with the test selection method. The optimal test stop moment for situation 1 is 111 hours of testing, and for situation 2 it is 70 hours of testing. From this figure we can conclude that it is profitable to use the critical module tests to test the system reliability instead of using only Test 1. Also, the remaining risk in the system is only 119 hours when using the critical module tests, while the risk is 160 hours when using only Test 1.

5.5 Conclusions

Reducing time-to-market or lead time for complex systems is extremely important. Test and integration time is almost always on the critical development path. Therefore, reducing this time will definitely reduce time-to-market or lead time. There are several approaches to reduce test time: testing can be made faster by automating test cases; testing can be made easier, for example, by changing the system; or testing can be done smarter, for example, by applying the method described in this chapter and in [23, 22]. The benefit of our method is avoidance of additional investments: we still use

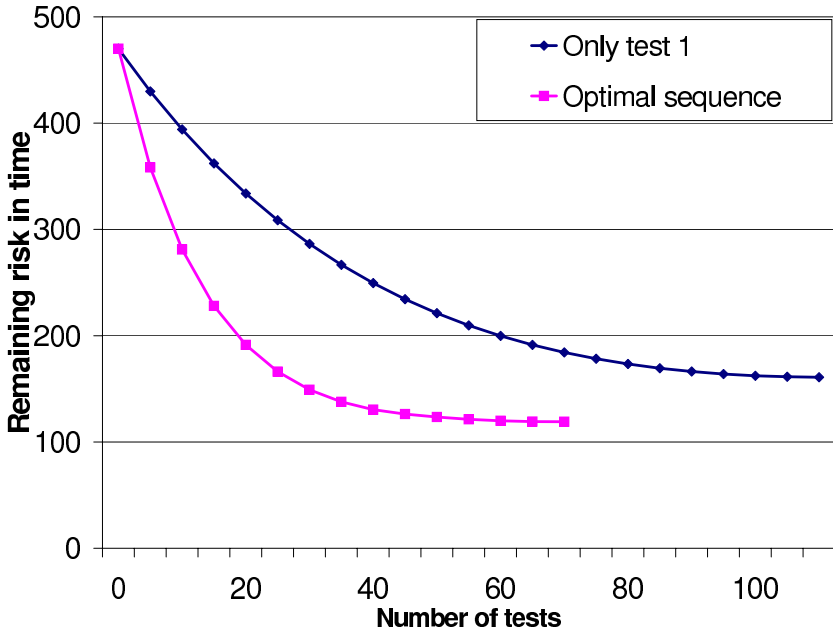


Figure 5.5: Risk decrease of the reliability case study.

the same tests on the same system. This method incorporates the possibility to define different optimization criteria for different industries and systems. A test sequence can be optimized towards time or costs. Testing stops when the maximal time or cost is reached, or the system has reached a certain quality, denoted with risk, or when the total cost is the lowest. Because the impact of a fault is different for each system, a different solution will be optimal. For example, the impact of a fault in an airplane is much higher and will therefore result in more testing than for a manufacturing machine. Test time within the testing of complex manufacturing systems can be reduced by using the presented methods. This is shown with two case studies in this chapter that apply to the development and manufacturing phases of a lithographic machine. From these case studies we learn that this method can reduce up to 20% in test time, compared with test sequences made by hand by ASML engineers. The test model as presented here is a very intuitive and easy way to write down a test sequencing problem. Furthermore, this model can be used for many other analyses, for instance, for test strategy simulation as described in [70], to decide which test needs to be developed next, or for a static analysis of the test problem.

Chapter 6

Optimal integration and test planning for lithographic systems

Authors: R. Boumen, I.S.M. de Jong, J.M. van de Mortel-Fronczak, J.E. Rooda

6.1 Introduction

In today's industry, time-to-market of systems is becoming increasingly important. The integration and test phase of a complex system typically takes more than 45% of the total development time [41]. Reducing this time shortens the time-to-market of a new system.

In the integration and test phase of system development, components which were concurrently developed are integrated into a subsystem. Subsequently, the subsystems are integrated into a system. In between these integration actions, tests are applied to check the system requirements. An integration plan describes the sequence of integration actions and tests that need to be performed. For new ASML lithographic systems, integration and test plans are currently created manually.

An inefficient integration and test plan may result in finding faults late in the integration and test process, because tests are performed late in the process. The rework caused by these faults increases the integration and test phase duration. Furthermore, not keeping a plan up to date causes an inefficient way of working that increases the duration of the complete phase. A good integration and test plan performs tests early and in parallel, as much as possible, such that faults are found early in the process. Furthermore, when a plan is kept up to date, it is easier to make the correct decisions during the often chaotic integration and test phase. An optimal integration and test plan generally does not increase or decrease the system quality but increases the efficiency

of working such that cost and/or time are minimized. Creating good or even optimal integration and test plans is getting more and more difficult because of:

- The growing number of components in today's systems. This results in numerous possible integration and test plans.
- The parallelism in the plan. Subsystems or modules should be tested in parallel as much as possible. Also, component models can be used to perform certain tests before actual components are delivered (see model-based integration in Chapter 7).

Maintaining an integration and test plan is also getting more and more difficult because of:

- The variability in delivery times of components. If a component arrives later than planned, the plan should be updated.
- The variability in test phase duration. Failing tests initiate a diagnosis and repair action and may increase the test phase duration.
- Varying number of components. During integration, it is possible that more components are needed than originally planned, such as software patches that were not included in the original system design.

Due to these difficulties, a method is needed that allows for automatic creation of integration and test plans that are optimal for the time-to-market of a system. This method should also minimize the effort needed to keep a plan up to date. In this chapter, we introduce such a method. This method is called the integration and test planning method and consists of the following steps. First, a model of the integration and test problem is created that describes the problem mathematically. Second, an algorithm is used to automatically calculate the optimal integration and test plan. Finally, the plan is executed. A new plan can be calculated automatically after updating the model if a plan update is needed during the execution of the original plan.

The chapter is structured as follows. Section 6.2 describes the integration and test phases of lithographic machines. Section 6.3 describes the proposed integration and test planning method. Section 6.4 describes two case studies where this method has been applied to the integration and test phases of lithographic machines. The last section gives conclusions. This chapter is based on the paper titled 'Optimal integration and test planning applied to lithographic systems' [20] presented at the International Council of Systems Engineering (INCOSE) 2007 Symposium.

6.2 Integration and testing of lithographic systems

To reduce time-to-market of a new type of lithographic system, often multiple prototypes are created to perform tests in parallel. Normally, each of these prototypes is

used for a specific goal, for example, the first prototype is used to test all functional requirements, whereas the second prototype is used to test all performance requirements. Normally, for each of these prototypes an integration and test plan is manually created by an integration engineer using Microsoft Project. The integration and test phase of these systems is characterized by a large time-to-market pressure and a huge number of multidisciplinary components (mechanical, electrical, optical, software) that are developed in parallel and should be integrated. During such an integration phase, first an old type lithographic system is manufactured and qualified. This system is then upgraded to the new type system by replacing specific modules with the new modules, upgrading the software and performing tests to check the system requirements. This approach reduces the risk of possible faults because a complete working machine is taken as starting point. Often, models or 'dummy' components are used during the integration phases to perform tests earlier in the integration phases, before the actual modules are delivered. During the execution of an integration and test plan, the plan often needs to be updated. If a module arrives later than planned, the duration of the module development is updated in the plan. Microsoft Project then automatically delays all tasks that depend on this development task. However, the sequence of tasks is not changed by Microsoft Project, which may result in suboptimal plans. Therefore, the sequence of tasks needs to be updated manually which increases the effort to update a plan.

6.3 Integration and test planning method

In this section, we introduce the integration and test planning method that allows to automatically create an optimal integration and test plan. The method originates from assembly sequencing methods as described by [83, 82] and object-oriented integration strategies as described by [55]. In [24] the assembly sequencing method and the object-oriented integration strategy method are combined into a method to solve integration and test planning problems. The method consists of three steps as shown in Figure 6.1: define the integration and test model, calculate the integration plan, and execute the plan. During execution it is possible that the model needs to be adjusted (for example because of delays in delivery times) and the plan needs to be updated. In the remainder of this section, we describe each step in more detail.

To calculate an optimal plan for a certain problem, the problem is defined in a mathematical way as an integration and test model. This model consists of:

- a set M of modules,
- for each module in M, the associated implementation duration of that module,
- a set I of interfaces that each connect exactly two modules with each other,
- for each interface in I, the associated construction duration of that interface,
- for each interface in I, the two modules of M associated with it,

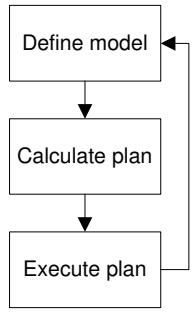


Figure 6.1: Integration and test planning method.

- a set T of tests,
- for each test in T, the associated duration of performing that test,
- for each test in T, its essential sets of modules; that is the sets of modules from M that must be integrated before the test can be performed.

This model needs to be defined by an engineer and contains all information needed to create an integration and test plan. The set M of modules can be obtained by decomposing the system into subsystems or components that are implemented or developed in parallel. Normally, this has already been done during the design phase. Furthermore, the set of modules may consist of the component models that can be used as replacements for other modules, see for more information Chapter 7. The implementation duration of a module denotes the time between the start of the implementation of the module and the end of the implementation of the module. The set I of interfaces between modules denotes how the modules can be integrated with each other. Every interface is created between exactly two modules. If two modules have an interface, they can be integrated with each other. Examples of interfaces are mechanical interfaces such as bolts and screws, but also software interfaces. The construction of an interface may take some time, for example a mechanical interface may take a few hours to construct.

The set T consists of the tests that need to be performed to check system requirements. We assume that each test needs to be performed exactly once. The duration of each test must be known in advance. The selection of tests that need to be performed is not considered part of this method. In Chapter 5 a test selection and sequencing method has been developed that can be used to determine this sequence of tests. For each test, the essential sets of modules must be defined. An essential module set denotes the minimal set of modules that need to be integrated before that test can be performed. If component models are used to replace certain modules, two essential sets of modules can be created to denote that either the real component or the component model should be integrated before the test may be performed.

T	Essential sets of modules	Duration
T1-T6	Reticle handler	1
T7, T8	Reticle handler and stage	2
T9-T11	Wafer stage	1
T12, T13	Wafer handler and stage	2
T14-T17	Lens, laser, illuminator	3
T18-T20	Wafer and reticle stage, lens, laser, illuminator	3
T21-T25	-all modules-	5

Table 6.1: Illustration model.

The assumptions on the integration and test model are:

- All modules in M must be connected with each other, so there exists a path of interfaces that connects every module in M with every other module in M. This also holds for an assembly.
- For every test in T, there exists at least one module that is present in all essential sets of modules belonging to this test, to make sure that each test is performed exactly once.
- Each test is performed exactly once when one of the essential sets of modules of this test has been integrated.
- The durations of the tests and the durations of constructing the interfaces are independent of the current assembly of modules.

We define that an assembly consists of one or more modules that have been integrated. An integration action is defined as creating all interfaces between exactly two assemblies sequentially. A test phase consists of the set of tests that are performed on an assembly.

We illustrate the integration and test model with a small example. In this example, all subsystems of a simple lithographic machine, see Chapter 2, must be integrated and tested. In Figure 6.2, the different modules, interfaces and their development and creation durations (denoted as t) are shown. In Table 6.1, the essential sets of modules per test and the test durations per test are shown.

After the model has been defined, the optimal plan can be calculated. The optimal plan is the plan that integrates all modules into one system and performs each test exactly once in the shortest possible integration and test time. Note that no tests are removed or skipped and that the total test duration is therefore always the same. The optimal plan is the most efficient plan because the tests and integration actions are performed in parallel as much as possible.

The optimal plan can be calculated using the algorithm described in [24]. The basic idea behind this algorithm is that the plan is constructed according to the 'assembly by

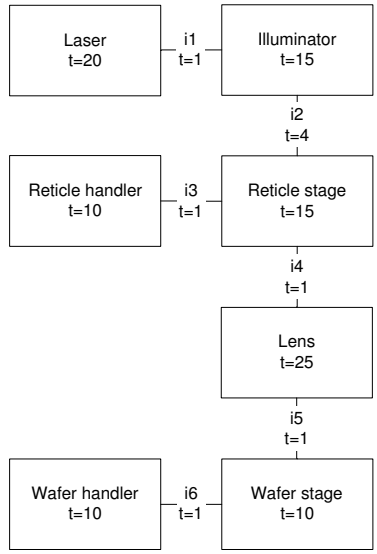


Figure 6.2: Illustration model.

disassembly' approach using an AND/OR graph search, as was suggested by [83, 82] to create assembly plans. This approach starts with the complete system and investigates all possible ways in which the system can be disassembled into two separate assemblies, which can again be disassembled into two separate assemblies, and so on until the single modules remain.

For the example model, the optimal solution is shown in Figure 6.3 as a tree. In this tree, the development of a module is shown as a square node, the construction of a set of interfaces is shown as a hexagonal node, the execution of a test phase is shown as an oval node and the sequence of actions is denoted by the edges between the nodes. The critical paths of this plan are the path of the lens and the path of the illuminator that both take 73 time units. The cost of the total integration and test plan is therefore also 73 time units. Another representation of the solution is the Microsoft Project Gantt chart in Figure 6.4. In this chart, the critical paths of the lens and illuminator modules are depicted in light grey.

6.4 Case studies

This section shows the results of two case studies that were performed during the integration and test phase of the development of two new ASML systems. The first case study shows the optimization of the integration and test plan of a new lithographic prototype and shows a plan update that was performed when the deliveries of certain

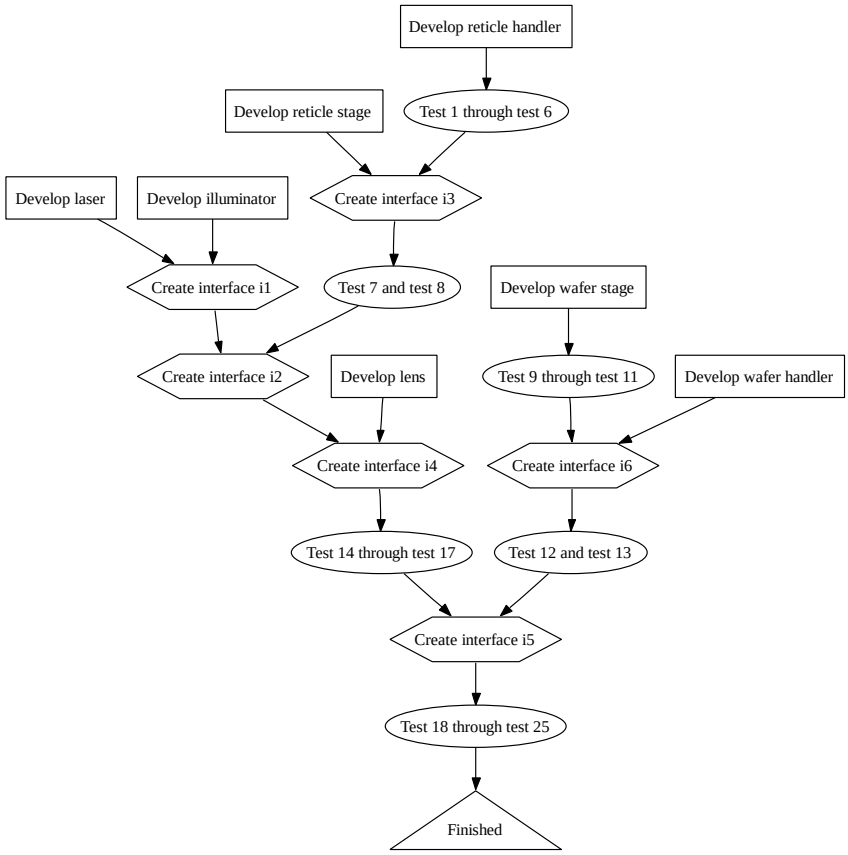


Figure 6.3: Illustration solution represented as a tree.

modules were delayed. The second case study shows the optimization of the integration and test plan of two prototypes of a completely new type of system where some tests must be performed on one specific prototype and other tests may be performed on either the first or the second prototype.

Case study 1

This new lithographic machine is constructed based on an old type system. Only the upgrade of certain modules is considered and not the integration of the old type system. Therefore, the old system is modeled as one assembly (M1) that is completely present at the start of the project. There are 16 other modules (M2 through M17) that are integrated in the old system to upgrade this system to the new system. Modules M10,

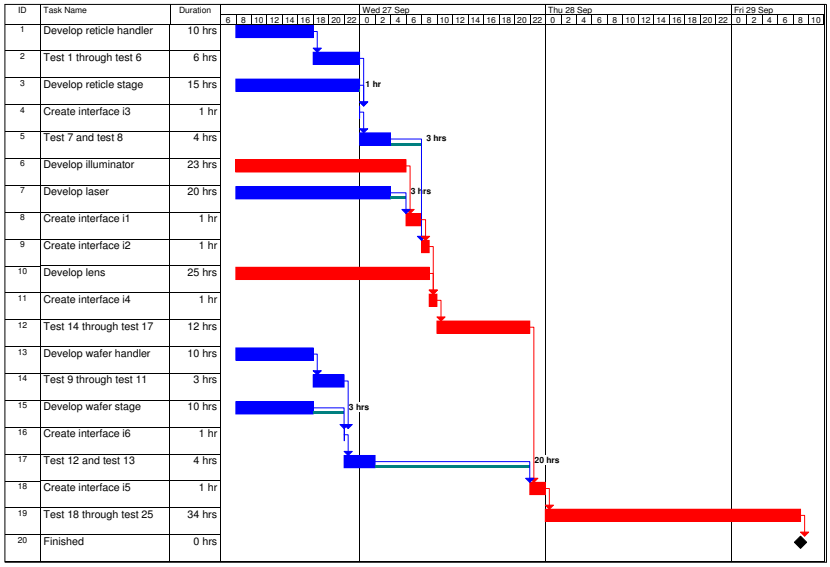


Figure 6.4: Illustration solution represented as an MS project Gantt chart.

M11 and M12 are different laser system types. Some tests require one of these modules to be integrated before they can be performed while other tests require one specific laser to be integrated. The complete model of this system is shown in Figure 6.5 and Table 6.2. All modules are connected to the old system (M1), while the three lasers (M10, M11, M12) are also connected to M9.

The integration plan for this model is shown in Figure 6.6(a). The total duration of the plan is 1469 hours. The critical path is the path that module M2 follows and is shown in light grey.

At a certain point in time during the execution of this plan the delivery times of some modules have been changed. In Table 6.3, the new development durations of these modules are shown. Furthermore, module M15 is removed in the new plan. After a simple update of the model, a new plan has been calculated automatically. This new plan shown in Figure 6.6(b) shows the new critical path of module M14 in light grey. The light grey vertical line in the figure shows the time at which the plan was updated. For this case study we have not made a comparison with a manually created plan.

T	Essential sets of modules	Time [hour]	*T*	Essential sets of modules	Time [hour]
T0	M1	96	T8	M1,M2,M3,M15	10
T1	M1,M2	165	T9	M1,M2,M3,M9	29
T2	M1,M3	68	T10	M1,M2,M3,M17	6.5
T3	M1,M2,M4	5	T11	M1,M2,M3,M16	12
T4	M1,M2,M3	278.5	T12	M1,M2,M3,M6, M9,M11	82
T5	M1,M2,M3,M9, M10	100	T13	M1,M2,M3,M5,M6,M8, M9,(M10 or M11 or M12),M13,M14,M15,M16	212
T6	M1,M2,M3,M13	10	T14	M1,M2,M3,M6,M9,M12	82
T7	M1,M2,M3,M14	10	T15	M1,M2,M3,M9,(M10 or M11 or M12)	10
T8	M1,M2,M3,M15	10	T16	M1,M2,M3,M7	120

Table 6.2: Case study 1 model.

M	Old development duration	New development duration
M7	904	1288
M8	688	1048
M11	688	1216
M12	888	664
M14	536	1416
M15	552	Removed

Table 6.3: Changed development times for case study 1.

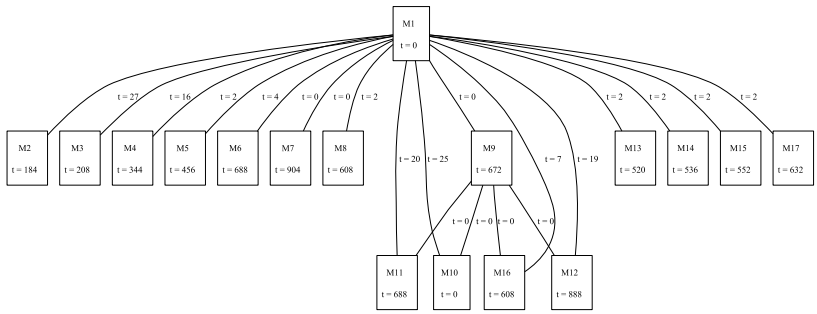


Figure 6.5: Case study 1 model.

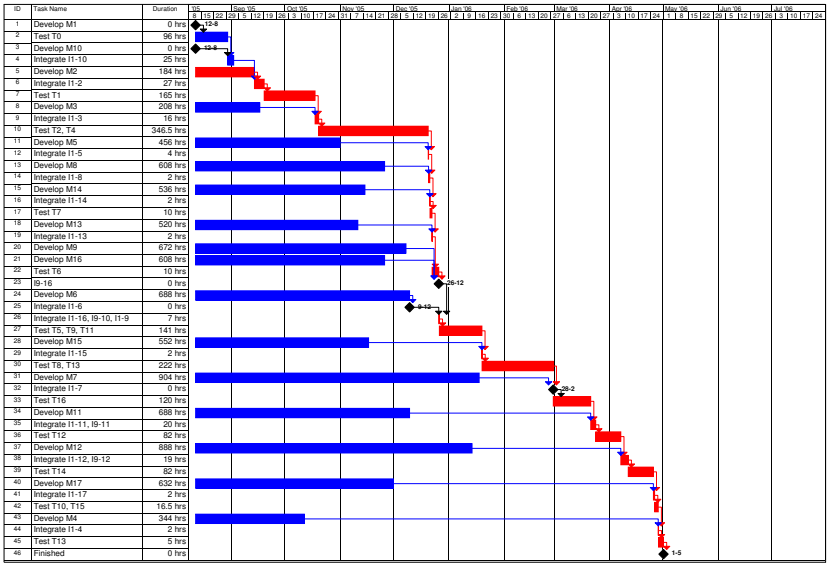
Element	Number	Min and max times
Modules	94	0 to 880 hour
Interfaces	113	0 to 40 hour
Tests	66	4 to 80 hour

Table 6.4: Case study 2 model properties.

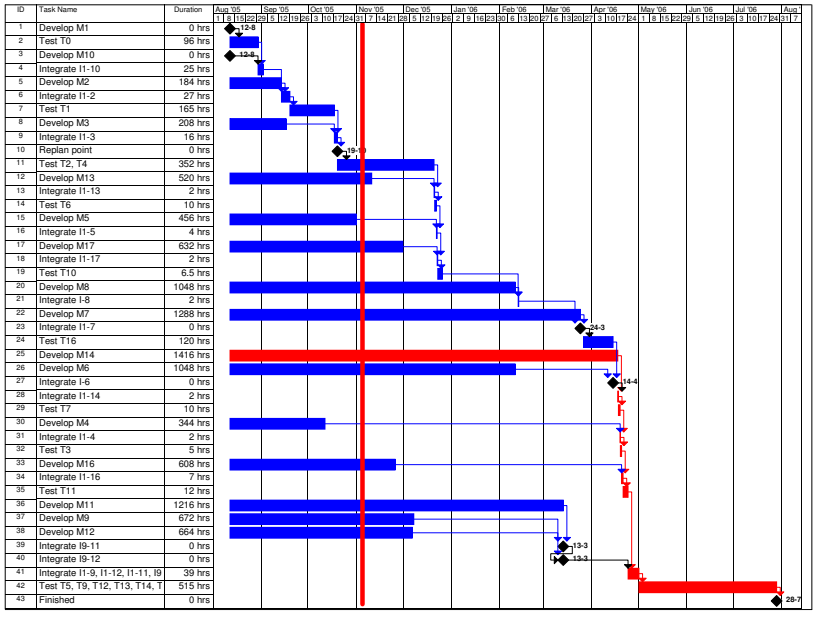
Case study 2

In this case study, two prototypes that have been developed in parallel are used to test some specific requirements of a new type of system. These two prototypes have been built from scratch, so no old system type is upgraded. An important detail of the problem is that 80% of the 66 tests are required to be performed on a specific system while 20% of the tests can be performed on either the first or the second prototype. Therefore, the two prototypes cannot be considered separately but have to be considered as one system. This means that both prototypes are defined in one model to create the optimal combined integration and test plan. Afterwards, the individual integration plans for both prototypes can be retrieved from the combined plan. The properties of the combined model are shown in Table 6.4.

The solution to this problem is unfortunately too large to be shown. The total duration of this plan is 1346 hours. The plan that was created manually by an engineer without using this method takes 1536 hours to perform. This is mainly due to the fact that tests are scheduled less efficiently over the two prototypes compared to the optimal plan. The optimal plan is therefore more than 10% shorter than the plan created manually. Note that we compare two initial plans with each other and not the actually executed plans. These initial plans do not contain the disturbances that may occur during the integration and test phase (although new plans can be created automatically as shown in the previous case study).



(a) Case study 1 solution represented as an MS project Gantt chart



(b) Case study 1 replan solution represented as an MS project Gantt chart

Figure 6.6: Case study 1 solutions.

6.5 Conclusions

In this chapter, we have introduced a method that allows to create optimal integration and test plans for the integration and test phase during the development of a complex system. This method consists of: 1) defining a model of the problem, 2) creating a plan and 3) executing the plan. Two case studies within the development of new ASML lithographic systems showed the benefits of the method, which are:

- The integration and test plans created with the proposed method are optimal and may therefore be shorter than manually created plans. The second case study shows that the optimal plan is more than 10% shorter than a manually generated plan.
- Planning and re-planning effort can be reduced. The first case study shows that it is very easy to re-plan when certain modules arrive later than planned. The only step that has to be performed is updating the model with the new times. The plan can then be updated automatically. Unfortunately, we cannot give any hard numbers on the actual effort reduction because the method has not been used on a large scale yet.

Another benefit of this method is the actual model. The model can be used as a knowledge container and denotes how the integration and test problem is defined in a very simple and precise way. This makes it easy to review the model with peer engineers. The planning method does not influence the quality of the system because the selection of tests is not taken into account. This is different from [21] where we used the presented method in combination with a test selection method to determine the optimal integration and test plan.

In this chapter, we have shown that the integration and test planning method can be used to optimize an integration and test plan for the development of a new system. However, this method is used to solve other problems, such as the optimization of integration and test plans for (evolutionary) software releases and the optimization of integration and test plans for the manufacturing of multiple systems. Of course, the presented method can also be used to optimize integration and test plans of complex systems other than lithographic systems. In the case studies we did not use lithographic system specific properties. Although we did not perform actual studies with other types of systems, the method may also be suitable for systems that have integration and test phases where large numbers of components developed in parallel should be integrated and where time to market is crucial.

Chapter 7

Model-based integration and testing in practice

Authors: N.C.W.M. Braspenning, J.M. van de Mortel-Fronczak, H.A.J. Neerhof, J.E. Rooda

7.1 Introduction

High-tech multidisciplinary systems like wafer scanners, electronic microscopes and high-speed printers are becoming more complex every day. Growing system complexity also increases the effort (in terms of lead time, cost, resources) needed for the, so-called, *integration and test phases*. During these phases, the system is integrated by combining component realizations and, subsequently, tested against the system requirements. Existing industrial practice shows that the main effort of system development is shifting from the design and implementation phases to the integration and test phases [32], in which finding and fixing problems can be up to 100 times more expensive than in the earlier requirements and design phases [14]. As a result, the negative influence of the integration and test phases on the Time–Quality–Cost (T-Q-C) business drivers of ASML (see Chapter 3) is continuously growing and this trend should be countered.

Literature reports a wealth of research proposing a *model-based* way of working to counter the increase of system development effort, like requirements modeling [34], model-based design [78], model-based code generation [60], hardware-software co-simulation [108], and model-based testing [33]; see also Chapters 9 and 10. In most cases, however, these model-based techniques are investigated in isolation, and little work is reported on combining these techniques into an overall method. Although model-based systems engineering [89] and OMG's model-driven architecture [72] (for software only systems) are such overall model-based methods, these methods mainly focus on the requirements, design, and implementation phases, rather than on the integration and test phases. Furthermore, literature barely mentions realistic industrial

applications of such methods, at least not for high-tech multidisciplinary systems.

Our research focuses on a method of *Model-Based Integration and Testing*, MBI&T for short. In this method, formal and executable models of system components (e.g., software, mechanics, electronics) that are not yet realized are integrated with available realizations of other components, establishing a *model-based integrated system*. Such a model-based integrated system can be established much earlier compared to a real integrated system, and it can effectively be used for early model-based system analysis and system testing.

This chapter, which is based on [28, 30], illustrates the application of the MBI&T method to the development of an ASML wafer scanner (see Chapter 2). The goal of the case study and this chapter is twofold: (1) to show the feasibility and potential of the proposed MBI&T method to reduce the integration and test effort of industrial systems; (2) to investigate the applicability and usability of the χ (Chi) tool set [114] as integrated tool support for all aspects of the MBI&T method.

The structure of the chapter is as follows. Section 7.2 gives a general description of the MBI&T method. Section 7.3 introduces the industrial case study to which the MBI&T method has been applied. The activities that have been performed in the case study are described in Sections 7.4 (modeling in χ), 7.5 (simulation), 7.6 (translation to and verification with UPPAAL), and 7.7 (integration and testing of models and realizations). Finally, the conclusions are drawn in Section 7.8.

7.2 Model-based integration and testing method

In current industrial practice, the system development process is subdivided into multiple concurrent component development processes. Subsequently, the resulting components (e.g., mechanics, electronics, software) are integrated into the system; see also Chapter 1.

The development process of a system S that consists of n components $C_{1..n}$ (in this chapter, a set $\{A_1, \ldots, A_i, \ldots, A_n\}$ is denoted by $A_{1..n}$) starts with the system requirements R and system design D. After that, each component is developed. The development process of a component $C_i \in C_{1..n}$ consists of three phases: requirements definition, design, and realization. Each of these phases results in a different representation form of the component, namely the requirements R_i, the design D_i, and the realization Z_i. The component realizations $Z_{1..n}$ should interact and cooperate according to system design D in order to fulfill the system requirements R. The component interaction as designed in D is realized by integrating components via an infrastructure I, e.g., using nuts and bolts (mechanical infrastructure), signal cables (electronic infrastructure), or communication networks (software or model infrastructure). The integration of realizations $Z_{1..n}$ by means of infrastructure I, denoted by $\{Z_{1..n}\}_I$, results in the realization of system S.

Figure 7.1 shows a graphical representation of the current development process of system S. The arrows depict the different development phases and the boxes depict the different representation forms of the system and the components. The rounded

rectangle depicts the infrastructure I that connects the components. For simplicity, the figure shows a 'sequential' development process, i.e., a phase starts when the previous phase has been finished. In practice, however, different phases are executed in parallel in order to meet the strict time-to-market constraints. Furthermore, the real system development process usually has a more incremental and iterative nature, involving multiple versions of the requirements, designs, and realizations, and feedback loops between different phases, e.g., from the realization phase back to the design phase.

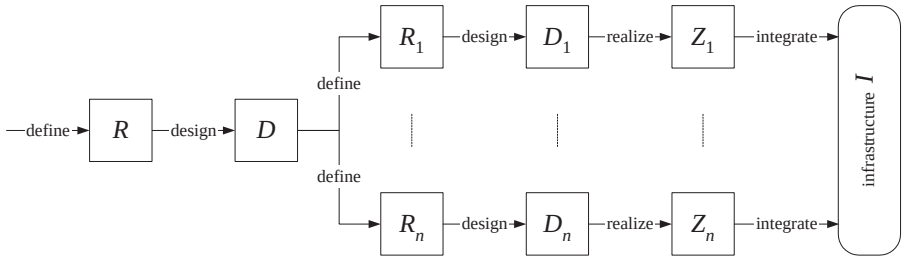


Figure 7.1: Current system development process.

In this chapter, we focus on analysis and testing on the *system level* rather than on the *component level*, since the behavior on the system level usually receives less attention during the development phase and causes more problems during integration. In the current way of working, only two system level analysis techniques can be applied. On the one hand, the consistency between requirements and designs on the component level and on the system level can be checked, i.e., $R_{1..n}$ versus R and $D_{1..n}$ versus D. This usually boils down to reviewing and comparing lots of documents, which can be a tedious and difficult task. On the other hand, the integrated system realization $\{Z_{1..n}\}_I$ can be tested against the system requirements R, which requires that all components are realized and integrated. This requirement means that if problems occur and the realizations, or even worse, the designs need to be fixed during the integration and test phases, the effort invested in these phases increases and on-time shipment of the system is directly threatened. If integration problems could be detected and prevented at an earlier stage of development, the effort invested in the integration and test phases would be reduced and distributed over a wider time frame and the final integration and test phases would become less critical. As a result, the system could be shipped earlier, i.e., an improvement of time-to-market T, or the saved test time could be used to further increase the system quality Q.

We propose a model-based integration and testing (MBI&T) method to reduce integration and test effort and its negative influence on the Time–Quality–Cost business drivers. This method takes the design documentation D_i of the components C_i as a starting point and represents them by formal and executable models M_i, depicted by the circles in Figure 7.2. The requirements documentation R and R_i is used to

formulate the properties of the system and components. An infrastructure I is used that allows the integration of all possible combinations of models $M_{1..n}$ and realizations $Z_{1..n}$, such that they interact according to the system design D. As an example, assume that $n = 2$, i.e., the depicted components C_1 and $C_n = C_2$ are the only components of the system. Then Figure 7.2 shows, corresponding to the depicted integration 'switches', the model-based integrated system $\{M_1, Z_2\}_I$. In the MBI&T method, different representation forms of components (models and realizations) need to be connected, which requires different forms of infrastructure I. In this chapter, however, we abstract from these different forms of infrastructure and only consider the generic infrastructure I. We refer to [30] for more details on the modeling, analysis and implementation of different forms of infrastructure in the MBI&T method.

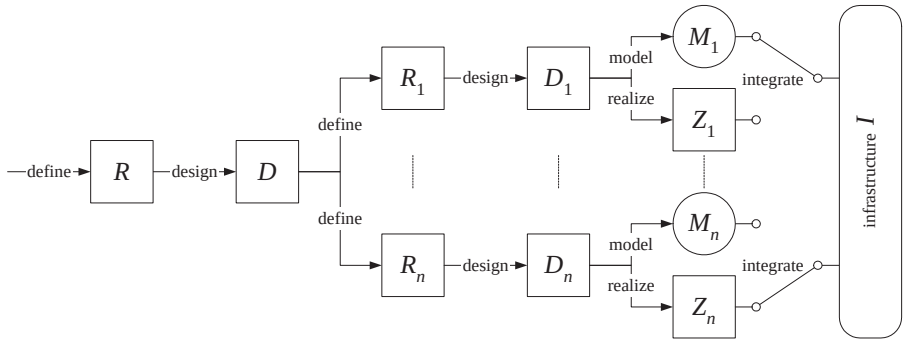


Figure 7.2: System development process in the MBI&T method.

The MBI&T method consists of the following steps, in which different model-based techniques are used to analyze and test the system at an early stage. Although these steps and Figure 7.2 show a sequential procedure for the MBI&T method for simplicity reasons, there will be more parallelism, increments and iterations in reality.

Step 1: Modeling the system components $M_{1..n}$ and the infrastructure, based on design documentation $D_{1..n}$ and D, respectively.

Step 2: Model-based analysis of the integrated system model $\{M_{1..n}\}_I$, using techniques like simulation (2a) and model checking (2b) to analyze particular scenarios and system properties derived from R and D.

Step 3: As soon as the realization of a component becomes available:

a. Model-based component testing of the component's realization with respect to its model, i.e., Z_i with respect to M_i, using automatic model-based testing techniques and tools.

b. Replacement of the component's model M_i by its realization Z_i using an infrastructure I that enables the integration of Z_i with the models and realizations of all other components.

c. Model-based system testing of the integrated system obtained in step 3b, by executing test cases derived from R and D.

Step 4: After all models have been substituted by realizations: testing of the complete system realization $\{Z_{1..n}\}_I$ by executing test cases derived from R and D. Note that only this step is performed in the current system development process as well.

In principle, the MBI&T method allows the use of different specification languages and tools, as long as they are suitable for modeling, analysis, and testing of the considered aspects of the considered components. In the presented case study, all components of the system are modeled in the process algebraic language χ [79, 4]. The χ language is intended for modeling, simulation, verification, and real-time control of discrete-event, continuous or combined, so-called hybrid, systems. The χ tool set [114] allows modeling and simulation of χ models, as well as their translation to different formalisms to enable verification of χ models using different model checking tools [16]. The χ language and simulator have been successfully applied to a large number of industrial cases, such as integrated circuit manufacturing plants, process industry plants, and machine control systems [54, 6, 87]. In the case study, we use the translation scheme from [17] to translate the χ model to UPPAAL timed automata [11], which is the same class of models used in Chapter 9 for timed model-based testing. Subsequently, the UPPAAL model checker [120] is used to verify system properties such as deadlock freeness, liveness, safety, and temporal properties. Although not presented in this chapter, χ models can also be used for automatic component testing using the model-based testing tool TORX [119], as reported in [29].

7.3 Case study: ASML EUV machine

To show proof of concept and to evaluate the MBI&T method, the method was applied to a new type of wafer scanner being developed within ASML, in which extreme ultra violet (EUV) light is used for exposing wafers. One of the most important technical challenges in the development of this lithography system is the need for strict vacuum conditions, since EUV light is is absorbed by nearly all materials, including air.

In the case study presented in this chapter, the focus is on the interaction between the vacuum system component C_v that controls the vacuum conditions and the source component C_s that generates the EUV light. These components need close cooperation to provide correct vacuum conditions and correct EUV light properties at all times. Since the internal states of these components are interdependent (e.g., the source may only be active under certain vacuum conditions to avoid machine damage), some combinations of component states are not allowed and should be prevented.

Figure 7.3 shows the components and interfaces involved in the case study. To exchange information about their internal states, the vacuum system C_v and the source C_s are connected by an interface consisting of four latches[1], three latches from vacuum system to source and one latch from source to vacuum system:

- 'vented': when active, this latch indicates that the vacuum system is vented.
- 'pre-vacuum': when active, this latch indicates that the vacuum conditions are sufficient to activate the source, however not sufficient for exposure.
- 'exposure': when active, this latch indicates that the vacuum conditions are right for exposure.
- 'active': when active, this latch indicates that the source is active and that the vacuum system is not allowed to go to the vented state (to avoid machine damage).

Besides these latches to interact with the source, the vacuum system provides a function *goto_state* to the environment of the system, which is represented here as component C_e. The environment, e.g., a control component or a vacuum system operator, can send a request via *goto_state_req* to instruct the vacuum system to go to either the vacuum or the vented state. After receiving a request from the environment, the vacuum system immediately sends a reply 'OK' via *goto_state_rep*. Note that, by design, this reply does not indicate that the requested state is reached; it only indicates that the request is successfully received and that the vacuum system will perform the actions necessary to get to the requested state. The progress of these actions and the state of the vacuum system can only be observed via the vacuum system user interface without explicit notification that a certain state is reached, which was sufficient for the system design considered in the case study.

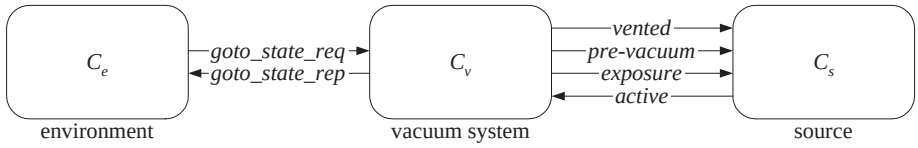


Figure 7.3: Components and interfaces involved in the case study.

The behavior of the integrated system under nominal conditions is depicted in the message sequence chart in Figure 7.4. This figure shows the different states of the components and the communication between them for the vacuum sequence. In order to go to the vacuum state, first the 'vented' latch is deactivated after which the vacuum

[1]Latch: electronic circuit based on sequential logic with inputs 'set' and 'reset' that is capable of storing one bit of information, i.e., a continuous high or a low voltage.

pumps are started and some initial preparation actions are executed by the source. After some pumping down, when the vacuum conditions are sufficient to activate the source, the ‘pre-vacuum’ and subsequently the ‘active’ latch are activated, and the source goes to the active state. Finally, when the vacuum conditions are right for exposure, the ‘exposure’ latch is activated and the source goes to the exposure state. For the other way around, going from vacuum/exposure conditions to vented/inactive conditions, a similar, reversed sequence has been specified.

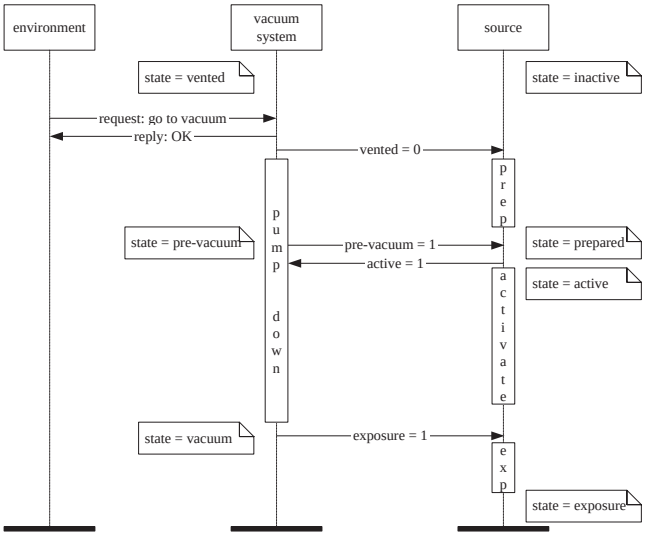


Figure 7.4: Nominal system behavior for the vacuum sequence.

The nominal sequence described above does not cover preemption. The environment can interrupt the sequence at any time by a new request, and the vacuum system should handle these interrupts. For instance, when the vacuum system operator decides to go back to the vented state while the vacuum system is performing the vacuum sequence (i.e., going from vented to vacuum as shown in Figure 7.4), the vacuum system should immediately interrupt the vacuum sequence and start with the venting sequence to go to the vented state. Finally, errors with different severity levels can be raised during operation, which lead to specified non-nominal behavior that is not covered in the message sequence chart.

In the remainder of this chapter, we describe the application of all steps of the MBI&T method, except steps 3a and 4. Step 3a, automatic model-based component testing using χ models, has been applied in a similar case study, as reported in [29]; see also Chapters 9 and 10 for more details on model-based testing. Although the MBI&T method contributed to real system testing in step 4 by detecting and preventing errors at an earlier stage (potentially reducing the time needed for step 4), we did not actively participate in this step of the case study.

7.4 Step 1: Modeling the components in χ

The informal design documentation was taken as a starting point for modeling the vacuum system and the source as χ processes. The system level design documentation (corresponding to D) described the interfaces between the vacuum system and source as shown in Figure 7.3. The design documentation for the source component, D_s, was reasonably complete and contained a good overview of the behavior in the form of an informal state diagram, which clearly showed the possible actions and communications for each state of the source. However, the design documentation of the vacuum system component, D_v, only described the internal actions for the vacuum and venting sequences, and did not contain information about the communication with the source. It became clear that the designers of the vacuum system were not yet fully aware of the communication behavior between their component and the source component. In general, the design documentation for both components described the nominal behavior only and hardly mentioned the handling of exceptional behavior, and also the action durations were not completely specified.

During our modeling activities, we obtained the missing information about the component designs by combining knowledge of both components and by discussion with the engineers involved. For example, the specifications of the communication of the vacuum system that was missing in D_v could be derived from the system and source design documents D and D_s. The updated and more complete design specifications were validated for their correctness by discussion with the designers of the vacuum system and source.

We experienced that in most cases, the issues that arose during the modeling activities in fact indicated unknown or incomplete design issues like missing states, obsolete states, or errors in the communication sequence. By incremental modeling, intermediate simulation, discussion, and design review, the system specification documentation was further corrected and completed, which also helped the engineers to obtain a better system overview.

There were some remaining modeling and design issues, for which no solution was available or for which the corresponding behavior was not known at all, even by the engineers involved. According to the engineers, the system behavior concerning these remaining issues was not important because it would never occur in the real system. Therefore, the behavior concerning these issues was abstracted in the model by putting an explicit indication of 'undefined behavior'. In step 2b of the case study, it will be verified whether this undefined behavior indeed can never occur.

To give an impression of the resulting model, Figure 7.5 shows the process layout of the system model and Figure 7.6 shows the χ code corresponding to a part of the source model M_s. Earlier, in Figure 7.3, we depicted the system design with three main components (the environment, the vacuum system and the source) and the communication between them. Figure 7.5 shows how the χ processes of the system model (depicted by rounded rectangles) and the communication between them (depicted by arrows) are mapped onto this system design. The environment is modeled as a single

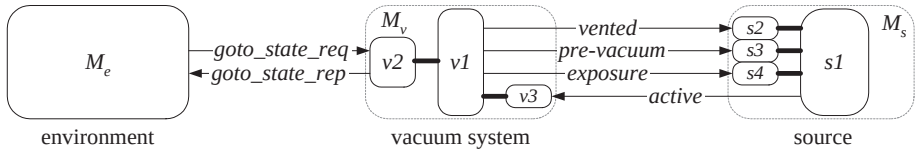


Figure 7.5: Process layout of the χ system model.

sequential process M_e that can be configured to send requests and receive replies at certain points in time, depending on the analysis technique. The vacuum system M_v is modeled as a parallel composition of the processes ($v1 \parallel v2 \parallel v3$), in which the parallel composition operator $\parallel$ synchronizes the processes on time and on communication actions. Process $v1$ is the core process and describes the internal behavior of the vacuum system, while the processes $v2$ and $v3$ model the interaction with the environment and the latch interaction with the source, respectively. The source M_s is modeled as a parallel composition of the processes ($s1 \parallel s2 \parallel s3 \parallel s4$), in which process $s1$ is the core process that models the internal behavior and the processes $s2$, $s3$, and $s4$ model the latch interaction with the vacuum system. Finally, the bold lines between processes of one component depict shared variables (two for the vacuum system and three for the source), which are used to exchange data between processes of one component.

```
1   M_s =
2   (*      ( src_state = inactive → ...
3             [] src_state = prepared → ...
4             [] src_state = active → skip
5              ; ( error = 5 → skip; Δ60; manual := true
6                [] error = 4 → manual := true
7                [] error < 4 → skip
8                 ; newvalue → newvalue := false
9                 ; (  vnt                          → error := 5
10                   [] ¬vnt  ∧  pre    ∧  exp      → src_state := 4
11                   [] ¬vnt  ∧  ¬pre   ∧  exp      → undefined := true
12                   [] ¬vnt  ∧  pre    ∧  ¬exp     → skip
13                   [] ¬vnt  ∧  ¬pre   ∧  ¬exp     → Δ60
14                                                    ; src_state := 2
15                                                    ; active !! false
16                )
17             )
18            [] src_state = exposure → ...
19            )
20  || *    (vented ?? vnt; newvalue := true)
21  || *    (pre_vacuum ?? pre; newvalue := true)
22  || *    (expose ?? exp; newvalue := true)
23  )
```

Figure 7.6: Part of the source model M_s in χ.

Figure 7.6 shows a part of the source model M_s, in which process $s1$ is shown on lines 2-19, and the processes $s2$, $s3$, $s4$ are shown on lines 20, 21, and 22, respectively.

For simplicity, the model of the core process $s1$ only shows the behavior in the active state and all omitted parts are denoted by three dots (...). When process $s1$ is in the active state, the process checks if there is an error and, depending on the severity of the error, it takes certain actions to prevent further problems. If there is no severe error ($error < 4$), the source process checks if any of the latch values has changed, indicated by the variable $newvalue$. If the $newvalue$ guard is false, the process waits until it becomes true (i.e., until a new value is received via one of the latches). If $newvalue$ is true, it is reset to false without a delay and the process continues by checking the current values of the three incoming latches, represented by the variables vnt, pre, and exp. Depending on the latch values, the source performs different actions, e.g., raising an error, going to another state, reaching a situation with undefined behavior, performing an internal action, or changing the value of a latch. More details on the χ system model can be found in [28].

7.5 Step 2a: Simulation of the integrated system model

In order to analyze the behavior of the integrated vacuum system-source model by means of simulation, different scenarios are needed that focus on different aspects of the system. A good source for possible scenarios is the intended system behavior specified in the system requirements and design documentation, R and D. Unfortunately, in the case study only one scenario could directly be derived from the documentation. This scenario corresponds to the nominal behavior of the system.

From ASML testing experience, it is known that analyzing only the nominal behavior is not sufficient. In most cases, it is the exceptional behavior which gives the problems, since this behavior is less documented and thus less clear when compared to the nominal behavior. Therefore, it is very important to analyze the exceptional behavior as soon as possible. Unfortunately, no simulation scenarios for exceptional behavior could be directly derived from the documentation. However, based on our system overview obtained by modeling the components, and by discussion with the involved engineers, four additional scenarios for exceptional behavior analysis were derived. These scenarios cover the behavior of the system when the vacuum and venting sequences are interrupted at certain points in time.

Besides incomplete documentation, there is another problem with the analysis of exceptional behavior in the current, non model-based, way of working. Since only realizations can be used for system analysis, it may be difficult or expensive to create the non-nominal circumstances that are necessary to analyze the exceptional behavior, for example, a broken component. Since the MBI&T method uses models for system analysis, creating these non-nominal circumstances is much easier and cheaper.

In the case study, we used specific configurations of the environment model M_e to analyze the integrated system behavior for the five scenarios mentioned above, one with nominal and four with exceptional behavior. The simulation results were visualized by means of animated automata, message sequence charts, and Gantt charts. One situation with incorrect behavior was detected, which surprisingly also occurred

in the nominal behavior scenario. The incorrect behavior occurs during the venting sequence, in which the vacuum system first deactivates the 'exposure' and 'pre-vacuum' latches. According to its design, the source should first observe the deactivated 'exposure' latch and perform some actions before observing the deactivated 'pre-vacuum' latch. However, the vacuum system was designed to deactivate both latches at the same time, which means that the source can also receive the deactivated 'pre-vacuum' latch during the actions it performs to reach the prepared state, or even before receiving the deactivated 'exposure' latch. In both cases, the source raises an error and switches to manual mode, which is certainly not acceptable for nominal behavior. Further diagnosis showed that this incorrect behavior indeed was an integration problem between the vacuum system and the source, which could now be solved early in the design phase.

7.6 Step 2b: Verification with UPPAAL

During simulation (validation) some problems were discovered and solved, which increased the confidence, but does not prove the correctness of the model. To check whether the model behaves correctly in all possible scenarios and to gain more knowledge about the system, it has to be verified. For the verification step, the environment model M_e was configured such that any possible delay can occur between subsequent vacuum system requests, which will exhaustively be analyzed by model checking tools such as UPPAAL.

UPPAAL is a tool for modeling, simulation, and verification. A system, modeled as a network of UPPAAL timed automata, can be simulated by the UPPAAL simulator and verified by the UPPAAL model checker. To be able to verify χ models in UPPAAL, a translation scheme from the process algebraic language χ to UPPAAL timed automata has been formally defined and the proofs of its correctness have been given in [18]. The translation scheme is defined for a subset of χ that consists of all χ models specified as parallel composition of one or more sequential processes. The translation scheme from χ to UPPAAL has been implemented and integrated into the χ tool set.

Since the original model was created without considering its translation to UPPAAL, it uses some constructs, for which no translation is defined. However, these modeling constructs could easily be transformed to make the model suitable for automatic translation to UPPAAL automata. Figure 7.7 shows the generated UPPAAL automata for the source component, which corresponds to the partial χ source model M_s of Figure 7.6. The following properties, derived from system requirements R and system design D, were verified using the UPPAAL model checker. The properties are expressed in the timed computation tree logic (TCTL) [11], in which *A[]* informally means that, for all traces, the statement behind it should hold *always*, while *A<>* informally means that, for all traces, the statement behind it should hold *eventually* (i.e., after some time).

(1) Deadlock freeness: *A[] deadlock imply env.end*, where *env.end* denotes the location of the environment automaton indicating successful termination, which is the only location at which deadlock (no further steps possible) is allowed.

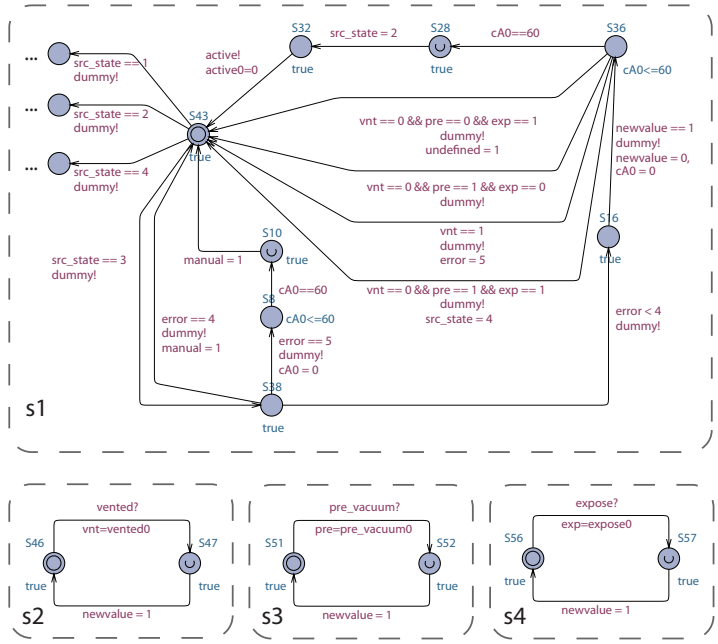


Figure 7.7: Generated UPPAAL automata for the source.

(2) Live-lock freeness: *A<> env.end*. When all requests are processed and confirmation is received, the environment process terminates. Based on this we can state that if the environment automaton eventually reaches its end state *env.end* for all traces, there is no live-lock in the system.

(3) No undefined behavior: *A[] undefined == 0*. While modeling, we discovered that in some particular situations the system behavior was unknown, for which the engineers claimed that it would never occur. These situations were modeled by assigning a non-zero value to the variable *undefined* on line 11 in Figure 7.6.

(4) No errors: *A[] error == 0*, where *error* is the variable indicating the error severity level of the source (*error* $>$ *0*), or the absence of an error (*error == 0*).

(5) The vacuum system may not be vented while the source is active to avoid machine damage: *A[] not (vnt and act)*, where the variables *vnt* and *act* indicate the vented state of the vacuum system and the active state of the source, respectively.

(6) The duration of the vacuum and vacuum sequences is at most 6 hours and 1 hour, respectively: *A[] vacuum imply clk* $\leq$ *21600* and *A[] venting imply clk* $\leq$ *3600*, where the variables *vacuum* and *vented* indicate which sequence is being performed, and *clk* is a clock variable used to determine the duration of the performed sequence (in seconds).

During the verification of the model in UPPAAL, the biggest number of states (20510) was explored while verifying the first property. Our system model contains 8 automata with 5 clocks and 29 variables, and the amount of memory used during model checking was just under 1 MB.

During verification of properties 1 and 2, two design errors were found. Both errors are causing deadlock and concern non-determinism in the interleaving of the main process $v1$ and the interrupt handling process $v2$ of the vacuum system. The way to handle this non-determinism in general was not specified in the design documentation, and model verification has indicated this design incompleteness. After informing the involved engineers, an alternative design for the interrupt handling process was proposed and subsequently modeled, after which properties 1 and 2 were satisfied. Property 3 is satisfied by the model, indicating that the engineers were correct when they claimed that the situations with undefined behavior would never occur. Verification of property 4 detected the same design error as found by simulation, i.e., incorrect behavior in the venting sequence, resulting in an error level larger than zero. Besides that, no other errors were raised in any trace of the model. Two minor mistakes were discovered by verification of properties 4 and 5. To verify the property 6, we decorated the model with additional boolean variables *vacuum* and *vented* to indicate which sequence is being performed and a clock *clk* to determine the duration of a sequence, without changing the behavior of the model. Both queries of property 6 are satisfied.

All design errors found with simulation and verification were discussed with the ASML engineers, and subsequently fixes were applied to the design and, correspondingly, to the model. The fixed model was verified again and now all properties as described above are satisfied by the model. The verification results of the fixed model give enough confidence that the model is a correct representation of the system design, making it suitable for model-based integration and system testing.

7.7 Step 3: Model-based integration and system testing

In this step of the case study, the source model M_s was replaced by its realization Z_s, i.e., the real EUV light source. This implies that the interaction between the vacuum system model M_v and the source realization Z_s needs to be established. In reality, the latch communication is established via a multi-pin cable, of which four pins relate to the four latches. In order to integrate M_v and Z_s, they must be able to communicate with each other via this multi-pin cable.

Integration of the environment model M_e, the vacuum system model M_v, and the source realization Z_s is achieved by using a model-based integration infrastructure based on publish-subscribe middle-ware [45], together with appropriate component connectors and a real-time χ simulator, as described in [30]. Using this integration infrastructure, the components communicate by publishing messages of a certain topic to the middle-ware, which are subsequently delivered to the components that are subscribed to the same topic.

The function call interaction type between M_e and M_v is implemented in the in-

tegration infrastructure by defining the publish-subscribe topics *request* and *reply* and by configuring the published and subscribed messages in M_e and M_v accordingly. For example, M_e is a publisher of the *request* topic and M_v is subscribed to the *request* topic. For the latch interaction type between M_v and Z_s, a topic for each of the four latches is defined. This results in the integration infrastructure configuration as shown in Figure 7.8, in which the rounded rectangle of infrastructure I as in Figure 7.2 is instantiated with the model-based integration infrastructure. In the figure, the vertical double headed arrow depicts the publish-subscribe middle-ware, the rectangles depict the component connectors, and the arrows depict the publish-subscribe communication with the associated topics. Note that the arrows for the three SR-latches from M_v to Z_s ('vented', 'pre-vacuum', and 'exposure') are combined and denoted by *3*latch*.

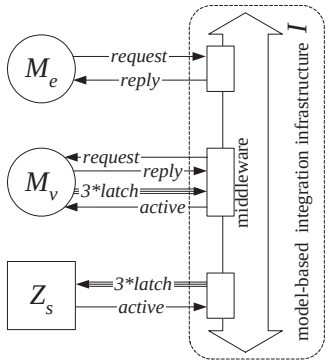


Figure 7.8: Model-based integration infrastructure for the case study.

Using the χ tool set and a suitable publish-subscribe middle-ware [86], the middle-ware configuration described above and the connectors for the models are automatically generated from the χ system model. The realization connectors, however, are case specific and should be separately developed, e.g., using the results from Chapter 14. In the case study, the connector for Z_s should adapt between the real latch communication via the multi-pin cable and the publish-subscribe communication used in the middle-ware. To achieve this, a SW/HW adapter is used in the form of a remote I/O unit that allows different forms of analog and digital input and output, e.g., from Opto 22 [93] or National Instruments [88]. In the case study, we used a digital input module to receive values of the 'active' latch and a digital output module to set values of the 'vented', 'pre-vacuum', and 'exposure' latches.

Using the model-based integration infrastructure as shown in Figure 7.8 and a real-time χ simulator, we were able to integrate and test the model-based integrated system $\{M_e, M_v, Z_s\}_I$ significantly earlier (20 weeks before all realizations were available and integrated) and cheaper (no critical clean room time was needed as for real system testing). Similar to the simulation analysis in step 2a of the case study, the environ-

ment model was configured with specific scenarios to test the model-based integrated system on different aspects, for both nominal and non-nominal behavior. Creating non-nominal circumstances for testing was easy in a model environment, whereas this may be quite difficult and time consuming when testing with realizations.

Besides showing the feasibility of this step of the MBI&T method, the profitability also became clear because six integration problems were detected. The problems, which appeared to be caused by implementation errors in the source, could potentially lead to source damage (i.e., long down times) and unnecessary waiting in the source (i.e., long test times) during the real integration and test phases in the clean room. Although not automated as in Chapter 12, the models supported immediate diagnosis and repairing of the detected errors, as well as immediate retesting of the repaired system. This means that the model-based integration and test activities probably saved several days of expensive clean room time during real integration and testing 20 weeks later (if the errors would remain undetected until that time). In the period after the model-based system tests (i.e., in step 4 of the case study, real system testing), no additional errors in the source realization were found, at least not for the aspects that were analyzed and tested using the MBI&T method. This means that no expensive fixing was necessary during real system integration and testing thanks to the MBI&T method.

The total amount of time used for testing, diagnosis, repairing, and retesting of the model-based integrated system was significantly lower than the estimated amount of time that would be required to perform the same tests on the real system: one half of a day against four days. Several reasons can be identified for this time reduction. First, experience in real system testing shows that setting up the system for testing can be very time consuming. In the case study, for example, a certain test may require that the initial vacuum system state is vented while the end state of the previous test was vacuum. This also holds for the re-execution of tests that change the system state (e.g., a test that starts in the vented state and ends in the vacuum state). In model-based system testing, less test setup time is required because setting up a model to another initial state usually boils down to changing some variables (e.g., changing the initial value of the vacuum system state variable). Second, testing with realizations may also suffer from time lost on solving minor system problems that are unimportant for the tests. In the case study, for example, the real vacuum system contains many potential problem sources (e.g., a malfunctioning sensor or valve) that could result in a system that is unable to initialize, thus prohibiting test execution. Model-based system testing does not suffer from this issue, since the models only contain the behavior that is important for the tests and abstract from the minor problems that potentially prohibit test execution. Third, the use of models for testing reduces the time spent on diagnosis of errors when compared to real system testing. On the one hand, the number of problem sources that could potentially cause an error is reduced since the models only contain the behavior important for the tests, i.e., abstracting from all other components and aspects which form potential problem sources in real system testing. On the other hand, the complete insight in and control over the models makes the distinction between the potential problem sources more clear.

7.8 Conclusions

The goal of the case study and this chapter was twofold: (1) to show the feasibility and potential of the proposed MBI&T method to reduce the integration and test effort of industrial systems; (2) to investigate the applicability and usability of the χ tool set as integrated tool support for all aspects of the MBI&T method.

Application of the MBI&T method proved to be feasible and successful for the realistic industrial case study, showing relevant advantages for the system development process. First, the modeling activities helped to clarify, correct, and complete the design documentation. By simulation and verification, five design errors were detected and fixed earlier and cheaper when compared to current system development. Two design errors were discovered by verification only, which both concerned the non-deterministic behavior of parallel processes. This is difficult, if not impossible, to understand and to analyze by simulation or by reviewing the design documentation. This illustrates that verification should be used for designing real industrial systems, which often involve both high-level parallelism and non-deterministic behavior. Finally, a part of the model was integrated with a realized component to enable early, fast, and cheap model-based system testing. Again, six integration problems were detected and fixed, saving significant amounts of time and rework during the real integration and test phases. After diagnosis, these integration problems appeared to be caused by implementation errors in the source. The errors could potentially lead to source damage and unnecessary waiting in the source during the real integration and test phases in the clean room. This means that the MBI&T method potentially saved several days of expensive clean room time that would be spent on diagnosis, fixing, and retesting. This clearly shows the applicability and value of the MBI&T method to reduce the integration and test effort of industrial system development and its negative influence on the Time–Quality–Cost business drivers.

The χ tool set is suitable for practical application of all MBI&T activities: modeling, simulation, verification, component testing, model-based integration and integrated system testing. The aspects considered in the case study (supervisory machine control in software, interrupt behavior, electronic communication) can easily be modeled in χ. The rather small state space of the modeled system (20510 states) indicates that similar sized or even more complex industrial systems can be handled.

Application of the MBI&T method is only beneficial when the potential effort reduction for integration and testing outweighs the required modeling effort. The difficulties of creating the models, choosing the right modeling scope and level of abstraction, and performing the validation, verification and tests with the models should not be underestimated and require certain skills and experience. Chapter 8 describes how modeling effort can be taken into account when making decisions on when and where it is most profitable to use models in the integration and test process.

Chapter 8

Using models in the integration and testing process

Authors: N.C.W.M. Braspenning, J.M. van de Mortel-Fronczak, D.O. van der Ploeg, J.E. Rooda

8.1 Introduction

This chapter, which is based on [31], describes how the Model-Based Integration and Testing (MBI&T) method described in Chapter 7 can be used in an industrial integration and test process such as used at ASML. In particular, we focus on a scenario that is common in the integration and test process of many high-tech embedded systems: upgrading a system with new hardware and software to improve the system performance, for example a new sensor with accompanying control software to improve the measurement accuracy. In such a scenario, the goals of integration and testing are to show the functionality and performance of the system upgrade as soon as possible, and to show that the system upgrade does not negatively affect the functionality and performance of the original system. We show how such a scenario is handled in both the current integration and test process and the model-based integration and test process.

Although the use of models in the integration and test process can significantly reduce the integration and test effort, this reduction has to outweigh the additional effort needed to enable model-based integration and testing. For example, the involved components must be modeled, analyzed, and integrated with models or realizations of the other components before the model-based integrated system can be tested. This means that there is a trade-off between the investments and benefits of using models in the integration and test process. In this chapter, we show how such a trade-off analysis can be performed using integration and test sequencing techniques as described in Chapter 6.

The structure of the chapter is as follows. First, Section 8.2 describes the current integration and test process for the system upgrade scenario and introduces the nine test categories that can be distinguished. In Section 8.3, the MBI&T method and techniques are applied to the integration and test process, showing how each of the test categories can be supported by models. Section 8.4 shows how integration and test sequencing techniques can be used to compare the current and model-based integration and test processes, and how this can be used to analyze the trade-off between modeling effort and benefits. Finally, the conclusions are given in Section 8.5.

8.2 Current integration and test process

For the description of the current industrial integration and test process as used within ASML, we consider an existing system that consists of several hardware and software components. The system will be upgraded by implementing some new or improved functionality, which is denoted by a delta sign (Δ). To implement this Δ-functionality, certain components of the original system need to be upgraded, or new components need to be developed and added to the system. Similar to Figure 7.1 in Chapter 7, our view on the development process of this Δ-functionality starts with the global requirements R_Δ and design D_Δ, as shown on the left hand side of Figure 8.1. After that, the software and hardware components for the Δ-functionality, denoted by ΔSW and ΔHW, respectively, are separately developed. The development process of these components consists of three phases: requirements definition, design, and realization, each resulting in a different representation form of the component, namely the requirements $R_{\Delta SW}$ and $R_{\Delta HW}$, the designs $D_{\Delta SW}$ and $D_{\Delta HW}$, and finally the realizations $Z_{\Delta SW}$ and $Z_{\Delta HW}$.

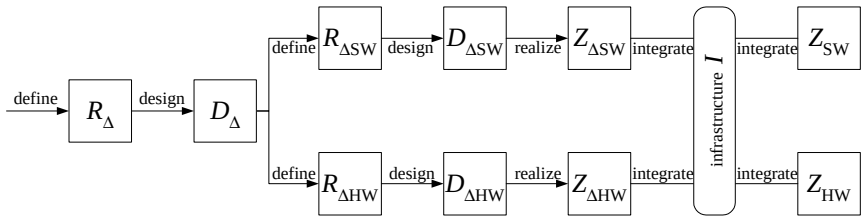


Figure 8.1: Current development and integration of a Δ-functionality.

The right hand side of Figure 8.1 shows the integration of the Δ components $Z_{\Delta SW}$ and $Z_{\Delta HW}$ with the other software and hardware components of the (original) system, which we denote as one software component Z_{SW} and one hardware component Z_{HW}, respectively. The four components are integrated by means of some infrastructure I. Similar to Chapter 7, we abstract from the different forms of infrastructure and only consider the generic infrastructure I (see [30] for more details).

Within the current integration and test process at ASML, nine different test categories can be distinguished, which focus on different aspects of the components or system and require different combinations of realized and integrated components. These nine test categories are listed in Table 8.1, where component integration by means of infrastructure I is denoted by $\{\ldots\}_I$.

Nr	Test category	Required components	Explanation
1	Software qualification	$\{Z_{SW}, Z_{HW}\}_I$ and later $\{Z_{\Delta SW}, Z_{SW}, Z_{HW}\}_I$	Periodic qualification of the so-called 'qualified baseline' or QBL [59], a common software repository that supports all machine types, by testing it on a set of representative hardware systems Z_{HW}.
2	Software component	$Z_{\Delta SW}$	Testing the new software component in isolation.
3	Software integration	$\{Z_{\Delta SW}, Z_{SW}\}_I$	Testing the new software component in combination with the original software system Z_{SW}.
4	Software regression	$\{Z_{\Delta SW}, Z_{SW}, Z_{HW}\}_I$	Testing whether any of the original system functions are negatively affected by the new software component, performed on the original hardware system Z_{HW}.
5	Hardware component	$Z_{\Delta HW}$	Testing the new hardware component in isolation.
6	Hardware integration	$\{Z_{\Delta HW}, Z_{HW}\}_I$	Testing the new hardware component in combination with the original hardware system Z_{HW}.
7	Δ-functionality test bench	$\{Z_{\Delta SW}, Z_{SW}, Z_{\Delta HW}\}_I$	Testing the new Δ-functionality (also called progression testing) on a 'test bench', i.e., a partial hardware system including the new hardware component $Z_{\Delta HW}$, which is used for development tests.
8	Δ-functionality system	$\{Z_{\Delta SW}, Z_{SW}, Z_{\Delta HW}, Z_{HW}\}$	Testing the new Δ-functionality on a complete system, i.e., Z_{HW} upgraded with $Z_{\Delta HW}$.
9	System	$\{Z_{\Delta SW}, Z_{SW}, Z_{\Delta HW}, Z_{HW}\}$	Testing the functionality and performance of the complete system with multiple Δ-functionalities before shipment.

Table 8.1: Test categories in the current integration and test process.

Figure 8.2 shows a typical integration and test process for the system upgrade scenario. From left to right, the sequence of test activities, denoted by vertical lines, is shown. The numbers correspond to the nine test categories mentioned above, and the dots indicate which components are integrated and tested. The horizontal lines depict the lifetime of a component: a dashed line means that the component is being developed; a flag symbol followed by a solid line means that the component realization is available. Note that the figure only shows a sequence, and does not contain information on the possible start times and durations of the activities. The flag symbols and the letters indicate the following milestones: (a) QBL Z_{SW} passes qualification tests; (b) development of $Z_{\Delta SW}$ and $Z_{\Delta HW}$ is started, possibly based on the original system (denoted by dashed upward arrows); (c) $Z_{\Delta SW}$ is available; (d) $Z_{\Delta SW}$ passes software tests and is integrated in the QBL Z_{SW} (denoted by downward arrow); (e) upgraded QBL $\{Z_{\Delta SW}, Z_{SW}\}_I$ passes qualification tests; (f) $Z_{\Delta HW}$ is available; (g) $Z_{\Delta SW}$ and $Z_{\Delta HW}$ pass test bench tests; (h) $Z_{\Delta HW}$ passes hardware integration tests

and the hardware system Z_{HW} is upgraded to $\{Z_{\Delta HW}, Z_{HW}\}_I$ (denoted by downward arrow); (i) similar to the depicted Δ-functionality, the other Δ-functionalities are integrated and tested; (j) complete system with all Δ-functionalities passes tests and is shipped to customer.

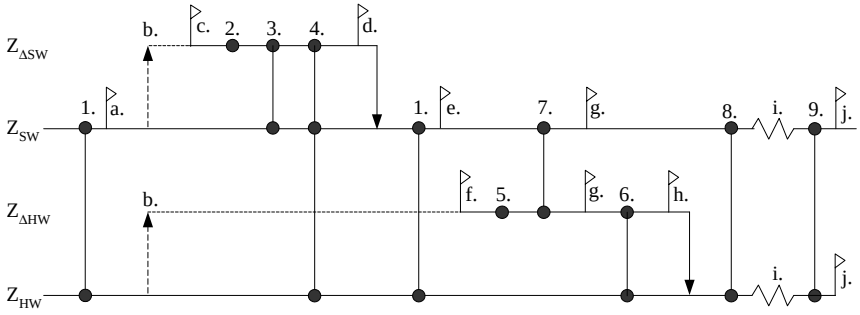


Figure 8.2: Typical integration and test process for system upgrade scenario.

The main disadvantage of the current integration and test process is that the test activities can only be performed when the realizations are available. Especially for testing on system level (test categories 7, 8, and 9) this is problematic, because it means that feedback on the system behavior and performance is obtained late in the process, where fixing the problems is expensive. Several case studies, e.g., the one described in Chapter 7, have shown that early integration and testing is possible by using models in the integration and test process, which is discussed in more detail in the next section.

8.3 Model based integration and test process

Figure 8.3 shows the development and integration of a Δ-functionality with the MBI&T method from Chapter 7, with models $M_{\Delta SW}$, $M_{\Delta HW}$, and M_{HW} of the Δ software component, the Δ hardware component, and the original complete hardware system, respectively. The reason for having M_{HW} but not M_{SW} is explained in [31]. The choice of integrating either the model or the realization of a component, or none of them, is depicted by the integration 'switches'.

The MBI&T activities can be applied to all nine current test categories of Table 8.1. In contrast to the current test activities, in which only realizations are used, the model-based test activities can be performed with models instead of realizations, which has several advantages. First, model-based test activities can be performed earlier since, in general, models are earlier available than realizations. Earlier testing means earlier (and thus cheaper) detection and prevention of problems. Second, testing with models is generally cheaper than testing with realizations. For example, testing with models can be performed on a common computer system using modeling and analysis software

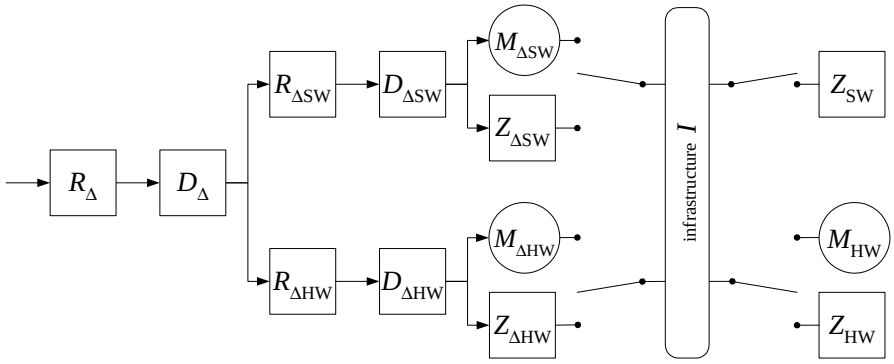


Figure 8.3: Development and integration of Δ-functionality in the MBI&T method.

tools. The test costs in such a desktop environment are much lower than the costs of realization tests, especially when these tests require expensive machine time and clean room facilities in the case of ASML. Finally, the models enable the use of powerful model analysis techniques and tools like simulation and formal verification, which provide a better system overview.

Although the MBI&T techniques can be applied in all nine test categories of the previous section, they can not fully replace testing with realizations, since models are always an abstraction of reality. Sooner or later, when the component and system realizations are available, they will also be tested. However, these tests with realizations can probably be performed faster and cheaper, since the model-based tests already prevented several problems. Furthermore, it may be difficult to perform certain tests with models, such that only realizations can be used for these tests. For example, this may be the case for tests involving components with complex physical interactions (e.g., heat, air flow, vibrations) or tests covering multiple system aspects at once.

The following list identifies all possible test activities for each test category, including both the current test activities from Table 8.1 and the model-based test activities of the MBI&T method. For each of the nine test categories, the test activities with realizations only are marked with a 'Z', and the model-based test activities are marked with an 'M', possibly followed by a letter in the case of multiple model-based test activities. This chapter only describes the model-based test activities of test categories 1, 4 and 7 in more detail. We refer to [31] for a complete overview.

Software qualification testing:

1Za: $\{Z_{SW}, Z_{HW}\}_I$
1Ma: $\{Z_{SW}, M_{HW}\}_I$
1Zb: $\{Z_{\Delta SW}, Z_{SW}, Z_{HW}\}_I$
1Mb: $\{Z_{\Delta SW}, Z_{SW}, M_{HW}\}_I$

Software component testing:

2Z: $Z_{\Delta SW}$
2Ma: $M_{\Delta SW}$
2Mb: $Z_{\Delta SW}$ vs. $M_{\Delta SW}$

Software integration testing:

3Z: $\{Z_{\Delta SW}, Z_{SW}\}_I$
3M: $\{M_{\Delta SW}, Z_{SW}\}_I$

Software regression testing:

4Z: $\{Z_{\Delta SW}, Z_{SW}, Z_{HW}\}_I$
4Ma: $\{M_{\Delta SW}, Z_{SW}, M_{HW}\}_I$
4Mb: $\{M_{\Delta SW}, Z_{SW}, Z_{HW}\}_I$

Hardware component testing:

5Z: $Z_{\Delta HW}$
5Ma: $M_{\Delta HW}$
5Mb: $Z_{\Delta HW}$ vs. $M_{\Delta HW}$

Hardware integration testing:

6Z: $\{Z_{\Delta HW}, Z_{HW}\}_I$
6M: $\{M_{\Delta HW}, M_{HW}\}_I$

Δ-functionality test bench testing:

7Z: $\{Z_{\Delta SW}, Z_{SW}, Z_{\Delta HW}\}_I$
7Ma: $\{M_{\Delta SW}, M_{\Delta HW}\}_I$
7Mb: $\{M_{\Delta SW}, Z_{SW}, M_{\Delta HW}\}_I$
7Mc: $\{Z_{\Delta SW}, Z_{SW}, M_{\Delta HW}\}_I$
7Md: $\{M_{\Delta SW}, Z_{SW}, Z_{\Delta HW}\}_I$

Δ-functionality system testing:

8Z: $\{Z_{\Delta SW}, Z_{SW}, Z_{\Delta HW}, Z_{HW}\}_I$
8Ma: $\{M_{\Delta SW}, Z_{SW}, M_{\Delta HW}, M_{HW}\}_I$
8Mb: $\{Z_{\Delta SW}, Z_{SW}, M_{\Delta HW}, M_{HW}\}_I$
8Mc: $\{M_{\Delta SW}, Z_{SW}, Z_{\Delta HW}, Z_{HW}\}_I$

System testing:

9Z: $\{Z_{\Delta SW}, Z_{SW}, Z_{\Delta HW}, Z_{HW}\}_I$
9M: $\{Z_{\Delta SW}, Z_{SW}, M_{\Delta HW}, M_{HW}\}_I$

In the current integration and test process of ASML, the software qualification tests (test category 1) consume quite some machine time, approximately one full day of testing each week. Besides that machine time is limited and expensive, experience shows that also setting up the system for testing is time consuming. Moreover, much time may be lost on solving minor machine problems that are unimportant for the tests. Test time and costs may be reduced by using hardware models instead of hardware realizations for certain parts of the qualification tests. For example, the qualification of the system throughput in principle depends on the sequence and durations of all hardware actions. When the durations of these hardware actions are modeled as time delays in a model M_{HW} of the hardware system Z_{HW}, and when the software Z_{SW} executes the sequence of actions on the model M_{HW}, the system throughput can be qualified without a hardware realization Z_{HW}. In this way, the software qualification tests can be performed in a cheap desktop environment with a hardware model M_{HW}. Furthermore, models require less test setup time, and they do not suffer from the minor problems that may occur in other hardware components, since the hardware model only contains the behavior important for the tests and abstracts from these problems.

A model $M_{\Delta SW}$ of the Δ software component can be used as replacement of $Z_{\Delta SW}$ for software regression testing (test category 4), i.e., $\{M_{\Delta SW}, Z_{SW}, Z_{HW}\}_I$ (4Mb) instead of $\{Z_{\Delta SW}, Z_{SW}, Z_{HW}\}_I$ (4Z). By real-time simulation of the model in combination with the other software Z_{SW}, tests can be performed on the original hardware system Z_{HW} to check whether any of the original system functions are negatively affected by the new software component. Similar to model-based software qualification testing in test activity 1Ma, the hardware realization Z_{HW} could also be replaced by its model M_{HW}, i.e., $\{M_{\Delta SW}, Z_{SW}, M_{HW}\}_I$ (4Ma).

Testing the complete Δ-functionality using a test bench (test category 7) can be supported by four model-based test techniques. First, the Δ-functionality can be tested by using the integrated models of the Δ components, i.e., $\{M_{\Delta SW}, M_{\Delta HW}\}_I$, in which $M_{\Delta HW}$ is a model of test bench $Z_{\Delta HW}$. Since only models are used, powerful

model-based system analysis techniques like model checking can be used to exhaustively analyze all possible scenarios, as shown in step 2b of the case study in Chapter 7. Second, the model $M_{\Delta SW}$ can be integrated with the other software Z_{SW}, and tested on the test bench model $M_{\Delta HW}$. Third, the realization of the upgraded software system, i.e., $\{Z_{\Delta SW}, Z_{SW}\}_I$, can be tested on the model of the test bench $M_{\Delta HW}$. Finally, in the case that $Z_{\Delta HW}$ is available before the software realization $Z_{\Delta SW}$, the model $M_{\Delta SW}$ can be tested with Z_{SW} on the test bench realization $Z_{\Delta HW}$, as shown in step 3 of the case study in Chapter 7.

Figure 8.4 shows all test activities of the MBI&T process, in a similar way as Figure 8.2. The flag symbols and the letters indicate different milestones, for example: (d) $M_{\Delta SW}$ and $M_{\Delta HW}$ pass model tests; (g) $Z_{\Delta SW}$ passes software tests and is integrated in the QBL Z_{SW} (denoted by a downward arrow); (j) $Z_{\Delta SW}$ and $Z_{\Delta HW}$ pass test bench tests; (k) $Z_{\Delta HW}$ passes hardware integration tests and both Z_{HW} and M_{HW} are upgraded (denoted by downward arrows); (m) complete system with all Δ-functionalities passes tests and is shipped to customer.

As an example, the circles indicate test activities of test category 7, Δ-functionality test bench testing. Their positions in the process clearly illustrate how models enable earlier testing on system level when compared to the current integration and test process, in which only the realization test 7Z can be performed late in the process.

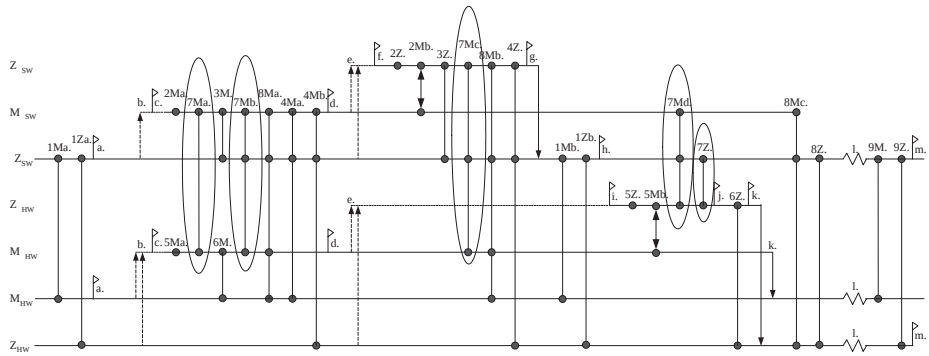


Figure 8.4: Model-based integration and testing process.

Although Figure 8.4 shows more activities than Figure 8.2, the length of these figures does not relate to the total duration of the integration and test process since the possible start times and the durations of the activities are not accounted for. For example, in the case that the hardware realization is available earlier than the software (i.e., milestone i. before f.), model-based activities 7Md and 8Mc could be performed before 2Z.

8.4 Process comparison and trade-off analysis

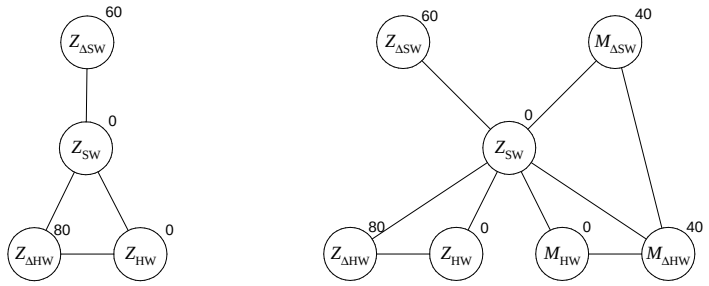
Although all possible integration and test activities have been defined in the previous section, there is still a problem that needs to be addressed before the MBI&T process can be applied to a real integration and test problem. This problem, which also exists in the current integration and test process, is called integration and test sequencing, see also Chapter 6. Integration and test sequencing involves deciding which components to integrate when, and which tests to perform in which order on which components. The goal of integration and test sequencing is to determine a sequence of integration and test activities according to certain optimization criteria like lead time, total test time, test costs, and remaining risk in the system.

For the MBI&T process, there is not only the problem of determining the optimal sequence of the integration and test activities, but there is also a choice of using models for certain integration and test activities or not. This involves a trade-off between the required modeling effort and the potential integration and test effort reduction. In some cases, it might be better to use models, e.g., when the realization of a component is available only late in the process or when testing with realizations is expensive. In other cases, it is wise not to invest in models but to perform the tests with realizations immediately, e.g., when the realization is already available, or when the realization is a mature component, i.e., the probability of finding errors is low and probably not worth the modeling effort.

In this section, we use a basic example to show how the integration and test sequencing method from Chapter 6 can be used for both the integration and test sequencing problem and for the trade-off between the effort and the potential profits of using models. For this example, we defined a fictitious but representative integration and test problem of the system upgrade scenario as used in the previous sections. The problem is instantiated once with realizations only (as in the current integration and test process), and once with the possibility to use models as well (as in the MBI&T process), so that the determined integration and test processes can be compared.

The input for the integration and test sequencing method is an *integration model* that describes the integration and test problem. Note that this integration model is different from the models M used in the MBI&T method, which describe the behavior of the components to be integrated (process vs. product model; see also Chapter 1). Figure 8.5 and Table 8.2 show the information used for the integration models. Figures 8.5(a) and 8.5(b) depict the components (circles) and interfaces (lines) for the current and for the model-based integration and test process, respectively. The numbers at the top right of each circle denote the development or delivery times of the component. This example uses typical development times for the system upgrade scenario, in which the original hardware and software are available from the start (development time is zero) and the Δ software component is available after 60 time units, which is 20 time units before the Δ hardware component. In the MBI&T process, the models of the Δ components are available after 40 time units.

Table 8.2 shows the available tests in the integration model, including the compo-



(a) Current integration and test process

(b) MBI&T process

Figure 8.5: Components and interfaces.

nents that need to be available and that must have been integrated for each test, and the test durations. The table contains all 28 test activities listed in the previous section, i.e., including both the current and the model-based test activities. Tests for which only realizations can be used (with a 'Z' in the identifier) have to be performed in both the current and the model-based integration and test process. The model-based test activities (with an 'M' in the identifier) cannot be executed in the current integration and test process since there are no models available. However, we assume that the aspects covered by a model-based test activity can also be covered by an equivalent test activity using realizations only. This choice of using models or realizations for a certain test activity is denoted by the parentheses in the second column in Table 8.2. For the current integration and test process, only the realization alternatives can be chosen, while for the MBI&T process, both alternatives can be chosen. Taking test category 1 as an example, we see that test activity 1Za requires Z_{SW} and Z_{HW} to be available and to have been integrated. Test activity 1Ma always requires Z_{SW}, but gives a choice between using the hardware system model M_{HW} or the realization Z_{HW}. For the current integration and test process, only the equivalent realization test with Z_{HW} can be used for 1Ma, while in the MBI&T process, the model M_{HW} can also be used for Ma. In this way, we can use a single set of tests to compare the current and model-based approach. The test durations in the third column of Table 8.2 are fictitious but give a representative distribution of test time over all test activities for an average system upgrade scenario.

Based on an integration model, the integration sequencing algorithm determines all feasible integration and test sequences and determines the best sequence according to the optimization criteria used. In our trade-off analysis, we used the duration of the complete integration and test process, the lead time, as the optimization criterion, since time (in particular time-to-market) is the most important business driver for ASML as explained in Chapter 3. The lead time is different from the total test time, which is

Test	Required components	Time
1Za	$\{Z_{SW}, Z_{HW}\}_I$	6
1Ma	$\{Z_{SW}, (M_{HW}/Z_{HW})\}_I$	2
1Zb	$\{Z_{\Delta SW}, Z_{SW}, Z_{HW}\}_I$	6
1Mb	$\{Z_{\Delta SW}, Z_{SW}, (M_{HW}/Z_{HW})\}_I$	2
2Z	$Z_{\Delta SW}$	1
2Ma	$(M_{\Delta SW}/Z_{\Delta SW})$	1
2Mb	$Z_{\Delta SW}$ vs. $M_{\Delta SW}$	1
3Z	$\{Z_{\Delta SW}, Z_{SW}\}_I$	2
3M	$\{(M_{\Delta SW}/Z_{\Delta SW}), Z_{SW}\}_I$	1
4Z	$\{Z_{\Delta SW}, Z_{SW}, Z_{HW}\}_I$	3
4Ma	$\{(M_{\Delta SW}/Z_{\Delta SW}), Z_{SW}, (M_{HW}/Z_{HW})\}_I$	1
4Mb	$\{(M_{\Delta SW}/Z_{\Delta SW}), Z_{SW}, Z_{HW}\}_I$	2
5Z	$Z_{\Delta HW}$	2
5Ma	$(M_{\Delta HW}/Z_{\Delta HW})$	1
5Mb	$Z_{\Delta HW}$ vs. $M_{\Delta HW}$	1
6Z	$\{Z_{\Delta HW}, Z_{HW}\}_I$	3
6M	$\{(M_{\Delta HW}/Z_{\Delta HW}), (M_{HW}/Z_{HW})\}_I$	1
7Z	$\{Z_{\Delta SW}, Z_{SW}, Z_{\Delta HW}\}_I$	4
7Ma	$\{(M_{\Delta SW}/Z_{\Delta SW}), (M_{\Delta HW}/Z_{\Delta HW})\}_I$	2
7Mb	$\{(M_{\Delta SW}/Z_{\Delta SW}), Z_{SW}, (M_{\Delta HW}/Z_{\Delta HW})\}_I$	1
7Mc	$\{Z_{\Delta SW}, Z_{SW}, (M_{\Delta HW}/Z_{\Delta HW})\}_I$	2
7Md	$\{(M_{\Delta SW}/Z_{\Delta SW}), Z_{SW}, Z_{\Delta HW}\}_I$	1
8Z	$\{Z_{\Delta SW}, Z_{SW}, Z_{\Delta HW}, Z_{HW}\}_I$	4
8Ma	$\{(M_{\Delta SW}/Z_{\Delta SW}), Z_{SW}, (M_{\Delta HW}/Z_{\Delta HW}), (M_{HW}/Z_{HW})\}_I$	2
8Mb	$\{Z_{\Delta SW}, Z_{SW}, (M_{\Delta HW}/Z_{\Delta HW}), (M_{HW}/Z_{HW})\}_I$	2
8Mc	$\{(M_{\Delta SW}/Z_{\Delta SW}), Z_{SW}, Z_{\Delta HW}, Z_{HW}\}_I$	2
9Z	$\{Z_{\Delta SW}, Z_{SW}, Z_{\Delta HW}, Z_{HW}\}_I$	60
9M	$\{Z_{\Delta SW}, Z_{SW}, (M_{\Delta HW}/Z_{\Delta HW}), (M_{HW}/Z_{HW})\}_I$	20

Table 8.2: Available tests in the integration model.

the sum of the durations of all separate test activities. By performing the test activities as much as possible in parallel, the total test time remains equal but the lead time is reduced.

Figure 8.6 shows the determined optimal sequences for the current (top) and model-based (bottom) integration and test processes, in the form of an MS Project Gantt Chart. The figure shows all development activities (dashed bars), integrations (diamonds), and test activities (solid bars) over time, and the precedences between the activities (arrows). On the first lines of the integration and test sequences, the long white bar with triangular ends indicates the lead time of the sequence.

Several conclusions can be drawn from these sequences. First, the lead time of the total MBI&T sequence is shorter, 167 time units against 188 time units for the current integration and test sequence, a reduction of 11%. Besides lead time, also the duration of the final system test phase (the long solid bars at the right hand side of Figure 8.6) is important for ASML, since this phase is on the critical path and has a major influence on the time-to-market T. The final system test phase is 78 time units for the MBI&T sequence, 25% less than the 104 time units for the current integration and test sequence.

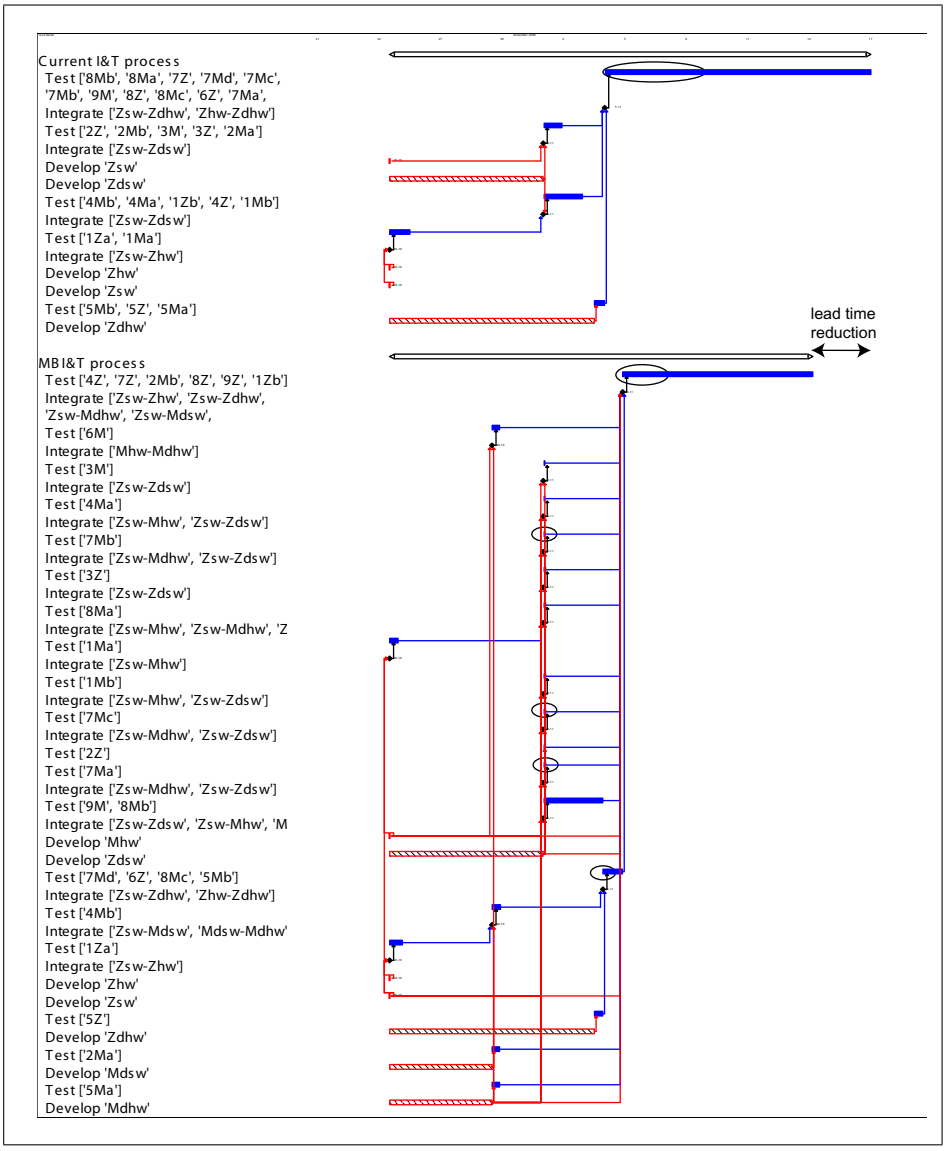


Figure 8.6: Current (top) and model-based (bottom) integration and test sequences.

Besides lead time, also the total test time and related costs can be used to compare both approaches. Looking at the total time spent on testing with realizations and models in this example, we see that the current integration and test process uses 136 time units of realization testing. The model-based integration and test process uses 92 time units of realization testing and 44 time units of testing with models. At ASML, the costs per time unit for realization testing are orders of magnitude higher than for testing software or models in a desktop environment. In this example, the reduction of test costs clearly outweighs the onetime investments needed in model development, which are 80 time units of modeling in a (cheap) desktop environment.

Finally, the integration and test sequences show that the test activities in the MBI&T process can be performed earlier and more in parallel by using models, see for example the position of the circles in both sequences, indicating when the tests of test category 7 are performed. This means that design and integration problems can be detected and prevented at an earlier and cheaper stage of development. Although this cannot directly be expressed in terms of test time or costs, the advantages of earlier testing can be explained in terms of quality and risk. By incorporating risk into the integration and test model, this can be dealt with analogous to Chapters 5 and 6.

For simplicity, the trade-off analysis example in this chapter only used a basic integration model. However, the integration model can be extended in several ways to perform a more detailed analysis. For example, by including test selection from Chapter 5, the test time and cost differences between testing with models and realizations can be incorporated in the model and in the decision making process, as well as test coverage and the risks of faults in the system. In this way, decisions like longer but cheaper model testing or shorter but more expensive realization testing can automatically be made by the sequencing algorithm. Also ‘what if’ scenarios can be investigated, for example the effects of developing more detailed models, which means higher model development times, but also a higher coverage of the model-based test activities and less test activities that can only be performed with realizations. This provides a systematic and automatic method for improving the current integration and test process by applying models at places where it is possible and profitable.

8.5 Conclusions

This chapter described an integration and test process for a system upgrade scenario that is common in industry, including nine different test categories that cover different system aspects. Since tests can only be performed with realizations, the test costs are rather high and the tests can only be performed late in the process, where fixing problems is expensive.

We applied the MBI&T method of Chapter 7 to the current integration and test process, resulting in additional model-based test activities that allow earlier, cheaper, and more parallel testing. The feasibility and potential of several of these techniques have already been demonstrated in industrial case studies such as the one of Chapter 7.

By using the integration and test sequencing technique from Chapter 6, we showed

how optimal sequences of integration and test activities can be determined and how the trade-off between the effort and potential benefits of using models for integration and testing can be analyzed. The results of a basic system upgrade example show that the lead time and costs of the current integration and test process can be reduced by performing certain tests earlier with models. Several extensions can be applied in order to perform automatic trade-off analysis for real integration and test problems.

Chapter 9

Timed model-based testing

Authors: H. Bohnenkamp, A. Belinfante

9.1 Introduction

Testing is one of the most natural, intuitive and widely used methods to check the quality of software. One of the emerging and promising techniques for test automation is *model-based testing*. In model based testing, a *model* of the desired behavior of the *implementation under test* (IUT) is the starting point for test generation. In addition, this model serves as the oracle for test result analysis. Large amounts of test cases can, in principle, be algorithmically and automatically generated from the model.

Most model-based testing methods deal with black-box testing of functionality. This implies that the kind of properties being tested concern the functionality of the system. Functionality properties express whether the system correctly does what it should do in terms of correct responses to given stimuli, as opposed to, e.g., performance, usability, or reliability properties. In black-box testing, the specification is the starting point for testing. The specification prescribes what the IUT should do, and what it should not do, in terms of the behavior observable at its external interfaces. The IUT is seen as a black box without internal detail, as opposed to white-box testing, where the internal structure of the IUT, such as the program code, is the basis for testing. In this chapter we will consider black-box software testing of functionality properties.

Model-based testing should be based on formal methods: methods which allow the precise, mathematical definition of the meaning of models, and of notions of correctness of implementations with respect to specification models. One of the formal theories for model-based testing uses labeled transition systems (LTS) with inputs and outputs as models, and a formal implementation relation called *ioco* for defining conformance between an IUT and a specification [117, 118]; see also Chapter 11. An

important ingredient of this theory is the notion of *quiescence*, i.e., the absence of output, which is considered to be observable. Quiescence provides additional information on the behavior of the IUT, and therefore allows to distinguish better between correct and faulty behavior. Moreover, the *ioco* theory defines how to derive sound test cases from the specification. The set of all *ioco*-test cases (which is usually of infinite size) is exhaustive, i.e., in theory it is possible to distinguish all faulty from all *ioco*-correct implementations by executing all test cases. In practice, *ioco*-test cases can be used to test software components and to find bugs. The testing tool TORX has been developed [10, 119] to derive *ioco*-test cases automatically from a specification, and to apply them to an IUT. TORX does *on-the-fly testing*, i.e., test case derivation and execution is done simultaneously. TORX has been used successfully in several industry-relevant case-studies [7, 9, 122]. Alternative approaches are, e.g., TGV [65], the AGEDIS TOOL SET [57], and SPEC EXPLORER [36, 84].

This chapter is about an extension of TORX to allow testing of real-time properties: *real-time testing*. Real-time testing means that the decisions whether an IUT has passed or failed a test is not only based on which outputs are observed, given a certain sequence of inputs, but also on *when* the outputs occur, given a certain sequence of inputs *applied at certain times*. Our approach is influenced by, although independent of, the *tioco* theory [26], an extension of *ioco* to real-time testing. Whereas the *tioco* theory provides a formal framework for timed testing, we describe in this chapter an algorithmic approach to real-time testing, inspired by the existing implementation of TORX. We use as input models nondeterministic *safety timed automata*, and describe the algorithms developed to derive test cases for timed testing.

Related Work. Other approaches to timed testing, based on timed automata, exist, described in particular in [74, 85]. The big difference is that we take *quiescence* into account in our approach.

TORX itself has in fact already been used for timed testing [7]. Even though the approach was an ad-hoc solution to test for some timing properties in a particular case study, the approach has shown a lot of the problems that come with practical real-time testing, and has provided solutions to many of them. This early case study has accelerated the implementation work for our TORX extensions immensely.

Structure of the Chapter. In Section 9.2, we introduce *ioco*, and describe the central algorithms of TORX. In Section 9.3, we introduce the class of models we use to describe specifications for timed testing and the adaptations to make it usable with TORX. In Section 9.4, we describe an algorithm for timed on-the-fly testing. In Section 9.5, we address practical issues regarding timed testing. We conclude with Section 9.6.

Notational Convention. We will frequently define structures by means of tuples. If we define a tuple $T = (e_1, e_2, \ldots, e_n)$, we often will use a kind of *record* notation known from programming languages in order to address the components of the tuple, i.e., we will write $T.e_i$ if we mean component e_i for T, for $i = 1, \ldots, n$.

9.2 Preliminaries

9.2.1 The *ioco* way of testing

In this section we give a summary of the *ioco* theory (*ioco* is an abbreviation for "*Input-Output-Conformance*"). Details can be found in [117, 118].

The *ioco* Theory A *labeled transition system* (LTS) is a tuple $(S, s_0, Act, \rightarrow)$, where S is a set of states, $s_0 \in S$ is the initial state, Act is a set of labels, and $\rightarrow \subseteq S \times Act \cup \{\tau\} \times S$ is the transition relation. Transitions $(s, a, s') \in \rightarrow$ are frequently written as $s \xrightarrow{a} s'$. τ is the *invisible* action. The set of all transition systems over label set Act is denoted as $\mathcal{L}(Act)$. Assume a set of input labels L_I, and a set of output labels L_U, $L_I \cap L_U = \emptyset$, $\tau \notin L_I \cup L_U$. Elements from L_I are often suffixed with a "?" and elements from L_U with an "!" to allow easier distinction. An LTS $L \in \mathcal{L}(L_I \cup L_U)$ is called an Input/Output transition system (IOTS) if L is *input-enabled*, i.e., $\forall s \in S, \forall i? \in L_I : \exists s' \in L.S : s \xrightarrow{i?} s'$. Input-enabledness ensures that IOTS can never deadlock. However, it might be possible that from certain states no outputs can be produced without prior input. This behavior is described by the notion of *quiescence*: let $L \in \mathcal{L}(L_I \cup L_U)$, and $s \in L.S$. Then s is *quiescent* (denoted $\delta(s)$), iff $\forall a \in L_U \cup \{\tau\} : \neg\exists s' \in L.S : s \xrightarrow{a} s'$. We introduce the *quiescence label*, $\delta \notin L_I \cup L_U \cup \{\tau\}$, and define the *$\delta$-closure* $\Delta(L) = (L.S, L.s_0, L_I \cup L_U \cup \{\tau\} \cup \{\delta\}, \rightarrow')$, where $\rightarrow' = L.\rightarrow \cup \{(s, \delta, s) \mid s \in L.S \wedge \delta(s)\}$.

We introduce some more notation to deal with a transition system L. For $a \in Act \cup \{\tau\}$, we write $s \xrightarrow{a}$, iff $\exists s' \in L.S : s \xrightarrow{a} s'$. We write $s \xrightarrow{a_1,\ldots,a_n} s'$ iff $\exists s_1, s_2, \ldots, s_{n-1} \in L.S : s \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \cdots s_{n-1} \xrightarrow{a_n} s'$. We write $s \Longrightarrow s'$ iff $s \xrightarrow{\tau,\ldots,\tau} s'$, and $s \overset{a}{\Longrightarrow} s'$ iff $\exists s'', s''' \in L.S : s \Longrightarrow s'' \xrightarrow{a} s''' \Longrightarrow s'$. The extension to $s \xRightarrow{a_1 \cdots a_n} s'$ is defined similarly as above.

For a state $s \in \Delta(L).S$, the set of *suspension traces* from s, denoted by $Straces(s)$, are defined as $Straces(s) = \{\sigma \in (L_I \cup L_U \cup \{\delta\})^* \mid s \overset{\sigma}{\Longrightarrow}\}$, where $\Longrightarrow$ is defined on top of $\Delta(L).\rightarrow$. We define $Straces(L) = Straces(\Delta(L).s_0)$. For $s \in \Delta(L).S$, we define $out(s) = \{o \in L_U \mid s \xrightarrow{o}\} \cup \{\delta \mid \delta(s)\}$, and, for $S' \subseteq \Delta(L).S$, $out(S') = \bigcup_{s \in S'} out(s)$. Furthermore, for $s \in \Delta(L).S$ and $\sigma \in (L_I \cup L_U \cup \{\delta\})^*$, $\underline{s \textbf{ after } \sigma} = \{s' \in \Delta(L).S \mid s \overset{\sigma}{\Longrightarrow} s'\}$, and for $S \subseteq \Delta(L).S$, $\underline{S \textbf{ after } \sigma} = \bigcup_{s \in S} \underline{s \textbf{ after } \sigma}$. We define $\underline{L \textbf{ after } \sigma} = \underline{\Delta(L).s_0 \textbf{ after } \sigma}$.

Let *Spec*, *Impl* $\in \mathcal{L}(L_I \cup L_U)$ and let *Impl* be an IOTS. Then we define

$$\textit{Impl} \textbf{ ioco } \textit{Spec} \Leftrightarrow \forall \sigma \in Straces(\textit{Spec}) : out(\underline{\textit{Impl} \textbf{ after } \sigma}) \subseteq out(\underline{\textit{Spec} \textbf{ after } \sigma}).$$

The last line basically says that an implementation is only correct with respect to *ioco* if and only if all the outputs it produces, or quiescent phases, are predicted by, and thus correct according to, the specification.

Testing for *ioco* Conformance: Test Case Derivation The most important property of the *ioco* theory is that it is possible to derive test cases from specifications *automatically*. If an IUT fulfills certain assumptions (these assumptions are commonly known as *testing hypothesis*) then the *ioco*-test cases are *sound*: failing an *ioco*-test case implies that the IUT is not *ioco*-conformant with the specification. Test cases are described as deterministic, finite, non-cyclic LTS with two special states **pass** and **fail**, which are supposed to be *terminating*. Test cases are defined in a process-algebraic notation, with the following syntax: $T ::= \textbf{pass} \mid \textbf{fail} \mid a;T \mid \sum_{i=1}^{n} a_i T_i$, for $a, a_1, \ldots, a_n \in L_I \cup L_U \cup \{\delta\}$. Assuming an LTS *Spec* $\in \mathcal{L}(L_I \cup L_U)$ as a specification, test cases are defined recursively (with finite depth) according to the following rules. Starting with the set $S = \{ s \mid \Delta(\mathit{Spec}).s_0 \Longrightarrow s \}$,

(1) $T := \textbf{pass}$ is a test case;

(2) $T := a;T'$ is a test case, where $a \in L_I$ and, assuming that $S' = \underline{S \textbf{ after } a}$ and $S' \neq \emptyset$, T' is a test case derived from set S';

(3) For $\overline{out(S)} = (L_U \cup \{\delta\}) \setminus out(S)$,

$$T := \sum_{x \in \overline{out(S)}} x;\textbf{fail} \quad + \sum_{x \in out(S)} x;T_x$$

is a test case, where the T_x for $x \in out(S)$ are test cases derived from the respective sets $S_x = \underline{S \textbf{ after } x}$.

With not too much phantasy it is possible to imagine an algorithm which is constructing test cases according to the three rules given above.

9.2.2 On-the-fly *ioco* testing: TORX

In Figure 9.1 we see the tool structure of TORX. We can distinguish four tool components (not counting the IUT): EXPLORER, PRIMER, DRIVER and ADAPTER. The

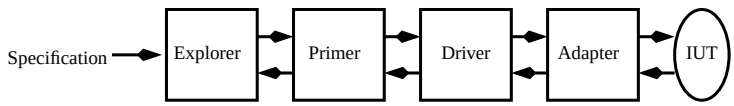


Figure 9.1: The TORX tool architecture.

EXPLORER is the software component that takes a specification as input and provides access to an LTS representation of this specification. The PRIMER is the software component that is *ioco*-specific. It implements the test case derivation algorithm for the *ioco* theory. In particular, the PRIMER interacts directly with the EXPLORER, i.e., the representation of the specification, in order to compute so-called *menus*. Menus are sets

Algorithm *Compute_Menu*

```
1   input: Set of states S
2   output: Sets of transitions in, out
3   in := ∅
4   out := ∅
5   already_explored := ∅
6   foreach s ∈ S
7       already_explored := already_explored ∪ {s}
8       S := S \ {s}
9       is_quiescent := true
10      foreach s -a-> q' ∈ Spec.→
11          if a = τ
12              is_quiescent := false
13              if q' ∉ already_explored : S := S ∪ {q'}
14          else :
15              if a ∈ L_I : in := in ∪ {s -a-> q'}
16              else :
17                  out := out ∪ {s -a-> q'}
18                  is_quiescent := false
19      end
20      if is_quiescent : out := out ∪ {s -δ-> s}
21  end
22  return(in,out)
```

Figure 9.2: Menu computation.

of transitions with input, output or δ labels, which according to the model are allowed to be applied to the IUT or allowed to be observed.

The PRIMER is triggered by the DRIVER. The DRIVER is the only active component and acts therefore as the motor of the TORX tool chain. It decides whether to apply a stimulus to the IUT, or whether to wait for an observation from the ADAPTER, and it channels information between PRIMER and ADAPTER.

The ADAPTER has several tasks: i) interface with the IUT; ii) translate abstract actions to concrete actions and apply the latter to the IUT; iii) observe the IUT and translate observations to abstract actions; iv) detect absence of an output over a certain period of time and signal quiescence.

The recursive definition of test cases as described in Section 9.2.1 allows to derive and execute test cases simultaneously, *on-the-fly*. The core algorithm is the computation of *menus* from a set of states S. The output menu contains transitions labeled with the actions from the *out*-set $out(S)$. The input menu contains all inputs that are allowed to be applied to the IUT, according to the specification. The reason to keep transitions, rather than actions, in menus is that it is necessary to know the destination states which can be reached after applying an input or observing an output. The computation of a menu requires for each state in S the bounded exploration of a part of the

Algorithm *Driver_Control_Loop*

```
1   input: —
2   output: Verdict pass or fail
3   (in, out) = Compute_Menu({s_0})
4   while ¬stop :
5     if ADAPTER.has_output() ∨ wait:
6       o = ADAPTER.output()
7       M = out after o
8       if M = ∅: terminate(fail)
9       (in, out) = Compute_Menu(M)
10    else:
11      choose i? ∈ {a | q -a-> q' ∈ in}
12      if ADAPTER.apply_input(i?) :
13        (in, out) = Compute_Menu(in after i?)
14  end
15  terminate(pass)
```

Figure 9.3: Driver Control Loop.

state space. Recursive descent into the state space is stopped when a state is seen that has no outgoing τ transitions, or that was visited before.

The algorithm for the computation of menus is given in Figure 9.2. We assume LTS $Spec \in \mathcal{L}(L_I \cup L_U)$. Input to the algorithm is a set S of states. Initially, $S = \{Spec.s_0\}$. After trace $\sigma \in (L_I \cup L_U \cup \{\delta\})^*$ has been observed, $S = \underline{Spec\ \mathbf{after}\ \sigma}$. Note that the transitions with δ labels are implicitly added to the out set when appropriate (line 20). Therefore, the EXPLORER does not have to compute the δ-closure of the LTS it represents.

Given the computed menus *in*, out, the DRIVER component decides how to proceed with the testing. The algorithm is given in Figure 9.3. In principle, the DRIVER has to choose between the three different possibilities that have been given for the *ioco* test case algorithm in Section 9.2.1: i) termination, ii) applying an input in set *in*, or iii) waiting for an output.

With the variables *wait* and *stop* we denote a probabilistic choice: whenever they are referenced, a dice is thrown and depending on the outcome either **false** or **true** is returned. The driver control loop therefore terminates with probability one, because eventually *stop* will return **true**. The choice between ii) and iii) is also done probabilistically: if the ADAPTER has no observation to offer to the DRIVER, the variable *wait* is consulted. To describe the algorithm of the DRIVER, we enhance the definition of $\underline{\cdot\ \mathbf{after}\ \cdot}$ to menus. If M is a menu, then we define $\underline{M\ \mathbf{after}\ a} = \{q' \mid (q \xrightarrow{a} q') \in M\}$.

Quiescence in Practice From the specification point-of-view, quiescence is a structural property of the LTS. In the real world, a non-quiescent implementation will produce an output after some finite amount time. If an implementation never produces an output, it is quiescent. Therefore, from an implementation point-of-view, quiescence

can be seen as a timing property, and one that can not be detected in finite time. In theory, this makes quiescence detection impossible. However, in practice it is possible to work with approximations to quiescence. A system that is supposed to work at a fast pace, like in the order of milliseconds, can certainly be considered as being quiescent, if after two days of waiting no output has appeared. Even two hours, if not two minutes of waiting might be sufficient to conclude that the system is quiescent. It seems to be plausible to approximate quiescence by waiting for a properly chosen time interval after the occurrence of the latest event. This is the approach chosen for TORX. The responsibility to detect quiescence and to send a synthetic action, the *quiescence signal*, lies with the ADAPTER.

9.3 Timed testing with timed automata

In this section we describe timed automata, the formalism which we use to formulate specifications for timed testing.

9.3.1 Timed automata

A timed automaton is similar to an LTS with some extra ingredients: apart from states, actions and transitions, there are *clocks*, *clock constraints*, and *clock resets*. Timed automata states are actually called locations, and transitions *edges* or *switches*.

Clocks are entities to measure time. They take nonnegative values from a time domain $\mathbb{T}$ (usually the nonnegative real numbers) and advance linearly as time progresses with the same rate. Let $\mathcal{C}$ be the set of clocks. Clock constraints are boolean expressions of a restricted form: an *atomic clock constraint* is an inequality of the form $b_l \prec x - y \prec b_u$ or $b_l \prec x \prec b_u$, for $x, y \in \mathcal{C}$, $\prec \in \{<, \leq\}$, and $b_l, b_u \in \mathbb{T}$ with $b_l \leq b_u$. Clock constraints are conjunctions of atomic clock constraints. The set of all clock constraints over clock set $\mathcal{C}$ is denoted by $\mathcal{B}(\mathcal{C})$. Clock constraints evaluate to either true or false. Since they depend on clock valuations, which change over time, also the evaluation of clock constraints changes generally over time. Clock constraints are used in two places in a timed automata: as *guards* and as *invariants*. Every transition has a guard, which describes the conditions (depending on the clock valuations) under which the transition is enabled, i.e., can be executed. Locations, on the other hand, are associated with an invariant. An invariant describes the conditions under which it is allowed to be in its corresponding location. Invariants describe an urgency condition: a location must be left before an invariant evaluates to false.

Clock resets are subsets of $\mathcal{C}$ and are associated to transitions. If a transition is executed, all clocks in the corresponding clock set are set to 0. As before, the action set is divided into inputs and outputs.

In Figure 9.4 we see an example for a timed automaton. This timed automaton has 7 locations, named S_0 to S_6. There is only one clock, c. The switches are named $e_0, \ldots, e_{12}$ and are labeled with actions, ending on ? or !, distinguishing inputs from outputs. Transitions without action labels are considered to be internal, i.e., labeled

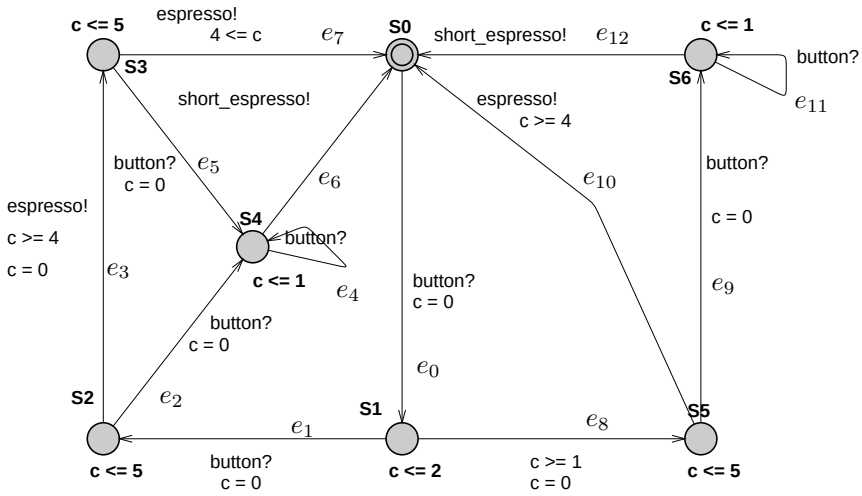


Figure 9.4: Timed Automaton.

with τ (e_8 in the example). The timed automaton describes the behavior of a coffee machine of which behavior depends on time. Starting location is S_0. Pressing a button (`button?`) brings us with e_0 to location S_1. Clock c is reset. The invariant of S_1 makes sure that within 2 seconds the location must be left again. This happens either by pressing the button again, which brings us with e_1 to location S_2. Alternatively, the internal transition e_8 can be executed: the guard enables it after 1 second of idling in S_1. Reaching location S_2 means that we have asked for two espressos. Consequently, going from S_2 to S_0 (edges e_3 and e_7) gives us two outputs `espresso!`. If we are in location S_5, we have pressed the button only once, and we can go with only one output `espresso!` back to location S_0. In locations S_2, S_3 and S_5 the invariants are always $c \leq 5$, and the transitions labeled with `espresso!` have a guard $c \geq 4$. This means that, since all transitions except those leading into S_0 reset clock c, one shot of espresso is produced within 4 and 5 seconds[1]. In locations S_2, S_3 and S_5 it is also always possible to receive another button press. This press cuts the coffee production short, i.e., via intermediate locations S_4 or S_6, we reach location S_0. The output is then, consequently, a short espresso (`short_espresso!`), which is obtained within 1 second.

A formal definition of a timed automaton follows.

Definition 1 (Timed Automaton) *A timed automaton T is a tuple $(N, \mathcal{C}, Act, l_0, E, I)$, where N is a finite set of locations, $\mathcal{C}$ is a set of clock variables, Act is a set of labels (partitioned into L_I and L_U as before), $l_0 \in N$ is the initial location, $E \subseteq N \times (Act \cup \{\tau\}) \times \mathcal{B}(\mathcal{C}) \times 2^{\mathcal{C}} \times N$ is the set of edges (or switches),*

[1] Actually a good espresso needs a bit longer than that.

and $I : N \to \mathcal{B}(\mathcal{C})$ assigns invariants to locations. We define $\mathcal{A}(Act)$ to be the set of timed automata over the label set Act.

If $e = (l, a, g, r, l') \in E$, we also write $l \xrightarrow{a,g,r} l'$, where a is the action label, g is the guard, and r the clock reset.

9.3.2 Quiescence

Using a timeout to approximate quiescence has immediate impact on an approach to timed testing. Whereas in the un-timed case quiescence detection via time-out can not be described in the theory itself, in timed testing it should and actually must be described: a timeout is a timing property which influences therefore a timed test run. Incorporating the quiescence timeout into the timed testing technique is, as it turns out, a problem with a straightforward solution, which we will describe below. However, there is one assumption that must be made on the behavior of implementations.

Definition 2 *For an implementation Impl there is an $M \in \mathbb{T}$ such that*

- *Impl produces an output within M time units, counted from the last input or output, or,*
- *if it does not, then Impl will never ever produce an output again (without prior input).*

Only if this assumption on a real implementation holds, our test approach will work. This assumption is thus part of the above mentioned testing hypothesis.

A timed automaton to be used as a specification must be modified in order to express when quiescence is allowed to be accepted. To do that, it is necessary to know what M to assume. Then,

(1) an extra clock QC is added to the timed automaton;

(2) a self-loop labeled with special action δ is added to each location. Its guard is $\text{QC} \geq M$;

(3) clock QC is added to the clock reset of every transition labeled with an input or output;

(4) the guard of every output transition is extended with $\text{QC} < M$.

If $\mathcal{A}$ is a timed automaton, we denote this modified timed automaton as $\Delta_M(\mathcal{A})$.

9.3.3 From timed automata to zone LTS

TorX assumes that a specification is modeled in terms of labeled transition systems. In order to use TorX for timed testing, it is thus necessary to derive an LTS representation from a timed automaton. Such an LTS will be called a *zone LTS*. The technical details are not of interest here and can be found in [15]. Important to know, however, are the following facts. We assume a timed automaton $\Delta_M(\mathcal{A}) = (N, \mathcal{C}, Act \cup \{\delta\}, l_0, E, I)$.

(1) Time is measured in absolute time counted from system start (i.e., from the time when the initial state of the LTS was initially entered).

(2) States of the underlying LTS are of the form (l, z), where $l \in N$, and z is a so-called clock zone. The whole tuple is called a zone; z describes information about time. In particular, it defines an interval $z^{\downarrow} = [t_1, t_2] \subseteq \mathbb{T}$ which describes that only for absolute time $t \in [t_1, t_2]$ the state (l, z) might be entered.

(3) If we have a transition $(l, z) \xrightarrow{a} (l', z')$, we also write $(l, z) \xrightarrow{a@[t_1,t_2]} (l', z')$, if $z'^{\downarrow} = [t_1, t_2]$.

(4) For every transition $(l, z) \xrightarrow{a} (l', z')$ there is a corresponding edge $e = l \xrightarrow{a,g,r} l' \in E$.

(5) Given state (l, z), the successor clock zone of z for edge $e = l \xrightarrow{a,g,r} l'$ is denoted $z' = Succ(z, e)$. The successor state of (l, z) for e is then consequently (l', z'). It can happen that $Succ(z, e)^{\downarrow} = \emptyset$, which indicates that the switch e can not be executed, i.e., (l', z') is then not a successor state of (l, z).

(6) Zones can be instantiated. If $(l, z) \xrightarrow{a@[t_1,t_2]} (l', z')$ (with corresponding edge $e \in E$), and $t \in [t_1, t_2]$, then we can derive a new successor state (l', z'') of (l, z) such that $z''^{\downarrow} = [t, t]$. We denote this instantiated successor clock zone as $z'' = Succ(z, e, t)$.

The first transition of the zone LTS of Figure 9.4 corresponds to switch e_0 and has the form $(S_0, z^0) \xrightarrow{\texttt{button?}@[0,\infty]} (S_1, z^1)$ for intial clock zone z^0 and $z^1 = Succ(z^0, e_0)$. Since e_0 has no guard, the button can be pressed any time, i.e., between absolute time 0 and ∞. If `button?` is pressed at time t, then we derive $z^1_t = Succ(z^0, e_0, t)$. Then we can derive from edge e_1 the transition $(S_1, z^1_t) \xrightarrow{\texttt{button?}@[t+0,t+2]} (S_2, z^2)$ with $z^2 = Succ(z^1_t, e_1)$. From e_8, we can derive a transition $(S_1, z^1_t) \xrightarrow{\tau@[t+1,t+2]} (S_5, z^5)$, and with e_{10} also $(S_5, z^5) \xrightarrow{\texttt{espresso!}@[t+5,t+7]} (S_0, z^6)$, where $z^5 = Succ(z^1_t, e_8)$ and $z^6 = Succ(z^5, e_{10})$. This derivation shows that if the button is pressed once at, say, time 10, then without further interference we can expect an espresso between time 15 and 17.

9.4 Timed automata testing with TORX

In the following we describe the algorithms implemented in TORX for timed testing. We assume a timed automaton *Spec* $\in \mathcal{A}(L_I \cup L_U)$ and consider its δ-closure $\Delta_M(\mathit{Spec})$ for an appropriately chosen value M. Similarly to the un-timed case, TORX computes input- and output-menus, and chooses between applying an admissible input and waiting for an output.

Algorithm *Compute_Menu_TA*

1 **input**: Set of zones S
2 **output**: Set of zone automata transitions *in*, out
3 *in* $:= \emptyset$
4 $out := \emptyset$
5 *already_explored* $:= \emptyset$
6 **foreach** $(l, z) \in S$
7 *already_explored* $:=$ *already_explored* $\cup \{(l, z)\}$
8 $S := S \setminus \{(l, z)\}$
9 **foreach** $e \in \{e' \in E \mid e'.l = l\}$
10 **if** $z' = Succ(z, e) \wedge z'^{\downarrow} \neq \emptyset$:
11 **if** $e.a = \tau$: $S := S \cup \{(e.l', z')\}$
12 **else** :
13 **if** $e.a \in L_I$: *in* $:=$ *in* $\cup \{(l, z) \xrightarrow{a} (e.l', z')\}$
14 **else** : $out := out \cup \{(l, z) \xrightarrow{a} (e.l', z')\}$
15 **end**
16 **end**
17 **return**(*in*,out)

Table 9.1: Computation of menus from timed automata.

9.4.1 Menu computation

Based on the zone-LTS described above, TORX computes menus. The algorithm is similar to the one in Figure 9.2, but is specialized to account for the admitted time intervals of a zone. This algorithm ***Compute_Menu_TA*** is given in Table 9.1. The input of the algorithm is a set of zones S (line 1). The output comprises two sets, the *in* menu and the out menu. (lines 3, 4, 17). The set *already_explored* is used to keep track of zones already explored (line 5). We have an outer loop over all states (i.e., zones (l, z)) in the set S (lines 6–16). The contents of S varies during the computation. All states considered inside the loop are added to *already_explored* and removed from S (lines 7, 8). The inner loop (line 9 – 15) considers every switch e with source location l. First, the successor clock zone z' of z according to switch e is computed (line 10). If $z'^{\downarrow}$ is not empty, transitions of the zone LTS are added to the sets *in* or out, depending on the labels of switch e (lines 11–14). Note that transitions with label δ are added to the out menu, i.e., the δ action is not treated any different from an output. In case of a τ label, the resulting zone is added to set S (line 11). In essence, the menu computation is a bounded state space exploration of the zone LTS with sorting of the generated transitions according to their labels.

Algorithm *Driver_Control_Loop_TA*

1 **input**: —
2 **output**: Verdict **pass** or **fail**
3 $(in, out) =$ ***Compute_Menu_TA***$(\{(l_0, \{x = 0 \mid x \in \mathcal{C}\})\})$
4 **while** $\neg$ *stop*:
5 **if** ADAPTER.*has_output*$() \vee$ *wait*:
6 $o@t :=$ ADAPTER.*output*$()$
7 **if** $out \ \underline{\mathbf{after_t}} \ o@t = \emptyset$: *terminate*(**fail**)
8 $(in, out) :=$ ***Compute_Menu_TA***$(out \ \underline{\mathbf{after_t}} \ o@t, t)$
9 **else**:
10 choose $i@t \in \{a@t' \mid (l, z) \xrightarrow{a} (l, z') \in in \wedge t' \in z'^{\downarrow}\}$
11 **if** ADAPTER.*apply_input*$(i@t)$:
12 $(in, out) =$ ***Compute_Menu_TA***$(in \ \underline{\mathbf{after_t}} \ i@t, t)$
13 **end**
14 *terminate*(**pass**)

Table 9.2: DRIVER control loop for timed systems.

9.4.2 Driver control loop

We define the following operator $\cdot \ \underline{\mathbf{after_t}} \ \cdot$, which maps menus on sets of zones.

Definition 3 ($\cdot \ \underline{\mathbf{after_t}} \ \cdot$) *Let $\Delta_M(\mathcal{A}) = (N, \mathcal{C}, Act \cup \{\delta\}, l_0, E, I)$ be a timed automaton, and let M be a menu. Then, for $a \in Act$ and $t \in \mathbb{T}$,*

$$\begin{aligned} M \ \underline{\mathbf{after_t}} \ a@t \quad = \quad & \{(l', z'') \mid (l, z) \xrightarrow{a} (l', z') \in M \\ & \text{and } z'' = Succ(z, e, t) \text{ with } z''^{\downarrow} \neq \emptyset\}, \end{aligned} \tag{9.1}$$

where e is the respective switch corresponding to the $(l, z) \xrightarrow{a} (l', z')$ transition.

If the set M is a menu computed by ***Compute_Menu_TA***, each transition $(l, z) \xrightarrow{a} (l', z')$ contains the interval of all times at which a is allowed to happen: the interval $z'^{\downarrow}$. The set $M \ \underline{\mathbf{after_t}} \ a@t$ then computes a set of successor zones from M which can be reached by executing a at exactly time t.

In Table 9.2, we see the algorithm for the DRIVER control loop of TORX, enhanced to deal with time. Menus are computed with ***Compute_Menu_TA***, and the successor states are computed with $\cdot \ \underline{\mathbf{after_t}} \ \cdot$. When an input is applied, not only an input $i?$ is chosen, but also a time instance $t \in z'^{\downarrow}$ (line 10), at which time to apply the input. The variables *wait* and *stop* have the same meaning as in the *ioco* algorithm (*cf.* Section 9.2.2).

9.5 Timed testing in practice

9.5.1 Notes on the testing hypothesis

The *testing hypothesis* is an important ingredient in the testing theory of Tretmans [117]. The hypothesis is that the IUT can be modeled by means of the model class which forms the basis of the testing theory. In case of *ioco* the assumption is that the IUT can be modeled as an input-enabled IOTS. Under this assumption, the results on soundness and completeness of *ioco*-testing do apply to the practical testing approach. In this chapter, we have not defined a formalism that we consider as model for an implementation, so we can not really speak of a *testing hypothesis*. Still, it is important to give some hints on what properties a real IUT should have in order to make timed testing feasible. We mention four points.

First, we require input enabledness, as for the un-timed case. That means, whenever it is decided to apply an input to the IUT, it is accepted, regardless of whether this input really does cause a non-trivial state change of the IUT or not.

Second, it is plausible to postulate that all time measurements are done relative to the same clock that the IUT refers to. In practice this means that the TORX ADAPTER should run on the same host as the IUT and reference the same hardware clock. If measurements would be done by different clocks, measurement errors caused by clock skew and drifts might spoil the measurement, and thus the test run.

Third, as has been pointed out in Section 9.3.2, it is assumed that the implementation behaves such that quiescence can be detected according to Section 9.3.2, Definition 2. This an assumption, part of the test hypothesis, which the system designer may have to ensure.

Fourth, up to now we left open which time domain $\mathbb{T}$ to choose for our approach. The standard time domain used for timed automata are real numbers, however, in practice only floating point numbers, rather than real numbers can be used. Early experiments have shown that floats and doubles quite quickly cause numerical problems. Comparisons of time stamps turn out to be to inexact due to rounding and truncation errors. In the TORX implementation we use thus fixed precision numbers, i.e., 64 bit integers, counting micro-seconds. This happens to be the time representation used for the UNIX operating system family.

9.5.2 Limitations of timed testing

Even though the timed testing approach described in this chapter seems to be easy enough, timed testing is not easy at all. Time is a complicated natural phenomenon. It can not be stopped. It can not be created artificially in a lab environment. Time runs forward, it runs everywhere, and, leaving Einstein aside, everywhere at the same pace. For timed testing this means that there is no time to waste. The testing apparatus, TORX, in this case, must not influence the outcome of the testing approach. However, the execution of TORX does consume time, and the question is when the execution time of TORX does influence the testing.

- Assume that input $i?$ is allowed to be applied at time $0 \leq t \leq b$. Assume that the testing tool needs $b/2$ to prepare to apply the input. Then the input can never be applied between time 0 and $b/2$. If there is an error hiding in this time interval, it will not be detected.
- Assume that the tester is too slow to apply $i?$ before b. Then this input can not be applied, and some behavior of the IUT might never be exercised.

This basically means that the speed of the testing tool and the speed of communication between tester and IUT determine the maximal speed of the IUT that can be reliably tested.

Springintveld et al. [112] define an algorithm to derive test cases for testing timed automata. They prove that their approach to test timed automata is possible and even complete, but in practice infeasible, due to the enormous number of test cases to be run. This is likely also the case for our approach and thus limits the extend to which timed testing can be useful. Automatic selection of meaningful test cases might be an important ingredient in future extensions of our approach. For the time being, our goal is to find out how far we can get with timed testing *as is* in practice. This will be subject of our further research.

9.6 Conclusions

In this chapter we have presented Timed TORX, a tool for on-the-fly real-time testing. We use nondeterministic safety timed automata as input formalism to describe system specifications, and we demonstrate how to use standard algorithms for zone computations in order to make our approach work. It turns out that the existing TORX algorithms, especially in the PRIMER and DRIVER can in principle be reused in order to deal with time.

The TORX implementation is still in a prototype stage. Yet, small systems of the size of the coffee machine in Figure 9.4 can be tested.

Our approach is strongly related to the *tioco* testing theory [26]. In fact, it has been shown (although not published yet) that the testing technique described here is in fact a sound and exhaustive instance of the *tioco* theory.

Chapter 10

Model-based testing of hybrid systems

Author: M.P.W.J. van Osch

10.1 Introduction

The goal of test automation is to reduce the test effort and to increase the quality of a system. With test automation tests can be performed faster and they can be endlessly repeated without much additional effort. The test engineer can focus on testing the parts of the system for which tests are not automated.

In model-based conformance testing, tests are automatically generated from a specification and executed on the system under test (SUT). The output behavior of the SUT is observed. If the observed output was allowed according to the specification testing may continue or stop with the verdict pass. If the observed output was not allowed according to the specification the test stops with the verdict fail. The advantages of model-based testing are that models can be reused to test every product in exactly the same way. Therefore, more tests can be generated and performed. Also, tests can be generated (e.g., by generating random tests) that a test engineer did not think of.

The first input output-conformance (ioco) theory for discrete event systems was developed by Tretmans [117]; see also Chapters 9 and 11. There are several academic tools that implement the ioco theory such as TorX [119, 8] and TGV [49]. With this theory and these tools it is possible to test whether a certain discrete output event takes place, given a certain input event, e.g., it is possible to test a laser controller by stimulating it with a "GO TO EXPOSE"-message and then observing whether it produces the correct sequence of messages for the laser to start exposure. In recent years, time has been added to model-based conformance testing [15, 74, 26, 75] as explained in

Chapter 9. This made it possible to formally test whether certain discrete events occur in time.

A hybrid system is a system that exhibits both discrete-event and continuous behavior. For example, a thermostat that observes a chamber temperature and turns on a heater based on the observed temperature (change) is a system with continuous input and discrete output. A robot arm that moves is a system with discrete input (e.g., a message "go left") and continuous output (e.g., the movement of the arm with a certain speed).

In the Tangram project we have developed a theory [94] and a proof of concept tool for model-based conformance testing of hybrid systems. Some of the issues involved were how to select the input from the specification for the test, how to connect the test tool to the SUT, and how to sample.

We have tried out our theory and tool in two toy example case studies and a case study in industry. The first toy example is a thermostat simulator that turns a heater on or off and observes the temperature of a tank through a sensor. The second toy example is a robot arm simulator that can accelerate or brake by increasing or decreasing its speed. For both cases we tested four different simulated implementations against one specification model. In each case one of the simulated implementations was correct, and three others were erroneous mutants. In both case studies we were able to detect the mistakes in the mutants. The industrial case concerned a vacuum control system. We were able to connect our test tool to this controller. The test tool applies in real-time (sampled) continuous input to the controller, and the tool automatically observes and validates output actions. The results are promising but we were not yet able to test the system extensively and find errors. We continue working on this case study.

The only hybrid model-based test tool available at the moment, to our knowledge, is the Charon tester [116, 115]. This is a prototype tool implemented in the Charon framework [1] with no underlying formal test theory. The Charon tester takes a different approach than our tool or the other test tools mentioned above. Charon is a tool set for hybrid simulation and verification that is capable of doing runtime verification. In runtime verification a program is generated from a property that is executed together with the implementation. If the property is violated, this is reported. With the Charon tester, besides properties, also an environment (of the implementation) is modeled. A test generator has been implemented that is also executed together with the implementation and that selects input for the implementation from the environment model. If a property is violated, then the test fails. The advantage of this approach is that inaccuracy in observations is not an issue, e.g., there is no clock skew because both test and implementation run on the same platform. The disadvantage of this approach compared to ours is that it is not flexible. For every different environment, different test generator, and different execution platform, new code needs to be written and generated, and embedded in the system under test.

In this chapter we give an overview how we dealt with the issues involved in automated model-based testing of hybrid systems, and our achievements so far. For further details of our hybrid input-output conformance theory we refer to [94]. In Section 10.4

we informally describe how a hybrid system is tested. In Section 10.3 we give an overview of our proof of concept test tool implementation and its current shortcomings. In Section 10.4 we present the results of our case studies. In Section 10.5 we discuss the possibilities for future work in this area.

10.2 Testing hybrid systems

In this section we first give an overview of model-based testing in general, then we present how tests are generated for hybrid systems in our approach, and finally we give an example of how a thermostat with continuous input and discrete output actions is tested in our approach.

Figure 10.1 illustrates the process of model-based testing. The specification is a model of the SUT containing input to be applied to the SUT, and output that can be observed from the SUT. The SUT is considered to be a black box. We do not know the internal behavior of the system. We assume that the SUT is input complete, that is, we assume that we can apply any input to an implementation and we can always observe something, namely output or the absence of output (during a certain time interval).

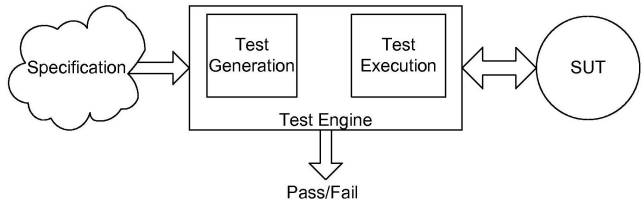


Figure 10.1: Conformance Testing.

Tests are generated from a (formal) specification that models the behavior of the SUT. It can restrict the input we want to use in our test. The specification does not need to be input complete, i.e., the specification does not need to specify every possible input. This allows us to guide the test with respect to the input applied to the SUT. If a certain input is not specified, the SUT is not tested for this particular input, for instance, we can test only the good weather behavior or only the bad weather behavior of the SUT.

For every potential output the generated test describes if observing this output means that the test should continue, lead to the verdict pass, or lead to the verdict fail. Because in general an SUT can non-deterministically produce a number of outputs, a test should be thought of as a tree. For every possible output in a certain state of the SUT, the test describes the next possible input or output. This means that a test can get very large. Therefore, in practice, test generation and execution take place in an *on-the-fly* manner. On the fly test generation means that a test is generated one step at the time. After a step (apply an input action or observe output) has been generated,

it is executed on the SUT. Depending on the output observed, the next step is generated and performed. The advantage of on-the-fly test generation is that for only one output the next step is generated. The disadvantage of on-the-fly test generation is that it takes additional computation time during test execution, which makes testing of real time systems harder.

For hybrid systems the specification consists of both continuous behavior (e.g., described by differential equations on variables) and discrete actions. This specification can, e.g., be fully made in a hybrid modeling language, or consist of a combination of a discrete model and a continuous model. The continuous behavior can also be derived from, e.g., the sensor logs from an already operational system.

An on-the-fly test for a hybrid system consists of steps of the following types:

- select and apply one discrete-event input from the set of possible input events according to the specification;
- observe an output event from the SUT, and if the observed output event was not allowed according to the specification, then the test stops with a verdict fail; if an output event should have been observed but it was not observed, then the verdict fail is concluded as well; or
- select and apply continuous input for a specific duration and because the continuous output takes place simultaneously, observe continuous output until the end of the selected continuous input is reached or an output event is observed, and if the observed (discrete-event and continuous) output was not allowed according to the specification, then the test stops with a verdict fail.

If the verdict fail is not concluded, the test continues with another step or stops with a verdict pass.

Figure 10.2 shows a specification of a thermostat. The behavior of the thermostat is as follows. In this specification it is assumed that $MinT < MinON < MinOFF < MaxT$. Initially the chamber temperature T is $MinT \leq T \leq MaxT$ and the thermostat is in the **Not Heating** mode. The thermostat observes the temperature T of a chamber (continuous input) and switches a heater on or off (by an output action). The thermostat can turn a heater on if the temperature is below the specified temperature $MinON$. The thermostat in this case switches to the **Heater ON** mode in which it observes the chamber (environment) being heated with a rate between $0\ ^oC/min$ and $1\ ^oC/min$. After the temperature $MinOFF$ is reached the thermostat returns to the **Not Heating** mode with the message !HeaterOFF and the temperature in the chamber starts to decrease (e.g., because the chamber is not perfectly isolated and placed in a colder environment). If the temperature T of the chamber increases too fast (with more than $1\ ^oC/min$) the thermostat switches to the **Over Heating** mode. Before the maximum temperature $MaxT$ is reached the heater thermostat switches the **OFF** mode and produces a message !Error. After an error the thermostat can only be reset by a discrete input action ?Reset (e.g., a button being pressed by an operator).

Note that this specification is not input complete because if it reaches **Over Heating** mode the temperature will always rise with more than $1\ ^oC/min$ until it switches to the **OFF** mode. Therefore, tests generated from this specification will not contain the behavior that the temperature increases with more than $1\ ^oC/min$ for a while and then (again) increases with less than $1\ ^oC/min$. If we want to test whether an SUT behaves correctly in this case, we first need to adapt the specification.

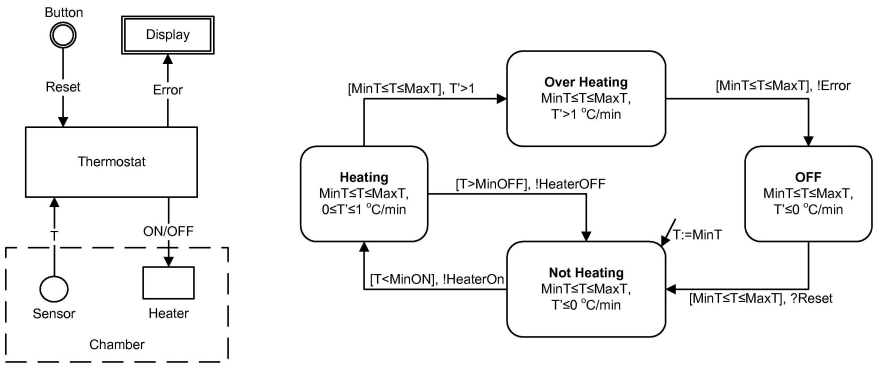


Figure 10.2: Thermostat Example.

Let for instance $MinT = 5\ ^oC$, let $MinOn = 10\ ^oC$, let $MinOff = 15\ ^oC$, and let $MaxT = 20\ ^oC$. Let the initial temperature be $T = MinOn$. Then, the following sequence of steps describes a scenario to generate and execute a test on-the-fly for the thermostat.

step 1: Decrease the temperature with $0.5\ ^oC/min$ for one minute. If a !heaterON output or no output is observed, then continue testing with step 2, otherwise (if, e.g., a !heaterOFF output or an error output is observed which is not allowed according to the specification), stop testing and conclude with verdict fail.

The SUT can either have produced no output, the !heaterON output, the !heaterOFF output, or the !Error output. Suppose that the SUT produced the !heaterON output after 30 seconds, then immediately after the output is observed and validated a new step can be generated.

step 2: Select a temperature increase with $T' = 0.1 * T\ ^oC/min$ for two minutes. Note that in this differential equation T' depends on T. If T increases T' increases as well.

At the start of step 3 the temperature is $9.75\ ^oC$. Therefore, at the start of the step $T' < 1\ ^oC/min$. However, as soon as $T = 10\ ^oC$ the thermostat goes to **Over Heating** mode. The test can continue as follows.

step 3: Select a temperature increase $T' > 1\ ^oC/min$ until $20\ ^oC$ is reached.If the output action Error is observed the test can continue with a new step or stop with the verdict pass, otherwise (if no output !error is observed, or an output action !HeaterON or !HeaterOFF is observed) stop testing with the verdict fail.

Suppose that the output !Error was observed from the SUT. Then it is possible to continue the test by applying a decreasing temperature at most until the temperature reaches $5\ ^oC$ and applying an input action ?Reset. It is also possible to stop testing at this point with verdict pass, because in this case the test did not fail.

This example does not illustrate all kinds of hybrid systems that can be tested according to our theory and with our tool. There are more kinds of hybrid systems that can be tested. The thermostat only has continuous input, input actions and output actions. It is also possible to test hybrid systems with continuous output. In this case we compare also the continuous output of an SUT with the specified continuous output. An example of such system is a robot arm which moves according to a certain speed. It is also possible to test SUTs with both continuous input and continuous output, and with discrete input and discrete output. An example of such system is a brake control system of a car that brakes (which is continuous output) in accordance with the pressure applied on the brake pedals (which is continuous input), and that brakes if it gets a message from the cruise control (which is an input action) and turns on a warning light (which is an output action). It is also possible to test systems with multiple continuous input flows and continuous output flows. An example of such system is a vacuum controller which besides the pressure in a chamber also takes into account the temperature of the chamber for its output.

Our test theory and tool do not provide the means of testing the kind of hybrid systems in which the continuous input instantaneously depends on its own continuous output. It is more complicated to test this kind of systems because in this case tests need to be generated that are continuously adapted to the continuous output observed from the SUT.

10.3 Test tool implementation

In this section we describe the main issues involved in implementing a test tool based on the theory described in the previous section.

We preferred not to build our test tool from scratch but reuse (libraries of) already existing hybrid simulation tools or verification tools. We chose the hybrid language χ (Chi) [5] for our specification language and we reused the libraries of the χ simulator [114] to build a prototype test tool. The tool has been implemented in Python [104].

The architecture of our test tool (see Figure 10.3) is the same basic architecture as the (timed) test tools TorX [8], TTG [73], and TRON [120].

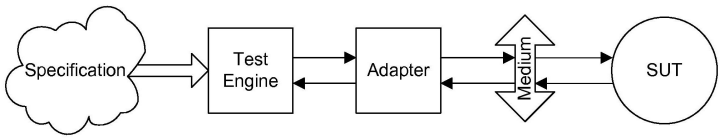


Figure 10.3: Test Architecture.

The test engine generates tests. It implements the on-the-fly test generation and execution procedure, with sampled continuous behavior. It steps through the specification and computes the sets of possible input (actions and sampled continuous flow) and observable output (actions and sampled continuous flow). It selects input, it validates the observed output and gives a verdict. The adapter transforms input to a format that is suitable to be sent over the communication medium to the SUT and it transforms output received over the communication medium from the SUT so that it can be compared with the specified output. The medium can be for instance a computer network or electronic wires.

10.3.1 The test engine

The test engine of our prototype tool performs a number of tasks. These tasks are:

- compute the input to be applied to the SUT and the output expected to be observed from the SUT
- select input;
- sample continuous behavior;
- compare the observed output with the specified output;
- decide when to stop testing; and
- return an error trace if the test failed.

In this section we discuss these tasks.

In the χ tool set a send or receive action in a channel communication consists of a channel, and a value or variable. A trajectory consists of a set of variables with their flow specified as ordinary differential equations (ODEs) and a duration. The χ tool set implements methods to compute the set of allowed transitions from a state, and given a state and a transition, to compute the state of the specification after taking this transition. With these methods we compute (given a set of states) the sets of input actions, output actions, and trajectories, and we compute the set of reachable states after these (input or output) actions, trajectories, or internal actions. Because in an SUT operating in a real environment time cannot stop, the specification does not contain

infinite sequences of actions or loops of actions (otherwise time can stop according to the semantics of χ). Because of this and because the continuous behavior is specified by ODEs, the set of trajectories is also finite.

Input can be selected either manually (by the user) or automatically by the test tool. For hybrid input-output conformance testing, manual input selection is only possible if the SUT does not run in real-time. In this case a user can select which continuous input or discrete input action to apply to the SUT. If an input action is selected, the user can select an input value. If continuous input is selected (e.g., specified by a differential equation for every input variable), the user can select the duration of the trajectory.

Automatic input selection has to be done according to some selection criteria. Currently, we have only implemented random input selection with some restrictions on the input domain. An input action and an input value are selected, or a trajectory and a duration are selected, at random. For input actions, in principle the selection domain is the type of the action (for instance boolean values, e.g., heater on is true or false). This domain can be infinite, and generate many tests that are not of interest to the user. Therefore, we made it possible to specify a set of values, from which the test tool selects the input. Some test selection mechanisms we are considering to implement in the future are test selection in which the user specifies some order in the input (e.g., after an "ON" signal, always an "OFF" signal is applied) and more intelligent test selection mechanisms (e.g., by using coverage criteria or test purposes).

With the proper interface to the SUT we potentially can apply and observe continuous input and output in the form of electronic signals. It is also possible to apply samples of continuous input and observe samples of continuous output from the SUT instead of real continuous behavior. This still has advantages over discrete event testing (in which samples of continuous behavior are modelled) because the sample rate can easily be adjusted without changing the model and already existing continuous models can be reused.

In our prototype implementation we apply and observe samples of continuous behavior instead of real continuous behavior (e.g., signals). In this way it is still possible to test for instance a vacuum controller on the software level. A (software) controller that is tested on the application level of the system architecture already observes samples of continuous input or samples the observations. In this way we keep away from the electronic/hardware domain and the need for, e.g., designated hardware.

The sample rate is chosen by the user of the test tool. It has to be chosen such that there is still enough time to compute new input and evaluate output between samples and such that the sample rate of the controller is maintained. In our prototype implementation, Maple [80] is used to solve the differential equations and compute the valuation of input variables and output variables at each sample point.

There are three problems with using samples of continuous behavior. The first problem is inaccuracy in applying and observing samples due to latency in the communication between the test tool and the SUT. A possible solution is to send the input before it has to be applied together with the time it has to be applied and implement a test environment around the SUT that applies the input on time. For output observa-

tions time stamps could be added in the test environment on the SUT side. However, this creates a problem if after sending the next input, but before applying it, a delayed output is observed. The second problem is inaccuracy in applying and observing samples due to clock skew. It can be the case that the system clock of the SUT runs slightly faster or slower then the system clock of the execution platform of the test tool. This creates the problem that an observed output is timed correctly according to the system clock of the test tool, but was incorrectly with respect to the system clock of the SUT or vice versa. The third problem is inaccuracy in applying and observing samples due to rounding off. It can happen that an output is incorrect according to the specification but because of rounding off by the test tool is validated as being correct. For now the only way to handle these inaccuracies is by making specifications which take these potential inaccuracies into account. However, a more constructive solution for these problems is desirable.

After an input sample for each input variable has been sent to the adapter, the test engine waits for the output samples to be received and compares the observed output with the output allowed according to the specification. The observed output actions and samples are received together with a time stamp. If the observed output is contained in the set of possible output actions and valuations, the test continues, otherwise the test stops with the verdict fail.

As long as the verdict fail is not given, the test can end with a verdict pass. After that, it is possible to generate a new test that tests other behavior of the system and may lead to the verdict fail. A test ends with verdict pass when some stop criterion is met. The stop criterion could, e.g., be the user of the tool pressing a "stop" button, a time limit, a limit to the number of inputs (actions or trajectories) applied and outputs observed, or a coverage criterion on the specification. Our prototype tool currently implements the first two stop criteria.

Together with a verdict a test tool also returns a trace consisting of actions and trajectories performed by the specification and observed from the implementation. Because in practice only samples of input are applied and observed, in our current prototype a trace of samples and actions is returned instead of specified trajectories.

10.3.2 The adapter

The adapter transforms input specified in the χ model to a format suitable for the SUT and it transforms output from the SUT to the format used in the χ model. The adapter deals with converting variable names, channel names, and data to interfaces specific to the SUT and the other way around, e.g., a discrete message specified as an "ON" or "OFF" signal may be a boolean variable in the SUT.

The adapter also implements the interface with the communication medium, for instance, when the medium is a TCP/IP network the adapter implements the communication protocol for that medium. For our prototype we assume that our communication medium is reliable. No messages or samples may get delayed or lost between the adapter and the SUT. If the communication medium is not reliable we need a way to

make it reliable through some communication protocol, or take this the unreliability into account when validating the received output.

Furthermore, the adapter deals with timing of input (actions and samples). In our current prototype implementation we assume the medium is part of the SUT. That is, input samples or input actions are sent over the medium at the specified time, and output samples or actions are considered to have occurred at the time they are received by the adapter.

10.4 Case studies

With our prototype test tool we did some case studies. First we tested a simulator of a temperature controller and a simulator of a moving robot arm. The behavior of the thermostat was similar to the thermostat described in section The goal of the these case studies was to validate our prototype implementation. The simulators did not run in real-time. That is, they waited for a sample of input or an input event. After they received an input they could produce an output event or sample. Then they waited for a new sample of input or an input event again. This significantly simplified connecting the test-tool to the simulators.

Secondly, we tried to test the real-time vacuum control system of an ASML wafer stepper. The goal of this case study was to see whether hybrid model based testing can be used in industry.

10.4.1 Testing a thermostat and a robot arm

The thermostat simulator was a discrete event χ model that received samples of temperature as input. It produced a heater-on event if a minimum temperature $MinT$ was reached. It produced a heater-off event if a maximum temperature $MaxT$ was reached. The robot arm simulator was a discrete χ model that received clock ticks and accelerate, stop, or brake messages as input. It produced change in speed as output. Initially the robot arm was not moving.

For both simulators a correct (equivalent) hybrid χ specification was made. Besides specifying the discrete actions of both simulators, the thermostat specification contained a differential equation for temperature change. The robot arm contained a differential equation for time ($T' = 1$) and a differential equation for the expected speed changes if accelerating, braking, and in standstill.

Then, for both simulators three mutants were created. For the thermostat these mutants were:

mutant 1: the output action "ON" was changed to "OFF",

mutant 2: the minimum temperature $MinT$ on which the thermostat was supposed to switch on the heater was raised, and

mutant 3: the minimum temperature $MinT$ was lowered.

For the robot these mutants were:

mutant 1: the simulated speed behavior was changed,

mutant 2: initially the robot arm starts moving, and

mutant 3: after receiving a brake message, the robot arm continues moving.

Testing a correct SUT of the temperature controller against the hybrid specification of the temperature controller did not lead to finding any mistake. In mutant 1 of the temperature controller the incorrect output action was detected. The test tool received the output "ON" event when it expected the output "OFF" event. In mutant 2 an unexpected output action was detected. The test tool received the output "ON" on a temperature that was higher then the specified $MinT$ and no output action was expected yet. In mutant 3 an output action was expected at temperature $MinT$ but it was not observed.

Testing a correct SUT of the robot arm against the hybrid specification of the robot arm did not lead to finding any mistake. Testing the mutants led to finding the following mistakes. In mutant 1 of the robot acceleration went faster than expected. The test tool observed a speed of $5.5\ m/sec$ when it expected a speed of $3.6\ m/sec$ after 2 time units. In mutant 2 the robot accelerated unexpectedly. The test tool observed a speed of $1.9m/sec$ after two time units when it expected a speed of $0m/sec$. In mutant 3 the robot accelerated when braking was expected. The test tool observed that after first accelerating correctly to the speed of $10m/sec$ and applying a "BRK" input action, that the SUT accelerated again.

These case studies have shown that we were able to generate tests from a hybrid specification and apply and observe input and output samples from an SUT. Testing the mutant SUTs showed that the tests generated by our tool lead to a verdict pass when we expected them to pass, and lead to a verdict fail when we expected them to fail. Besides a verdict the test tool also gave us the correct reason of failure, e.g., it returns that it observed an "OFF" message when it expected an "ON" message. This also gives insight in where the mistake in the SUT was made.

10.4.2 The vacuum system case study

At the time of writing we are using the test tool to test the controller of a vacuum system (Figure 10.4). This vacuum system is used in a wafer stepper machine of ASML. The Vacuum Controller observes the pressure within the Vacuum Chamber through a set of sensors. Each sensor is capable of measuring the pressure for a specific pressure range. The software controls a network of Valves, Gauges and Pumps. Depending on the desired pressure within the machine, valves are opened or closed (to vent air into the chamber) and pumps are turned on or turned off (to pump air out of the chamber). The controller needs to perform a specific sequence of discrete actions in order to increase the pressure in the chamber (by venting) or to decrease the pressure in the chamber (by pumping).

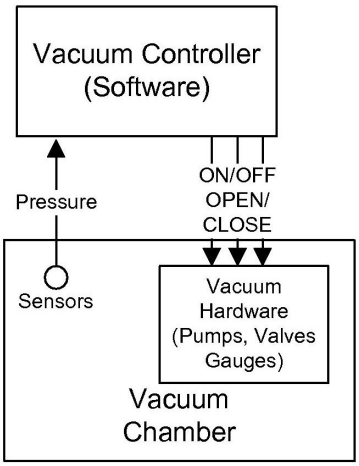


Figure 10.4: The Vacuum Controller.

In order to test the vacuum control software we have made a hybrid χ specification containing the discrete event pump down sequences and venting sequences. These are the observable output actions of the controller. The vacuum controller normally observes pressure flow from the chamber. This pressure flow is now also part of the specification. The specification contains the pressure flow that is the continuous input of the test.

The vacuum control software has been implemented in Labview. Inside the machine this control software is connected via a hardware module to the sensors. In order to connect our tool with the controller we had to implement an adapter. We have made a connection with the Labview software via TCP/IP. We have also made a mapping of the modeled pressure variables to the variables in the controller used for storing the sensor input. The specified output actions for controlling the pumps, valves, and gauges are mapped to the corresponding events of the controller.

We were able to stimulate the controller with pressure samples that were generated from the specification in real time. Input application was done in real time, with respect to the sample rate. We were able to start pump down and venting sequences. We were able to observe messages to turn pumps on or turn them off, and we were able to observe messages to open valves or to close valves. For these output messages we were able to give a verdict with respect to the behavior defined by our specification.

We observed small inaccuracies (in the order of milliseconds) in the application of input samples. The output messages were validated with respect to time stamps that we added to the output message at the moment the output occurred.

We are currently refining the specification in order to generate complicated tests for the vacuum controller and possibly find mistakes and unexpected behavior of the controller.

10.5 Conclusions and future work

We have developed a prototype test tool for hybrid systems based on our hybrid input output conformance theory. The test tool uses the language χ as specification language. Tests are generated on-the-fly, during test execution. With an adapter we are able to connect our test tool to a system under test (SUT). A part of this adapter can be reused for other SUTs. This is the part that implements the connection between the test-engine and SUT. Another part of this adapter needs to be implemented every time a different SUT is tested. This part of the adapter takes care of transforming the specified input to applicable input and transforming observed output to specified output. The latter needs to be done in order to validate the observed output. The tool applies both discrete input and continuous input, and it observes both discrete output and continuous output. In our tool, it is the case that continuous behavior is applied and observed as samples. We still consider this a form of hybrid testing because we are still testing a hybrid system against a hybrid specification.

We were able to test our example temperature controller and robot, and found the mistakes that we introduced ourselves in mutant implementations. We were also able to stimulate (in real time) an industrial vacuum control system with (sampled) pressure flow, generated from a hybrid χ model, and we were able to observe and evaluate the discrete output behavior of the controller. Unfortunately, due to performance issues we have not yet been able to do extensive test runs. We will continue developing our prototype tester and we will continue with our case studies.

The χ language does not allow to differentiate between input variables and output variables. The solution we chose for this problem is to specify this distinction separately. Another solution is to extend the language χ so that this distinction can be made in the specification.

The next step is to define two variations on our hybrid theory that better fit with how hybrid testing is implemented in practice. In our original theory we assumed that the SUT has real continuous behavior. In practice, the test tool applies samples and observes samples of continuous trajectories. We want to define a new hybrid conformance relation that defines whether a sampled SUT is conforming to a hybrid specification. In our theory we consider an SUT conforming to a specification when the output of the SUT was also allowed according to the specification. However, observing exactly that behavior is impossible because of the rounding off of valuations, delays, and clock skew. This problem can be solved by introducing a margin of inaccuracy. We also want to define a new hybrid conformance relation that allows small deviations from the specified trajectories.

Chapter 11

Test-based modeling

Author: T.A.C. Willemse

11.1 Introduction

Today's systems engineering is predominantly evolutionary in nature: the bulk of newly developed systems consist of minor and major modifications of existing systems. Apart from the addition of new features, these modifications should lead to improvements of the stability and the quality of the system. An important method that helps to assure that this is indeed the case is *regression testing*.

Regression testing aims at determining whether modifications that have been made to a system have no adverse effect on those parts of the systems that should not have been affected by the changes. Currently, testing is mostly a manual and labor intensive process, often deprived of effective automation, leading to high costs and sometimes mediocre product quality. Insights indicate that the testing effort typically consumes up to 50% of the total budget that is spent on developing a system [14], with regression testing consuming a large amount [56, 77] of the total budget.

Model-based testing (MBT) is a mathematically sound analysis technique that is used to assess the level of quality of a system. The key idea is to use mathematical models of a system to automatically generate and execute tests; see Chapters 9 and 10. Proponents of such techniques are quick to point out the benefits of this approach: the models are easier to understand and maintain, amenable to verification techniques such as *simulation* and *model-checking*, cf. Chapter 7, and are less prone to complex changes, while the automation that is potentially achieved goes well beyond the mere automatic execution of manually crafted test cases.

While model-based testing has been shown to work well on real-life systems, it comes with its own set of limitations. For instance, a major obstacle in applying MBT is rooted in the necessity to have mathematical models of a system to start with. In

practice, these required models are often unavailable. Obtaining the models *a posteriori* from informal documentation and by conducting interviews, et cetera, is either too time consuming, or even impossible (e.g., for third-party systems or legacy systems). As a result, model-based testing is left without its engine.

The techniques we outline in this chapter (collectively called *test-based modeling* techniques) are a step in the direction of applying model-based testing tools, and indeed also other formal tools such as simulations, to systems for which it is currently hard to obtain models. Our approach leans on the ideas from machine learning, such as initiated by Angluin [2], but, instead of computing an *exact* model representing the running system (i.e., actual implementations and their behaviors as observed from their external interfaces), it relies on experiments to obtain *partial* models from a running system. The partiality of our method turns out to be a powerful tool in making machine learning techniques tractable in practice. Note that while the models that have been obtained by learning are useless for testing the same system again (when the learning is done properly, all tests should result in the verdict *pass*), they are valuable for other purposes, such as for regression testing or for testing different configurations of the same system.

This chapter describes an approximation-based basic algorithm for constructing a model of a system. For this, it relies on counter-examples that are found by **ioco**-based model-based testing using the model that is under construction. Together with the algorithm, we discuss heuristics that guide the learning effort and reduce the run-time complexity of the basic algorithm. Underlying the algorithm is a representation of the constructed model using a subclass of *suspension automata* [117], called *valid* suspension automata [124], which provides a canonical, deterministic representation of a specification. Moreover, we have tested the hypothesis that a constructed model can effectively be used for regression testing and for testing of different configurations of the same system. This is demonstrated by running a prototype implementation of our algorithm on the *conference protocol*. The conference protocol is a well-known, mutant-based, bench-marking problem for testing (see e.g., [10]). The overall effectiveness of our approach is attested by the fact that 85% of all mutants of the correct system can be detected. Although the idea of using models that have been extracted from an implementation for regression testing purposes is not new (see e.g., [62]), to our knowledge, ours is the first study that actually quantifies the effectiveness of such an approach by means of mutant testing.

This chapter is organized as follows. In Section 11.2, the testing theory **ioco** is outlined; see also Chapter 9. In Section 11.3, we introduce our *test-based modeling* algorithm. Section 11.4 describes three heuristics to make the algorithm of Section 11.3 tractable in practice. Section 11.5 demonstrates the techniques using a case study. We conclude our contribution in Section 11.6.

Related work Berg *et al* [12] are among the few to have studied the effectiveness and applicability of Angluin's learning algorithm, and an optimization thereof. The studied systems are generally small (up to 100 states). The authors conclude that the per-

formance of Angluin's algorithm on prefix-closed automata comes close to its worst-case complexity, which they find disappointing, since reactive systems can usually be modeled using prefix-closed automata. Performance-wise, they remark that Angluin's algorithm has long execution times and a huge memory consumption. In [12], the information that is needed as input for Angluin's algorithm is extracted from formal models; this contrasts our experiments, which are conducted on real implementations from which we learn on-the-fly. Note that this also explains our long run-times when compared to [12].

Hungar *et al* [61, 62] and Margaria *et al* [81] also build their work around Angluin's learning algorithm. Several domain-specific optimizations over this basic algorithm are discussed. The optimizations are fueled by expert (human) knowledge. Such knowledge involves information concerning the symmetry of components and the independence of actions, and techniques to reduce the number of redundant membership queries that are generated by Angluin's algorithm. While these techniques are developed within the framework of testing of Finite-State Machines (FSM), they seem complementary to the techniques we describe in this chapter, and it is very likely that these can be combined in some form. A follow-up study on our methods is needed to substantiate this.

Peled *et al* [98] advocate a different approach, combining model checking, testing and automata learning. Logical properties, given by domain experts, are checked against a model. Counter-examples are subsequently checked against the actual system and may lead to improvements of the model or to documented faults. FSM-based conformance testing is used when no counter-examples are found; the test outcome can again lead to a modified model. The downside of this approach is that it relies on FSM-based conformance testing theory, which makes very strict assumptions and poses requirements on the implementation and specification which are difficult to meet in practice. Related to this approach is the tool VeriSoft [53] by Godefroid, which can be used to verify *concurrent systems*. VeriSoft usually requires that all components of the concurrent system that is verified are deterministic. For the verification, it relies on *bounded model-checking techniques*, rather than testing techniques.

As noted, most methods use an FSM-based testing theory. This testing theory relies on the assumption that the implementation behaves deterministically and has a finite number of states; our assumptions with respect to the system are more liberal, i.e., our techniques can also deal with non-deterministic systems with infinite state spaces. Furthermore, our techniques do not require human intellect to drive the exploration technique, in contrast to the approaches using model checking techniques, which require interesting properties to be given by a human user.

11.2 Formal testing theory

The testing theory used in this chapter is based on *refusal testing* for Labeled Transition Systems, which is also used in Chapters 9 and 10. We briefly introduce the basic ingredients and the conformance relation **ioco** [117]; most terminology and notation is

taken from [117]. The amount of formalization is kept at a bare minimum for understanding and presenting the test-based modeling concepts. In **ioco**-based testing, the fundamental notion is that of a transition system, which is akin to a non-deterministic state diagram, in which there is always exactly one label (an action representing an event or activity) on the edges connecting states.

Definition 11.2.1. A labeled transition system (LTS) is a four-tuple $\langle S, s_0, \text{Act}, \rightarrow \rangle$, where S is a nonempty set of states, $s_0 \in S$ is the initial state, and, Act is a finite set of observable actions. The relation $\rightarrow \subseteq S \times \text{Act} \times S$ is the transition relation, and for elements of the transition relation we write $s \xrightarrow{a} s'$ rather than $(s, a, s') \in \rightarrow$. We use the name of the LTS and its initial state interchangeably.

We do not consider *unobservable* events, the reasons being brevity and the fact that such events are not vital for understanding the main concepts explained in this chapter; in general, the penalty for including such events is poor readability.

The set of all LTSs over actions Act is henceforth denoted $\mathcal{L}(\text{Act})$. In practice, there is a distinction between actions that are *inputs* to a system, and actions that are *outputs* of a system. We denote the set of all LTSs with inputs Act_I and outputs Act_U by $\mathcal{L}(\text{Act}_I, \text{Act}_U)$. When referring to both inputs and outputs collectively, we still use the set Act, which in this case stands for the set $\text{Act}_I \cup \text{Act}_U$. Note that we assume that the sets of inputs and outputs are disjoint.

For the remainder of this section, let $L = \langle S, s_0, \text{Act}, \rightarrow \rangle \in \mathcal{L}(\text{Act}_I, \text{Act}_U)$ be an arbitrary LTS with inputs Act_I and outputs Act_U. A system, modeled by an LTS with inputs and outputs typically models a reactive system; such a system can interact with an environment. The typical, smallest observables of such a system are the actions. Moreover, apart from input and output actions, we also assume that a user can observe the *absence* of output. Those states in the LTS that admit an observation of absence of output — henceforth referred to as *quiescent* states — are augmented with extra δ-labeled self-loops; the observation itself is referred to as *quiescence*. Formally, we have $s \xrightarrow{\delta} s$ iff for all outputs $a \in \text{Act}_U$, $s \not\xrightarrow{a}$. Note that the label δ is assumed not to be part of the set of actions of any LTS; as a shorthand, we write Act_δ rather than $\text{Act} \cup \{\delta\}$. The *behaviors* of an LTS are then sequences consisting of actions and quiescence, i.e., elements from the set Act_δ^*, the set of finite words over the alphabet Act_δ; the transition relation $\rightarrow$ of an LTS then automatically induces a new relation $\Rightarrow$ between states and behaviors, which is defined as the least relation that is obtained by following the following rules:

(1) For all states $s \in S$, the empty behavior ϵ does not change state: $s \overset{\epsilon}{\Longrightarrow} s$,

(2) If by a behavior σ, we can move from state s to state s' (i.e., $s \overset{\sigma}{\Longrightarrow} s'$ for some $\sigma \in \text{Act}_\delta^*$), and via an action $a \in \text{Act}_\delta$, we can move from state s' to state s'' (i.e., $s' \xrightarrow{a} s''$), then by behavior $\sigma\ a$, we can move directly from state s to state s'', (i.e., $s \overset{\sigma\ a}{\Longrightarrow} s''$).

We write $s \overset{\sigma}{\Longrightarrow}$ when there is a state s' that can be reached upon execution of the behavior σ. The *size* of a behavior σ, denoted $|\sigma|$ is defined as the *length* of σ. For

brevity, we use the following shorthands throughout this chapter:

(1) The set of behaviors starting in a state s is denoted *s-traces*(s), where:

s-traces$(s) \stackrel{\text{def}}{=} \{\sigma \in \text{Act}_\delta^* \mid s \stackrel{\sigma}{\Longrightarrow} \}$,

(2) The set containing all behaviors of size less than n that start in a state s is denoted *s-traces*$_n(s)$, where:

s-traces$_n(s) \stackrel{\text{def}}{=} \{\ \sigma \in$ *s-traces*$(s) \mid |\sigma| < n\}$,

(3) The set of behaviors consisting of actions only is denoted *traces*(s), where:

traces$(s) \stackrel{\text{def}}{=} \text{Act}^* \cap$ *s-traces*(s),

(4) The set of outputs (or quiescence) that can be observed from a set of states S' is denoted *out*(S'), where:

out$(S') \stackrel{\text{def}}{=} \bigcup_{s \in S'} \{x \in \text{Act}_U \cup \{\delta\} \mid s \stackrel{x}{\longrightarrow}\}$

(5) The set of states reachable from a state s is denoted *der*(s), where:

der$(s) \stackrel{\text{def}}{=} \{s' \mid \exists \sigma \in \text{Act}^*.\ s \stackrel{\sigma}{\Longrightarrow} s'\}$,

(6) Given a set of states S', then the set of states that are reachable from one or more states in S', after performing a behavior σ is given by S' *after* σ, where:

S' *after* $\sigma \stackrel{\text{def}}{=} \{s' \mid \exists s \in S'.\ s \stackrel{\sigma}{\Longrightarrow} s'\}$.

When S' consists of a single state s only, we write s *after* σ rather than $\{s\}$ *after* σ.

A crucial feature of Labeled Transition Systems is the possibility to specify *non-deterministic* systems, i.e., systems for which the effect of executing an action in a state is not pre-determined. This means that after executing some behavior σ in a state s, the system can be in any of the reachable states s *after* σ. In contrast, a system is *deterministic* if none of its behaviors can reach more than one state.

Conformance testing. *Conformance testing* is the act of assessing whether an implementation of a system does what is prescribed by a specification of the system. We focus on a conformance relation for dynamic behaviors, called **ioco**. The conformance relation formalizes the relation between implementations and specifications. Instead of real implementations (which are black-box, non-formal objects), it assumes there is *some* labeled transition system that exactly describes its behaviors; the **ioco** conformance relation is therefore a relation between a *model of an implementation* and a specification. Only a subset of LTSs is assumed to represent real implementations, viz. *input-enabled* LTSs. An LTS L is *input-enabled* when it accepts *all* inputs in *all* states it can reach. If the LTS $L \in \mathcal{L}(\text{Act}_I, \text{Act}_U)$ is input-enabled, we call L an *input/output transition system* (IOTS), and the set of all IOTSs over inputs Act_I and outputs Act_U is denoted $\mathcal{IO}(\text{Act}_I, \text{Act}_U)$.

The main idea behind the conformance relation **ioco** is as follows. Given an arbitrary behavior σ of the specification, and suppose the implementation somehow is able to mimic this behavior. In order for the implementation to *conform to* the specification it may subsequently produce only outputs that are predicted by the specification. Note that we do not demand that all predicted outputs *are* (at some time) produced. Formalising this intuition, we obtain the following definition:

Definition 11.2.2. Let $L \in \mathcal{L}(\text{Act}_I, \text{Act}_U)$ be a specification, and $I \in \mathcal{IO}(\text{Act}_I, \text{Act}_U)$ an implementation. I is a **ioco**-correct implementation of L, denoted I **ioco** L, when:

$$\forall \sigma \in \textit{s-traces}(L).\ \textit{out}(I\ \textit{after}\ \sigma) \subseteq \textit{out}(L\ \textit{after}\ \sigma) \tag{11.1}$$

Example 11.2.3. As an example of the **ioco**-implementation relation, we consider the two input-output transition systems $L_1, L_2 \in \mathcal{IO}(\{m?\}, \{c!, t!\})$, depicted in Figure 11.1. Both model a coffee-vending machine, where $m?$ represents the insertion of money, $c!$ represents coffee and $t!$ represents tea. We find that L_1 **ioco** L_2 (i.e., an implementation may be more selective in its output, so, even after the behavior $m?\ \delta\ m?$, the implementation is allowed to produce only $c!$, whereas the specification permits output $t!$ as well). L_2 **ioco** L_1, on the other hand, does not hold (i.e., an implementation may not produce unpredictable outputs: the output $t!$ is possible in L_2 after executing behavior $m?\ \delta\ m?\ t!$, but it is not possible in L_1 after executing the same behavior). Note that by removing transition $t!$ from L_1, L_1 **ioco** L_2 no longer holds, because we then introduce a possibility to observe quiescence in L_1, which is not allowed by L_2: the sequence $m?\ m?$ would give rise to quiescence in L_1, but L_2 would give rise to the outputs $t!$ or $c!$.

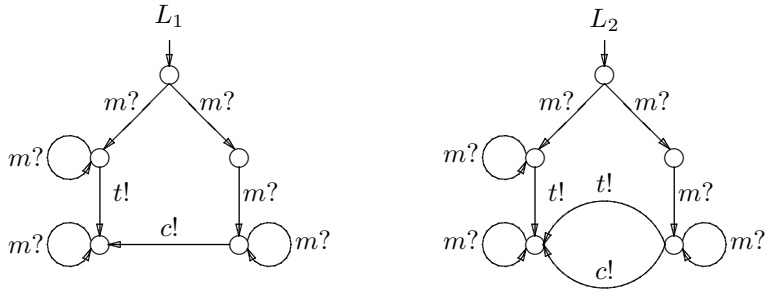


Figure 11.1: Two input-output transitions systems.

Since the model for the implementation I is not necessarily known, proving that I **ioco** L holds is usually not feasible, which is why *tests* are often derived from L that can be executed on the running implementation I to obtain confidence that I **ioco** L holds (or not). Tretmans [117] provides a detailed study of **ioco**, and also gives a sound and complete test case derivation algorithm for testing for **ioco** conformance. This algorithm underlies the tool TorX [10]. Note that the completeness result says that a test can be

derived to detect any non-conformance. It does not state that by running such a test once, the non-conformance *will* be detected.

Testing for **ioco**-correctness is in practice not exhaustive, since e.g., infinite behaviors of a system (if present) are never tested, due to the finite nature of the testing activity. We weaken general **ioco** to n-bounded **ioco** which makes the finiteness in depth explicit.

Definition 11.2.4. Let $L \in \mathcal{L}(\text{Act}_I, \text{Act}_U)$ be a specification and let $I \in \mathcal{IO}(\text{Act}_I, \text{Act}_U)$ be an implementation. Let $n \in \mathbb{N}$ be an arbitrary natural number. We say that I is an *n-bounded* **ioco**-correct implementation, denoted I n-**ioco** L, when:

$$\forall \sigma \in \textit{s-traces}_n(L).\ \textit{out}(I\ \textit{after}\ \sigma) \subseteq \textit{out}(L\ \textit{after}\ \sigma) \tag{11.2}$$

n-Bounded **ioco**-correctness guarantees that all behaviors of the implementation which are of length smaller than n are followed by an observation that is permitted by the specification. Behaviors of length n or larger are therefore ignored.

11.3 Test-based modeling

In practice, most systems (e.g., legacy systems and third party components), do not come with an adequate formal specification, which means that model-based testing techniques cannot be applied out-of-the-box. Theoretically, this problem could be solved by employing *automata learning* techniques, such as Angluin's learning algorithm [2], to obtain these models. However, to be industrially applicable as a technique, a practical learning algorithm should be able to deal with systems that have very large state spaces, usually even infinite ones (which prohibits the use of *Finite* State Machine-based techniques); Angluin's algorithm currently seems to be unfit for such systems [12].

Most research focuses on optimizing Angluin's algorithm. We take a different approach, one that is orthogonal to the representation problem that is solved by Angluin's algorithm. In this section, we outline our test-based modeling algorithm, which can be used to obtain a *partial (in depth) model* from a system. The algorithm relies on **ioco**-based test techniques. In Section 11.4, we discuss three heuristics that make the algorithm described in this section applicable for industrially sized systems, and that give rise to partiality in the 'width' of the model.

Representing models: valid suspension automata The non-deterministic behavior of a system is a major source of complexity when learning its model by experimenting. A straightforward determinization of the learnt model is in general impossible without compromising **ioco** conformance. We therefore recall the definition of *suspension automata* (SA) [117]. Suspension automata are deterministic LTSs with explicit inputs, outputs and a quiescence label δ, which is considered to be an output, and no internal actions. We denote the set of all SAs over inputs Act_I and outputs Act_U

by $\mathcal{L}_\delta(\text{Act}_I, \text{Act}_U)$. Tretmans [117] describes a transformation $\Delta{:}\mathcal{L}(\text{Act}_I, \text{Act}_U) \rightarrow \mathcal{L}_\delta(\text{Act}_I, \text{Act}_U)$ that converts an arbitrary LTS with inputs and outputs to a suspension automaton. It satisfies the following property:

Theorem 11.3.1 (*Tretmans [117]*). Let $L \in \mathcal{L}(\text{Act}_I, \text{Act}_U)$ be a specification, and let $\Delta(L)$ be its SA, with initial state s_δ. Then, for all implementations I:

$$I \textbf{ ioco } L \text{ iff } \forall \sigma \in \mathit{traces}(s_\delta).\ \mathit{out}(I \textit{ after } \sigma) \subseteq \mathit{out}(s_\delta \textit{ after } \sigma) \qquad (11.3)$$

The implications are that we can use the suspension automaton obtained from a specification when testing for **ioco**-conformance instead of the specification itself. We write I **ioco** M for a suspension automaton M when I is **ioco**-conform to a specification represented by M. There is, however, a large class of suspension automata that do not correspond to specifications given by LTSs, as illustrated below. This means that care should be taken that any test-based modeling algorithm or heuristic stays within the fragment of SAs for which an LTS specification exists.

Example 11.3.2. Consider suspension automaton M_1 from Figure 11.2. M_1 models an 'anomalous' system: it can produce an output *after* an observation of quiescence (in state p_1), and, it has a state (viz. state p_0) in which the system is neither quiescent, nor does it produce output. Next, consider suspension automaton M_2 (Figure 11.3).

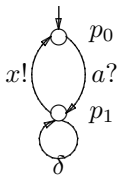


Figure 11.2: SA M_1.

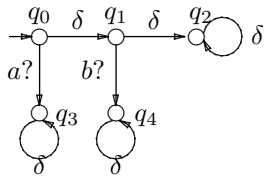


Figure 11.3: SA M_2.

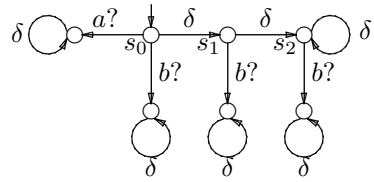


Figure 11.4: SA M_3.

The trace $\delta b?$ is a valid trace in M_2, the trace $b?$ is not. Hence, here the observation of quiescence *adds* new possibilities, which is impossible in SAs derived from LTSs. Further, M_2 is 'instable' after the observation of quiescence: M_2 allows for a $b?$ after one observation of δ, but not after two observations of δ. M_3 on the other hand, does represent the behavior of an SA that could have been the result from transforming an LTS to an SA. An LTS that would correspond (**ioco**-wise) to M_3 requires a silent transition to move from the initial state (in which both $a?$ and $b?$ are possible) to a state in which only $b?$ is possible (see also [124] for a transformation from a subclass of SAs to LTSs). □

In [124], it is shown that when an SA $M = \langle S, s_\delta, \text{Act}_\delta, \rightarrow \rangle$ satisfies the following four requirements (it is then called *valid*), it exactly corresponds to an LTS specification for **ioco**:

(1) M should be *non-blocking*, i.e., for any reachable state in M, there is at least one enabled output (this can also be *quiescence*),

(2) M should be *quiescent reducible*, i.e., any behavior σ that can follow an observation δ, should also immediately be possible, i.e., without first observing δ,

(3) M should be *anomaly-free*, i.e., in none of the behaviors of M, a non-quiescent output (i.e., an action $x \in \text{Act}_U$ different from δ) may immediately follow a quiescent output (i.e., δ)

(4) M should be *stable*, i.e., the behaviors *after* observing quiescence are not changed by observing quiescence repeatedly. For instance: if $s \xrightarrow{\delta} s' \xrightarrow{\delta} s''$, then it must be true that $\textit{traces}(s') = \textit{traces}(s'')$.

The transformation function Δ of [117] always yields *valid* SAs (see [124]). This means that validity is a requirement that is respected by all suspension automata that can be derived by translating LTSs.

Learning hypothesis and oracles **ioco**-Based testing is rooted in several assumptions, collectively known as the *testing hypothesis*, the most important assumption being that implementations can be modeled using input/output transition systems. These assumptions make testing practically applicable. We strengthen the testing hypothesis with the following assumption, leading to the *learning hypothesis*:

> all output actions (and quiescence) that can follow an experiment (sequence of inputs and outputs or quiescence) can, and will be observed by conducting the same experiment a finite (*a priori* known) number of times.

Note that the learning hypothesis quantifies the fairness of the resolution of a non-deterministic choice in a system, without attaching real values to this resolution. The learning hypothesis provides us with a powerful oracle: the system-under-test itself.

Algorithm Let I be an (unknown model of an) implementation of a system. Algorithm 1 (hereafter referred to as the TBM-algorithm) automatically constructs a suspension automaton $\mathcal{H}$, such that I N-**ioco** $\mathcal{H}$ holds upon termination of the algorithm. By 'closing' it using the technique described in [124], it even becomes a *valid* SA such that I **ioco** $\mathcal{H}$ holds. This closing, however, has no real practical significance, but is only there to guarantee general correctness for the **ioco** theory and demonstrates that validity can be achieved; practical implementations of the TBM algorithm can safely omit this step, which is why it is omitted in the current exposition.

The TBM-algorithm computes a tree-like hypothesis (with δ-loops) that is such that I is at least N-bounded **ioco** correct w.r.t. $\mathcal{H}$. The *learning phase* (lines 3–8) is the most crucial part of the algorithm. In this iteration, the hypothesis $\mathcal{H}$ is tested for $n+1$-bounded **ioco**-correctness, and, possibly modified to cope with counterexamples (lines 5-6). These counterexamples (line 4) are obtained by means of standard model-based testing techniques; in particular, we use the **ioco**-theory and test derivation to test the hypothesis $\mathcal{H}$ against the real implementation. The testing can be done in e.g., an on-the-fly manner which is implemented in [10]. The *extension phase* (line 9),

Algorithm 1 Basic TBM-algorithm.

Pre: Implementation I with inputs Act_I and outputs Act_U and depth $N \in \mathbb{N}$

Post: Suspension Automaton $\mathcal{H} = \langle S, s_\epsilon, \text{Act}_\delta, T \rangle$, where:

- $S = \{s_\sigma \mid \sigma \in \Sigma\}$, with $\Sigma = \text{Act}_\delta^* \setminus \text{Act}_\delta^* \; \delta\delta \; \text{Act}_\delta^*$
- T is computed by the algorithm.

1: $n, T := 0, \emptyset$;
2: **while** $n \neq N$ **do**
3: **while** $\neg(I \; (n{+}1)$-**ioco** $s_\epsilon)$ **do**
4: **choose** counterexample $\sigma \; x \in \Sigma$ for '$I \; (n{+}1)$-**ioco** s_ϵ' with $|\sigma| = n$;
5: **if** $x = \delta$ **then** add transitions $s_\sigma \xrightarrow{\delta} s_{\sigma\delta}$ and $s_{\sigma\delta} \xrightarrow{\delta} s_{\sigma\delta}$ to T;
6: **else** add transition $s_\sigma \xrightarrow{x} s_{\sigma x}$ to T;
7: **end if**
8: **end while**
9: for all $a \in \text{Act}_I$, $s_\rho \in \mathit{der}(s_\epsilon)$ with $|\rho| = n$, add $s_\rho \xrightarrow{a} s_{\rho a}$ to T;
10: $n := n + 1$;
11: **end while**

extends the $n{+}1$-bounded **ioco**-correct hypothesis at each node at depth n with new input transitions.

The size of the state-space of the hypothesis that is learnt by the TBM-algorithm is bound from below by $|\text{Act}_\delta \setminus \text{Act}_U|^N$ and from above by $|\text{Act}_\delta|^N$. The number of experiments (tests) that are needed is also bound from below by $M \cdot |\text{Act}_\delta \setminus \text{Act}_U|^N$. Note that M is the maximal number of times an experiment must be repeated to observe all outputs that might follow, which, by virtue of the learning hypothesis, exists. A complicating factor in practice is that conducting a single experiment of length J, consisting of U outputs ($U \leq J$), is only guaranteed to succeed in M^J tries, due to the non-determinism of the implementation. Statistics may be used to find out the expected number of experiments, based on the actual observed frequencies of outputs following an experiment, but we leave this as a topic for future research.

Example 11.3.3. Applying the TBM-algorithm with $N \geq 2$ on IOTS L_2 of Figure 11.1, we obtain the hypothesis of Figure 11.5 after variable n of the algorithm has been incremented to 2 (line `10`). The depicted hypothesis is constructed as follows: initially, the hypothesis consists of the state s_ϵ only, and the only experiment preventing I 1-**ioco** s_ϵ is an observation of quiescence; the hypothesis is extended with a δ-transition and a δ-loop to state s_δ. The transition $m?$ to state $s_{m?}$ is subsequently added in line `9`. In the next iteration, the two experiments violating I 2-**ioco** s_ϵ are $m?t!$ and $m?\delta$, so the hypothesis is extended accordingly, et cetera. □

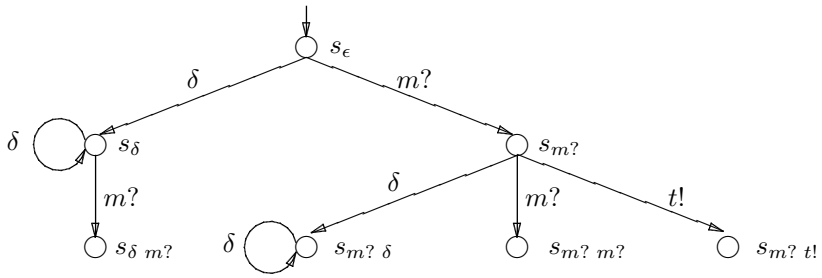


Figure 11.5: Hypothesis for IOLTS L_2 when n is incremented to 2.

11.4 Heuristics

As a consequence of the large number of required experiments, the TBM-algorithm has little practical significance. Our hypothesis is that the extension phase of the TBM-algorithm is a root cause in the exponential blow-up of the state-space, since the number of different outputs that can follow an experiment is for most sensible systems severely limited. Reducing the number of newly introduced inputs therefore leads to a large reduction in the state-space that is built. Consequently, the number of experiments needed to build and validate the hypothesis is also reduced. Not all inputs can be removed without compromising the correctness of the TBM-algorithm, only some can. On the one hand, valid suspension automata always remain non-blocking and anomaly-free by removing input transitions [124]. On the other hand, a valid suspension automaton may turn into a non-quiescent reducible or unstable suspension automaton by removal of a randomly chosen input transition. This is demonstrated in Example 11.4.1.

Example 11.4.1. Let M_3 be given by the suspension automaton of Figure 11.4 (page 150). Clearly, M_3 is a valid suspension automaton. Removing the transition $s_0 \xrightarrow{b?}$ will make M_3 non-quiescent reducible, since in that case, $\delta\ b?$ is a valid behavior starting in s_0, but $b?$ no longer is. Removing transition $s_2 \xrightarrow{b?}$ will make M_3 unstable, as the behaviors in states s_1 and s_2 are different, while both states can be reached by one and two δ transitions, respectively. □

In the remainder of this section we study three heuristics that allow us to safely prune the state-space of the hypothesis dynamically, i.e., the heuristics *preserve the validity of the computed suspension automaton*, while, at the same time, the heuristics try to weed out branches in the hypothesis that are uninteresting from some particular point of view. The heuristics achieve this by preventing the addition of inputs that are somehow not rewarding. All heuristics are defined for the hypothesis $\mathcal{H}$, in the context of the TBM-algorithm.

Input causality The first heuristic that we study utilizes the logs of the interactions of a system with its environment, which are often available for diagnostic purposes. Such a log can be represented by a non-empty set of *traces* of an implementation I (i.e., subsets of *traces*(I)), and we refer to such a collection of traces as a *usage profile*. The added value of a usage profile lies in the fact that it implicitly defines a causal relation between possible stimuli. It is such a *causality relation* that is at the basis of our first heuristic: we only wish to add a specific input action to the computed hypothesis when that input action is preceded by an input action that also preceded the new input action in a trace in the usage profile.

Definition 11.4.2. Let I be an implementation, and let U be a usage profile of I. *Input causality* is defined as a relation $< \subseteq (\text{Act}_I \cup \{\perp\})^2$, where $\perp$ is a reserved constant, and:

$$\begin{cases} \perp < b \text{ iff } \exists \sigma \in \text{Act}_U^*, \sigma' \in \text{Act}^*.\ \sigma b \sigma' \in U \\ a < b \text{ iff } \exists \sigma, \sigma'' \in \text{Act}^*, \sigma' \in \text{Act}_U^*.\ \sigma a \sigma' b \sigma'' \in U \\ \perp < \perp \text{ iff not } \exists \sigma, \sigma' \in \text{Act}^*, b \in \text{Act}_I.\ \sigma b \sigma' \in U \end{cases} \tag{11.4}$$

Intuitively, $\perp < \perp$ holds if no input action appears in a usage profile; $\perp < b$ holds if there is a trace in the usage profile for which b is the first input action that appears in this trace; $a < b$ holds whenever there is a trace in which input a is (after some possibly empty sequence of outputs) followed by input b. Note that input causality does not depend on a usage profile *per se*: it can also be derived from available partial specifications or manually constructed via interviews. It involves high-level information, which often does not need deep knowledge about the system. Modifying the input causality relation by hand can be used to select and isolate behaviors that have to (should) be avoided in learning the system.

Example 11.4.3. Suppose $U = \{a?\ x!\ x!\ a?, a?\ b?\ y!\ a?, y!\ b?\ b?\ d?\}$ is a given usage profile. The causality relation that can be obtained from the usage profile is: $\perp < a$, $\perp < b$, $a < a$, $a < b$, $b < a$ and $b < d$. □

Note that in general, it is possible to obtain a causality relation in which there are inputs a that are never followed by another input, i.e., $\forall b \in \text{Act}_I.\ a \not< b$. One way to deal with such inputs is to explicitly 'close' the causality relation, i.e., we add the causality $a < c$ for all inputs c that occurred first in the usage profile. The closed causality relation is denoted $<_c$.

Example 11.4.4. Take again the usage profile of Example 11.4.3. Input $d?$ is, for all traces of U, never eventually followed by another input action. This also follows from the causality relation $<$ that is constructed on the basis of U: neither $d < a$, nor $d < b$, nor $d < d$. If we wish to use input causality as a means to select which inputs will be considered next when extending a hypothesis, we would be stuck after input d. To prevent this from happening, we act as if the system has not had an input before at such a point. This is formalised by the closed causality relation which for this example adds $d <_c a$ and $d <_c b$ to $<$ (only these, since we have $\perp < a$ and $\perp < b$). □

The closed causality relation is used to selectively add new inputs to the hypothesis in the TBM-algorithm. We define the following set of transitions in the context of the TBM-algorithm:

$$T_{cw} \stackrel{\text{def}}{=} \{(s_\sigma, a, s_{\sigma a}) \in S \times \text{Act}_I \times S \mid |\sigma| = n \wedge s_\sigma \in \textit{der}(s_\epsilon) \wedge \textit{trailing}(\sigma) <_c a \}$$

By *trailing*(σ), we mean the last input action in σ, if it exists, and $\perp$ otherwise. T_{cw} contains the set of input transitions that extend states s_σ at depth n with an input a if and only if the input *trailing*(σ) that was last found on the path σ to a state s_σ was causally *before* the input action a. Then heuristic 1 is obtained by replacing the assignment to T in line 9 in the TBM-algorithm with the following assignment: $\mathsf{T} := \mathsf{T} \cup \mathsf{T}_{cw}$.

Penalty functions The selection of interesting inputs can also be based on information derived from the hypothesis model itself. From a learner's (or tester's) point of view, the outputs a system generates are valued higher than the inputs the learner provides: the outputs are in some sense new to the learner (tester). Based on this viewpoint, we aim at quantifying the amount of valuable information a particular behavior (i.e., a trace) adds to the hypothesis. We start with the basic observation that the length of the largest interval of inputs (after removing observations of quiescence, since quiescence means the system remains in a stable but non-verbose state) in a behavior is a good indicator for the amount of its information. Let λ:$\text{Act}_\delta^* \rightarrow \mathbb{N}$ be the function that returns the length of the largest subsequence of input actions (not counting possible observations of quiescence) in a behavior:

$$\lambda(\sigma) = \max\{n \mid \exists a_1, \ldots, a_n \in \text{Act}_I.\ \exists \sigma', \sigma'' \in \text{Act}_\delta^*.\ \sigma = \sigma' a_1 \delta^* \ldots a_n \delta^* \sigma''\}$$

When $\lambda(\sigma) > t$ for some threshold $t \in \mathbb{N}$, it is reasonable to consider the information in behavior σ too low to invest in further investigating this behavior. This is because the interval of input actions of length $\lambda(\sigma)$ has not led to new observations in the meantime; in the extreme case, the system might even 'hang' at such a point, which means that adding any number of inputs will not give rise to a visible output.

Example 11.4.5. Take two traces $\sigma_1 = a?\ b?\ x!$ and $\sigma_2 = a?\ \delta\ b?\ c?\ x!$. We have $\lambda(\sigma_1) = 2$ and $\lambda(\sigma_2) = 3$. □

Let T_{pf} be a set of transitions, defined in the context of the TBM-algorithm, now extended with threshold t as an additional input parameter, as follows:

$$T_{pf} \stackrel{\text{def}}{=} \{(s_\sigma, a, s_{\sigma a}) \in S \times \text{Act}_I \times S \mid |\sigma| = n \wedge s_\sigma \in \textit{der}(s_\epsilon) \wedge \lambda(\sigma) \leq t\}$$

T_{pf} contains the set of input transitions that extend states s_σ at depth n with an input a if and only if the largest interval of input actions that occurs in the path σ is at most size t. Note that adding an input may lead to an experiment with an input interval of size $t+1$; such an experiment will subsequently not be extended further. Alternatively, one could prevent such experiments to be created in the first place by requiring $\lambda(\sigma\ a) \leq t$. Heuristic 2 is then obtained by replacing the assignment to T in line 9 in the TBM-algorithm with the following assignment: $\mathsf{T} := \mathsf{T} \cup \mathsf{T}_{pf}$.

Example 11.4.6. Consider Figure 11.5. Taking $t = 0$ as a penalty, we only add an additional input to experiments that do not yet contain inputs. Using heuristic 2 to decide which states will be extended with another input would not have introduced the transition $s_{m?} \xrightarrow{m?} s_{m?\ m?}$ in Figure 11.5, while all other transitions would remain unaffected. Of course, in practice one wishes to have $t > 0$. □

One can envision several extensions and modifications to this basic heuristic, all of which are in a similar vein. For instance, instead of using a constant t to compare with in the set T_{pf}, one could use a function that depends on, e.g., the length of the experiment σ.

Non-repetitive quiescence Repetitive quiescence is a powerful tool in the test-based modeling as it enables one to find out which behaviors lead to outputs, which never do, and which lead to non-deterministic behavior when both quiescence and actual outputs are valid observations. The observation of quiescence is also quite costly: in practice, it takes time to conclude that no output will come. While the reasons for doing so may sound rather technical, disabling the notion of repetitive quiescence speeds up the execution of the TBM algorithm. Theoretically, it turns the TBM-algorithm into an algorithm for test-based modeling with respect to a slightly weaker testing relation, known as **ioconf** [117]. Let T_q be a set of transitions defined in the context of the TBM-algorithm as follows:

$$T_q \stackrel{\text{def}}{=} \{(s_\sigma, a, s_{\sigma a}) \in S \times \text{Act}_I \times S \mid |\sigma| = n \wedge s_\sigma \in \mathit{der}(s_\epsilon) \wedge \sigma \in \text{Act}^*\}$$

T_q contains the set of input transitions that extend states s_σ at depth n with an input a if and only if the experiment σ did not end with a δ observation. Heuristic 3 is obtained by replacing the assignment to T in line 9 with the assignment: $\mathsf{T} := \mathsf{T} \cup \mathsf{T}_q$.

Example 11.4.7. Again, consider Figure 11.5. Using heuristic 3 to decide which states will be extended with another input would not have introduced the transition $s_\delta \xrightarrow{m?} s_{\delta\ m?}$ in Figure 11.5, while all other transitions would remain unaffected. □

Combining heuristics All of the heuristics proposed in the previous sections are complementary, which means that all heuristics can be combined. Let X be a non-empty subset of $\{cw, pf, q\}$. A combination of heuristics is achieved by replacing line 9 of the TBM-algorithm with the assignment $\mathsf{T} := \mathsf{T} \cup (\bigcap_{x \in X} \mathsf{T}_x)$.

Example 11.4.8. Consider Figure 11.5. Combining heuristics 2 (with $t = 0$) and 3 would have reduced its state space at depth 2 by two states: states $s_{\delta\ m?}$ and $s_{m?\ m?}$, including the transitions that lead to these states would not have been part of Figure 11.5, while all other transitions would remain unaffected. This is a reduction of nearly 30% of the original state space. At greater depth, the influence is even larger. □

11.5 Case study: the conference protocol

The conference protocol provides a rudimentary *chat-box service* to users participating in a conference. A conference is formed by a collection of users that can exchange messages with all conference partners in that conference. The unbounded number of messages that can be exchanged makes the system effectively infinite-state. The partners in a conference can change dynamically using *join* and *leave* primitives. Different conferences can exist at the same time, but a user can only participate in at most one conference at a time. The conference protocol relies on the service provided by UDP, i.e., data packets may get lost or duplicated or be delivered out of sequence but are never corrupted or mis-delivered.

We have used our approach for learning and testing a running ANSI-C implementation of the *conference protocol*. This setup was previously used to benchmark testing theories [10] using *mutant testing*[1]. The mutants are ANSI-C implementations of the conference protocol that have been derived from the correct implementation by deliberately injecting a single error. These erroneous implementations are categorized in three different groups: *no outputs*, *no internal checks* and *no internal updates*. The first group contains implementations that sometimes fail to send output when they are required to do so. The second group contains implementations that do not correctly check whether they are allowed to participate in a conference, and the third group contains implementations that do not correctly administrate the set of conference partners. Given the large set of documented mutants, the conference protocol makes for an ideal setup for measuring the efficacy of the approach for regression testing and for the testing of different configurations, since one can use the documentation to validate the results found by the TBM methodology.

Experimental setup Table 11.1 highlights which combination of heuristics was used for a specific hypothesis. All hypotheses were derived from running the –what is believed to be– correct implementation of the conference protocol, connected to a prototype implementation of our TBM-algorithm. Two different usage profiles were used in our experiments, and these were also used to determine the size of the input interface (the set Act_I) when the input causality was used. The first usage profile (I) has $|\mathrm{Act}_I| = 19$ and is chiefly a run of the conference protocol in which all parties behave nominally. The second usage profile (II) has $|\mathrm{Act}_I| = 31$ and consists of three runs which combine aspects of nominal behavior with 'bad weather' behavior. Each hypothesis represents the best hypothesis that can be guaranteed by 'learning' the implementation for 48 hours. For the operationalization of the observation of quiescence we adopted the standard approach in testing by setting a time-out on the observation of output (i.e., we observe quiescence when the system did not produce output for two seconds when asked for output). The learning hypothesis was operationalized by conducting each derivable (test) experiment 15 times. Note that since longer experiments

[1] The conference protocol implementation and its mutants, together with a more detailed description, are available via `http://fmt.cs.utwente.nl/ConfCase/`

Heuristic ↓ / Hypothesis →	A	B	C	D	E	F	G
cw		x	x	x	x	x	x
pf			1		1	2	2
q				x	x		x

Table 11.1: Characteristics of the computed hypotheses (identified by the letters A through G). A number for heuristic pf indicates the heuristic was used in combination with the named number as threshold. The 'x' for heuristic cw and q indicates that these heuristics were used.

partly rerun shorter experiments, the confidence in the outcomes of shorter experiments is generally much higher.

Figure 11.6 (page 159) shows the growth characteristics in terms of number of states for a given depth for each computed hypothesis (the I and II preceding the number indicate the used usage profile for obtaining the particular hypothesis). The growth characteristic is an important indicator as it can be used to estimate the overall run-time that is required to guarantee a certain depth-of-correctness. For instance, hypothesis I-E is more likely to reach depth-of-correctness 15 than, e.g., I-D. Additional learning time can therefore best be put into I-E.

Test results The computed hypotheses were subsequently used to test the 27 mutants of the system. For this, we used standard model-based testing techniques. We used a test suite consisting of tests that aimed at covering each output transition (including quiescence) of a used hypothesis. The test results are listed in Table 11.2. In some cases, it was not immediately clear whether the test-failure was due to an incorrect hypothesis (recall that each experiment was conducted 15 times, which may have been too conservative for some experiments) or the mutant. These cases were resolved by hand using the informal documentation and a formalization thereof.

Profile ↓ / Hypothesis →	A	B	C	D	E	F	G
I	1	7	13	10	15	9	10
II	1	10	13	14	20	12	13

Table 11.2: Test results obtained by testing mutants of the conference protocol against the derived hypotheses. The figure indicates the total number of correctly identified mutants.

Analysis The derived hypotheses are remarkably effective at singling out the mutants, although some hypotheses perform significantly better than others. For instance, the combined detection power of hypotheses I-E and II-E (there is a large overlap in the mutants that were detected) turns out to be 85% of all mutants.

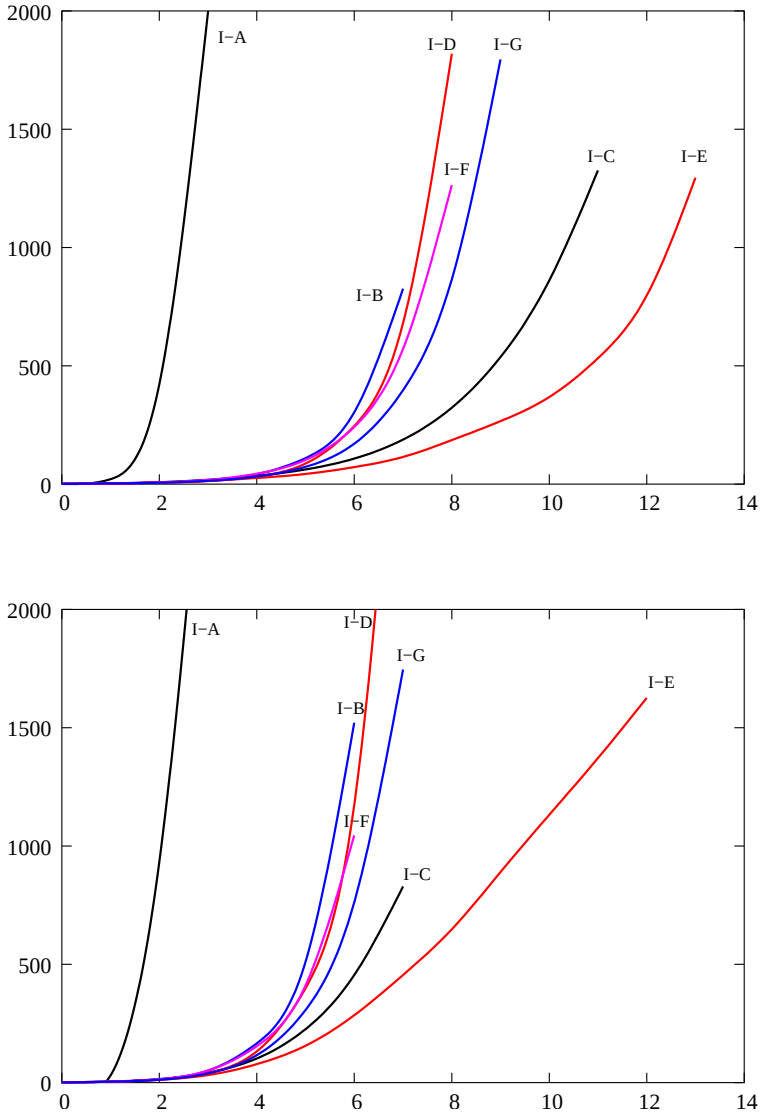


Figure 11.6: Growth characteristics for the hypotheses with inputs determined by usage profile I and II. The horizontal axis denotes the depth of a particular hypothesis; the vertical axis depicts the number of states of a particular hypothesis.

Analyzing the influence of depth-of-correctness on the defect detection capability, we find that usage profile I and usage profile II give slightly different results. For usage profile I, defect detection at a given depth-of-correctness is smaller than defect detection at the same depth for usage profile II. Since usage profile II includes 'bad weather' behavior, this may suggest that most robustness issues can be found at relatively small depths. This may be explained from the fact that programming for robustness is generally trickier than programming for nominal behavior. Issues with nominal behavior generally show at greater depth. An explanation for this may be the increase in intrinsic complexity in nominal behaviors with depth.

It is also clear from the test results that there is no single combination of heuristics that should be used, even though the combination of heuristics that was used for hypothesis E turned out to be quite effective in our setting. Experiences with other combinations of heuristics and different usage profiles (not reported here) show that the usage profiles appear to be a minimum requirement. A combination of several heuristics using more than one usage profile to compute different hypotheses appears to be most effective. The effect of the heuristics is clearly illustrated by the great difference in detection power of the computed models without heuristics (only I-A and II-A), and with heuristics (all other hypotheses).

11.6 Summary and perspectives

In this chapter, we described a pragmatic approach to obtaining models of a system using black-box testing techniques, with the goal of using these models for regression testing. The approach has been demonstrated using a well-known case study and the effectiveness of the approach has been illustrated using mutant-testing. The results of the case study, viz. the detection of 85% of all mutants illustrate that the approach is feasible, and, moreover, effective for regression-testing and for the testing of different configurations.

From a practical point of view, the results are encouraging. Still, there is also some room for improvement, which is demonstrated by the fact that in our case study, approximately 15% of the mutants elude detection. Concerning issues for future research, we feel that it is important to develop additional heuristics and use techniques from statistics. It is important to ascertain that the new heuristics respect validity of the computed hypothesis if one wishes to stay within the realm of **ioco**-based testing. However, the issue of cleverer ways to represent a hypothesis becomes more important with the increase of the state-space, but also with the availability to somehow observe the system's state. While our experiments show that for now, representation is not yet an issue, it will become problematic when experiments are run for weeks rather than days. At that point, representation techniques such as employed and developed by Angluin [2] become important. Reconciling Angluin's L^* algorithm with **ioco**-based testing may, however, be quite tricky, if not impossible, as (1) all transformations have to respect the validity of suspension automata, and (2) the basic starting assumptions are different (e.g., **ioco**-based testing does not require the implementation to have a

finite number of states).

Applying the TBM methodology in an industrial setting is expected to require little effort, whereas the potentials are immense, as indicated by our case study. The effort that is required is certainly less than the effort that is required to apply MBT techniques, as a major time-consuming factor in MBT is the manual construction of a suitable model. There is, however, a large overlap in the techniques and tools that are required to get the TBM methodology and MBT techniques up and running and interact with the actual system; in fact, the TBM methodology may be a first step in introducing MBT techniques in an industrial environment. There are some practical issues to be overcome. For one, the learning hypothesis implies that the outcomes of an experiment are only reliable when the experiment was conducted a number of times. However, this requires a hard or soft reset of the system to ensure that the system is again in its initial state. In large systems, such a reset may consume a significant amount of time, and, therefore has a negative impact on the performance of the TBM algorithm. In some cases, a reset may be avoided, e.g., if one can (partially) observe the current state of the system. The theoretical implications of partial observability of state information have not been studied in this context, and they may give rise to more efficient algorithms.

Chapter 12

Model-based diagnosis

Author: J. Pietersma, A.J.C. van Gemund

12.1 Introduction

Fault diagnosis is the process of finding the root causes of non-nominal system behavior. In the last decades it has become more challenging as technology development shows a trend of ever increasing system complexity driven by innovation, as reflected by Moore's Law, and by demand from society. Software which for many systems has a crucial role of integrating system functionality, creates even more complexity. Furthermore, diagnosis is a combinatorial problem with respect to the number of system components. Due to the increasing complexity and the inherent combinatorial nature, fault diagnosis is a challenging engineering problem.

During the integration and testing phase difficulties with diagnosis can increase a system's time-to-market. During operational life, diagnosis also affects the dependability of systems. Inefficient diagnosis may lead to long down-times and reduced system availability. For systems that have limited or no possibility for inspection and repair, e.g., satellites, self diagnosis is the very first step in system autonomy. This requires a built-in diagnosis function, usually implemented in software, which further increases the complexity of the system and its development.

From the field of artificial intelligence, Model-Based Diagnosis (MBD) has been put forward as a solution for this problem. MBD is a method of automated model-based reasoning that automatically infers a diagnosis from a compositional, behavioral system model and real-world observations. MBD is faster and less error-prone than human reasoning. In contrast to methods that directly map symptoms to causes, MBD captures correct system functionality and therefore isolates all faults that cause deviating system behavior, including those that were not anticipated. MBD also adapts

better to evolving system design because models are derivable from this design, as they capture system functionality that closely resembles it.

To be applicable in an industrial setting, MBD should efficiently yield diagnostic results with a certain level of quality. This has a number of consequences.

(1) MBD should use an implementation of a sound and efficient, i.e., fast, fault finding algorithm.

(2) Models should be amenable to fast fault finding. This puts a limit on model complexity.

(3) The diagnostic quality should be predictable and quantifiable.

(4) The benefits of MBD should outweigh the costs that are invested in modeling.

In the past, academia have given most attention to the first point. The last three points are typical issues that surface in an industrial environment, such as ASML. These three issues, model complexity, diagnostic quality, modeling costs, and their underlying relations are the subjects of our research as performed in the Tangram project.

In this chapter we discuss the obtained research results. We begin by explaining the principles behind MBD. We discuss the modeling issues that most strongly affect model complexity, diagnostic quality, and modeling costs. These issues are: model composition, the use of behavioral modes, modeling of sensors, modeling time-dependent behavior, and model derivation. Furthermore, we discuss how to measure diagnostic quality and present a summary of all research results. In Chapter 13 we present the actual case studies as performed in the Tangram project and discuss the costs and benefits of MBD as applied at ASML.

12.2 Methodology

Fault diagnosis finds the root causes of differences between expected and actually observed system behavior. Based on first principles, this expected behavior is compositionally modeled with logical constraints which are conditional upon the system health.

Figure 12.1 shows the relation between reality and models. For simulation a model is used to predict the output based on the known input and health state. This is also used for model-based testing in which a test outcome is based on the comparison between real and simulated outputs, see Chapters 9 and 10. For diagnosis the unknown health state is inferred from the known, i.e., observed inputs and outputs. The inference requires the algorithm to automatically reason in terms of the model and observed variables. The automatic reasoning traces back the causes of observed behavior to faulty components. This type of reasoning requires a declarative rather than an imperative model. The latter are more commonly used for simulation, such as used, e.g., with Matlab.

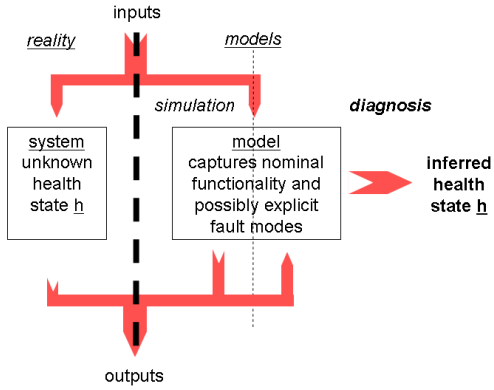


Figure 12.1: Relation between reality and models for simulation and diagnosis.

MBD was first proposed in [107] and [71] and implemented in the General Diagnostic Engine (GDE). Since then a lot of effort has been put in making MBD computationally more efficient. Different strategies have been pursued such as conflict detection [125], hierarchical approach [48], and using different knowledge representations [47]. At Delft University of Technology, MBD is implemented with a modeling language called LYDIA. The core of the language is propositional logic, enhanced with syntactic sugar for easier modeling. Currently a number of different diagnostic engines and a simulator are available[1]. An example of another implementation of logic based MBD is the Livingstone II software tool [2]. The latter has been used successfully in an autonomous, fault-tolerant control experiment onboard NASA's the Deep Space One probe.

We use the following notations and theory for MBD. First, we present some basic terminology.

Definition 12.2.1. [Diagnosis Problem] A diagnosis problem Δ is the ordered triple $\Delta = \langle M, C, O \rangle$, where M is the model that represents the system behavior in terms of propositions over a set of variables V, C is the set of components contained in the system, and $O \subset V$ is a set of observable variables in M.

Let n represent the component index number and $N = |C|$ the total number of components. For each component $c_n \in C$ there is a corresponding variable h_n representing its health mode that determines component behavior. We will call these variables h_n *health variables* and $\underline{h} = (h_1, \ldots, h_n, \ldots, h_N)$ the system *health vector*.

We assume that O can be divided in an input vector $\underline{x}$ and output vector $\underline{y}$ such that $O = \underline{x} \cup \underline{y}$. For a stateless system the model is represented with a function f_M that

[1] `http://fdir.org/lydia`

[2] `http://www.nasa.gov/centers/ames/research/technology-onepagers/livingstone2-modelbased.html`

maps the health and input vector to an output vector,

$$\underline{y} = f_M(\underline{x}, \underline{h})$$

Practically, it is never possible to model f_M for all possible health modes as many failure modes are not anticipated. More fundamentally, the inverse of f_M, which would trivially yield a diagnosis, is hard to derive. In most cases the inverse does not exist as there are many possible explanations for a particular observation in which case there is no behavioral mapping to $\underline{h}$.

Because of this irreversible mapping we resort to describing M in terms of behavioral constraints expressed as propositional sentences. This leads to the following definition for a diagnosis.

Definition 12.2.2. [Diagnosis] A *diagnosis* for the system $\Delta = \langle M, C, O \rangle$ and observations $\underline{x}_0$ and $\underline{y}_0$ is a set D of diagnosis candidates $\underline{d}_k$ such that $M \wedge (\underline{x} = \underline{x}_0 \wedge \underline{y} = \underline{y}_0) \wedge \left[\bigwedge_{\underline{d}_k \in D} \underline{d}_k = \underline{h}\right] \not\models \bot$.

From Definition 12.2.1 and 12.2.2 it is visible that a diagnosis algorithm should use an entailment mechanism that finds the set D that is consistent with $M \wedge O$. The system model M describes the system behavior in terms of relations between variables in the Boolean domain. The relations are expressed with standard Boolean connectives $\neg, \Leftrightarrow, \Rightarrow, \wedge, \vee$.

To illustrate MBD we use the following example system. We abstractly model a valve as a component with an incoming and outgoing flow, denoted f_i and f_o respectively. For a healthy valve, the valve control variable c determines the outgoing flow. A true control variable implies an open valve for which the outgoing flow is equal to the incoming, and a false control variable implies a closed valve for which the outgoing flow is zero, i.e., false. The propositions are,

$$\begin{aligned} c &\Rightarrow (f_o = f_i) \\ \neg c &\Rightarrow \neg f_o \end{aligned}$$

In our modeling language LYDIA this corresponds to the following code,

```
if (control) {flowOut = flowIn;}
else {not flowOut;}
```

where $control$ corresponds to c, $flowOut$ to f_o, and $flowIn$ to f_i.

For a diagnosis that is not trivial, the number of observable variables is typically limited. For this component we assume that only the control variable and the outgoing flow are observable. The first half of Listing 12.1 shows the complete LYDIA model in which the valve behavior is dependent on the health variable h.

In LYDIA, the keyword **system** indicates the definition of a component. Health variables, which are solved by the LYDIA diagnostic engine, are declared by setting the attribute **health** to **true**. The attribute **probability** declares an a priori probability distribution for the health variable h. This is used as a search heuristic for the diagnostic

```
system Valve (
  bool h, flowIn, flowOut, control )
{
  // variable attributes
  attribute health (h) = true;
  attribute probability (h) = h ? 0.99 : 0.01;
  attribute observable (flowOut) = true;
  attribute observable (control) = true;

  // valve model: control=true implies open valve
  if (h) {
    if (control) {flowOut = flowIn;}
    else {flowOut = false;}
  }
}

system twoValves (
  bool h[1:2], flowIn, flowOut[1:2], control[1:2] )
{
  system Valve valve[1:2];

  forall (i in 1 .. 2) {
    valve[i] (h[i], flowIn, flowOut[i], control[i]);
  }
}
```

Listing 12.1: LYDIA valve model.

inference. It is also used for the ranking of the inferred diagnoses, since observations usually admit multiple diagnoses, as shown in the sequel. The attribute **observable** marks those variables that are observable.

As $flowIn$ is not observable the only exclusive fault that can be detected is that of a leaky valve. The observations for this fault are $control = false$ and $flowOut = true$ which is only consistent for $h = false$. For all other observations $h = false$ and $h = true$ are both consistent, which illustrates that limited observability typically leads to limited diagnosability, i.e., multiple or ambiguous diagnoses.

As in many systems, components share connections which can be sensed for better diagnostic reasoning. Consider a system of two identical, parallel valves A and B as depicted in Figure 12.2. The LYDIA model is shown in the second half of Listing 12.1. Again, each valve has a control variable that, for a healthy valve, determines if the valve is open or closed, and a variable for the outgoing flow. Both valves are fed with the same ingoing flow.

If we observe that both valves are commanded open ($controlA = controlB = true$) while only valve B has outflow ($flowOutA = false, flowOutB = true$) we obtain the diagnoses and accompanying a priori probabilities listed in Table 12.1. Diagnosis 1 and 2 are single faults candidates that are equally probable, while diagnosis 3 is a double fault that is less probable.

The second diagnosis, however, does not represent expected physical behavior as

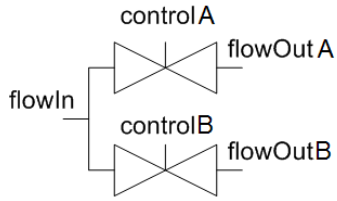


Figure 12.2: Two valves system.

	hA	hB	Pr
1.	$false$	$true$	0.0990
2.	$true$	$false$	0.0990
3.	$false$	$false$	0.0001

Table 12.1: Initial diagnosis for two valves, with $controlA = controlB = true$ and $flowOutA = false, flowOutB = true$.

$hA = true$ implies $flowIn = flowOutA = false$ which implies that the second valve would have a failure that spontaneously generates flow. To exclude this non-physical behavior from the model we need to extend it with the following constraint,

```
if ( not flowIn ) { not flowOut; }
```

Now the diagnosis reduces to either a single failure of valve 1 or a double failure of both valves, corresponding to the expected physical behavior of the (failed) system.

12.3 Model composition

In this section we discuss the first steps in modeling a system for fault diagnosis. These steps are to choose, relevant to diagnosis, the system boundaries and within those boundaries the level of detail. These choices affect the complexity and usefulness of the diagnosis. The required level of detail dictates the composition of the model. The health of each component that is identified in the model is modeled with one health variable. A principal assumption in the theory of MBD is that health variables are considered to be independent of each other. This means that in a proper diagnosis model, components need to be modeled such that this is indeed the case. As mentioned in Section 12.2, component behavior is modeled conditional on this health variable.

The system boundary and model composition determine the number of health variables in the model. This number determines the complexity of the model and, in the worst case, the diagnosis time increases exponentially with it. It also determines the

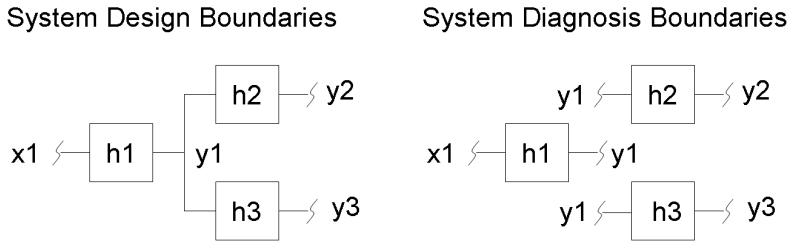


Figure 12.3: System boundaries for design and diagnosis, with y_1 observable.

practical usefulness of a diagnosis. Hence a proper decision on boundary and composition has important consequences for the complexity and usefulness of the diagnosis.

Selection of the system boundary for diagnosis need not necessarily be in agreement with the boundaries as designed. Consider a simple system topology as depicted in Figure 12.3. On first sight this system seems to consist of three components that all share one variable y_1. However if y_1 is known (i.e., an observable variable), then a solution for the diagnosis of all three components can be found independently of each other. Thus, for the purpose of diagnosis, we can consider the three components to be three independent *systems*. Generally speaking, we can find system boundaries by identifying the set of components that are connected to all other components by observable variables only. The diagnosis for this isolated set can be found independently from the others.

To determine the level of detail in the model we need to decide on what level of detail we require our diagnosis to be. For a digital circuit, e.g., the smallest unit of interest may be a gate, while for a wafer scanner the subsystem level may be sufficient. As a general rule of thumb we look at which level system repair and planning actions, in response to diagnosis, take place. Components that are replaced as a whole by the system operator should be the smallest unit represented in the model. These components are also known as Line Replaceable Units (LRU's). For autonomous systems, redundant components are the smallest unit represented in the model.

12.4 Behavioral modes

Fundamental for MBD is that the models describe *nominal* system behavior. This makes it possible to have a model that lies closely to the system design and diagnose faults that are not anticipated as opposed to models based on anticipated system failures, such as models that directly map symptoms to root causes. In this context, faults are any deviance from nominal behavior. For example, we model an inverter gate as

follows,

```
h => (y=not(x));
```

leaving the faulty behavior implicit. This type of models is called *weak* because it is completely unconstrained for faulty behavior.

For some components however, the failure mode may actually be known and this knowledge may be used to improve the diagnostic quality, i.e., the diagnosis not only explains which component is the root cause but also *how* it has caused the observed behavior. For example, if an inverter has a stuck-at-zero fault mode, we may extend the earlier inverter model to,

```
h => (y=not(x));
not(h) => not(y);
```

Such a model is called strong. While improving a diagnosis by providing information on how the component has failed, this model also poses a problem if a component has no unique failure mode. Consider the observation $x = True, y = True$, which can not be explained by the strong model. This inconsistency results in an empty diagnosis without any use (zero quality).

From these examples it is clear that we need a combination of both modeling methods that exploits known failure modes but will never lead to an inconsistency between model and observations. For this purpose we introduce more failure modes each with its own type of faulty behavior. The theory of MBD as presented in 12.2 can be extended to include variables in a finite integer domain. This extension and its effect on complexity is discussed in [46].

In LYDIA, finite integer domains are implemented with the enumerations type **enum**. Using this type and the **switch** statement, the inverter model becomes,

```
type Health = enum {nominal, stuck0, stuck1, unknown};
Health h;
switch (h) {
    Health.nominal -> {y=not(x);}
    Health.stuck0 -> {not(y);}
    Health.stuck1 -> {y;}
    Health.unknown -> {}// undefined
}
```

It is recommended to always leave one mode undefined, in this case the unknown mode. This insures that, for any observation, there is always at least one implicit explanation.

Another approach to prevent inconsistencies is to have a complete coverage of possible combinations of inputs and outputs. For the inverter model with a Boolean health, a failure simply negates the nominal behavior. This can be modeled as,

```
h = (y=not(x));
```

There are two disadvantages to this otherwise powerful approach. First, the model can no longer be used for sequential diagnosis, i.e., diagnosing multiple, sequential observations with one model to improve the quality of a diagnosis. With a weak model it is possible to diagnose any sequence, while with a complete model only one sequence

```
system Sensor (bool pos, pos_meas)
{
  bool h;

  h => (pos_meas=pos);

  attribute health (h) = true;
  attribute probability (h) = h ? 0.99 : 0.01;
}

system main (bool pos_meas1, pos_meas2)
{
  bool pos;

  system Sensor sensor1, sensor2;

  sensor1 (pos, pos_meas1);
  sensor2 (pos, pos_meas2);

  attribute observable (pos_meas1, pos_meas2) = true;
}
```

Listing 12.2: Model of two position sensors.

belonging to a single behavioral mode can be diagnosed. Sequential diagnosis is discussed in more detail in Section 12.6 in relation to the modeling of time-dependent systems. Second, current algorithms are optimized for weak fault models.

12.5 Sensors and the real world

A very useful source of diagnostic knowledge is the consistency between different components and their representation of the physical state of the system and its environment. It is important to realize that in many cases observations are in fact data collected by sensors, which should also be modeled as components with behavior that is conditional on their health modes. The nominal behavior of a sensor is correct measurement of physical parameters in the real world. An underlying model of the physical behavior of the system and its environment can thus be exploited for diagnostic purposes.

This is illustrated with an example model of two location sensors shown in Listing 12.2. If we observe contradicting values for pos_meas1 and pos_meas2 we get a diagnosis that at least one of the sensors has failed. If we would add a third sensor to this model, a single deviating sensor would always be unambiguously diagnosed. Such a model would in fact implement majority voting for diagnostic purposes.

The above can be extended by also including actuators that respond to and act on real world variables. An example of this is shown in Listing 12.3. This is a model of a system with two position sensors and two safety pins (with sensors) that must go up

Measured positions				**Best ranked diagnosis**				**Scenario**
pos1	pos2	pin1	pin2	sensor1	sensor2	pin1	pin2	
false	false	false	false	true	true	true	true	nominal
true	true	false	false	true	true	true	true	nominal
true	false	true	true	**false**	true	true	true	out of sync,
				true	**false**	true	true	but pins up
false	true	true	true	**false**	true	true	true	out of sync,
				true	**false**	true	true	but pins up
true	false	false	true	**false**	true	**false**	true	out of sync,
				true	**false**	**false**	true	but pin1 is up
true	false	true	false	**false**	true	true	**false**	out of sync,
				true	**false**	true	**false**	but pin2 is up
true	false	false	false	**false**	true	**false**	**false**	out of sync,
				true	**false**	**false**	**false**	but pins not up

Table 12.2: Diagnosis scenarios for synchronized pin model.

in case the position sensors are no longer synchronous. Here we have added another physical constraint that says that the real (physical) positions should always be in sync. This can be thought of as two sensors connected very robustly to a mechanical frame. If necessary, the structural health of the frame can also be made conditional on its health. Table 12.2 shows the best ranked diagnosis results for some scenarios. Note that we cannot discern between two position sensors failing (interchangeable) but that we can discern between pin failures.

12.6 Time-dependent systems

Up to this point, we have discussed purely combinatoric systems. In practice, however, we encounter systems with time-dependent behavior such as intermittency, propagation time delays, and state. Consider again the single valve model from Section 12.2 (weak model) and suppose that we have a timed sequence of observations where each observation x becomes a function of time (i.e., $x(t)$). If at time t_1 the observations are $c(t_1) = false$ (closed valve) and $o(t_1) = true$ (flow out) then the diagnosis becomes $h(t_1) = false$ (leaky valve). Let the second set of observations be $c(t_2) = false$ and $o(t_2) = false$, indicating that the leak has stopped. If we assume $h(t_1) = h(t_2) = h$, i.e., health that is not time-dependent, we use the following model,

```
system static_valve (bool h, i[1:2], c[1:2], o[1:2]) {
    system Valve valve[1:2];

    valve[1] (h, i[1], c[1], o[1]);
    valve[2] (h, i[1], c[1], o[2]);
}
```

The diagnosis then becomes $h = false$, as this also explains the second set of observations due to the weak model. Note that this sequence of observations does not yield

```
system Sensor (
  bool pos,          // the real (physical) position, input
  bool pos_measured)     // the measured position, output
{
  bool h;   // health

  attribute probability (h) = h ? 0.99 : 0.01;
  attribute health (h) = true;

  h => (pos_measured=pos);
}

system main (
  // measured position of chuck 1 and 2
  bool pos1_measured, pos2_measured,
  // measured position of pin 1 and 2
  bool pin1_measured, pin2_measured
)
{
  bool pos1, pos2;  // real physical positions of chuck 1 and 2;
  bool pin1, pin2;  // real physical position of pin1 and pin2;
  // synchronsiation condition and measured one
  bool sync, sync_measured;

  system Sensor sensor1, sensor2, pin1, pin2;

  sensor1 (pos1, pos1_measured);
  sensor2 (pos2, pos2_measured);

  pin1 (pin1, pin1_measured);
  pin2 (pin2, pin2_measured);

  // synchronisation condition
  sync = (pos1 = pos2);
  // synchronisation condition measured
  sync_measured = (pos1_measured = pos2_measured);

  // if we measure out of sync the pins go up
  not(sync_measured) => (pin1 and pin2);

  // synchronisation, like a physics law, e.g., gravity,
  // this can also be implemented as the
  // synchronisation health of e.g., the software
  sync = true;

  attribute observable (pos1_measured, pos2_measured,
    pin1_measured, pin2_measured)= true;
}
```

Listing 12.3: Model of two synchronized sensors and reactive pins.

a diagnosis for the 'complete' valve model discussed in Section 12.4. For this model, h can explain only one type of behavior and is therefore unable to explain the temporal variations. The same is true for strong models.

If we want better time resolution and we want to know where in the sequence, i.e., at what time, the valve has leaked, we need to model h as time-dependent variable, similar to c and o. This leads to the following model,

```
system dynamic_valve (bool h[1:2], i[1:2], c[1:2], o[1:2]) {
    system Valve valve[1:2];

    valve[1] (h[1], i[1], c[1], o[1]);
    valve[2] (h[2], i[], c[2], o[2]);
}
```

Now the diagnosis becomes $h[1] = false, h[2] = true$.

We can extend this approach to modeling temporal relations, e.g., two serial buffers with a propagation delay of 1 time unit,

```
system Buffer (bool i,o) {
    h => (o=i);
}

system serial_buffers (bool i[0:2], o_a[0:2], o_b[0:2]) {
    system Buffer buffer_a[0:2], buffer_b[0:2];

        buffer_a[1] (i[0],o_a[1]);
        buffer_b[1] (o_a[0], o_b[1]);

        buffer_a[2] (i[1],o_a[2]);
        buffer_b[2] (o_a[1], o_b[2]);
}
```

This models temporal constraints as the output of each buffer is dependent on prior input. Note that LYDIA provides a shorthand for array expansion with the **forall** statement.

This temporal modeling technique also allows recurrent systems and introduces state behavior in models. Consider a Set/Reset (SR) latch as shown in Figure 12.4. When set, s is $true$, the outputs q and $\overline{q}$ are latched $true$ and $false$ respectively, after

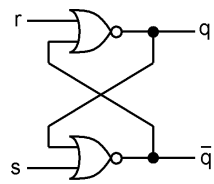


Figure 12.4: SR latch diagram.

some propagation delay. For reset r, the outputs are opposite. This latching is imple-

mented with recurrent relations that express state behavior for q and $\overline{q}$. The model is as follows

```
forall (k in 1..2) {
  h1[k] => ( q[k] = (r[k-1] nor _q[k-1]) );
  h2[k] => ( _q[k] = (s[k-1] nor q[k-1]) );
}
```

where k models time in terms of discrete steps.

We have generalized the above concepts by introducing the concept of (real-valued) delays, which allow for more convenient modeling without the need to introduce separate model instances for each discrete time step. In [101], we have shown that by symbolically converting real-valued propagation delay models to combinatoric ones, we are able to efficiently diagnose time-dependent systems.

12.7 Diagnostic quality

Due to limited observability and implicit fault models, a diagnosis typically yields many solutions. The number of solutions is indicative of the diagnostic quality of a model. In the following we discuss how to measure the residual uncertainty of a diagnosis after making an observation and how to reduce this uncertainty. We measure the uncertainty as the amount of information contained in the set of diagnosis candidates D. We use Shannon entropy [111] to quantify this amount of information and denote it with H. Shannon entropy is a logarithmic formula that expresses the amount of information for stochastic variables and, when used with base 2, expresses it in bits.

We illustrate these concepts using a Boolean model of a digital circuit, consisting of the three inverters depicted in Figure 12.5. The behavior of a single inverter with health h_c, input x_c, and output y_c is defined by the proposition,

$$h_c \Rightarrow (y_c = \neg x_c)$$

which is equivalent to the model in Section 12.4. Let 1 and 0 represent $True$ and $False$ values, respectively. Thus the set of variables V is $\{x, z, y_1, y_2, h_1, h_2, h_3\}$ and their domain denoted by S is $\{0, 1\}$.

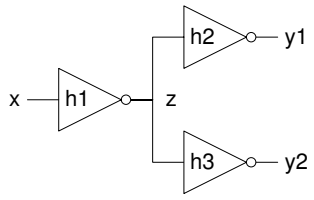


Figure 12.5: Three-inverters example.

Without any observations, there are no constraints for $\underline{h}$. Hence, any value for $\underline{d}_k$ is consistent with this model and D is the enumeration of all these values. In this case, H is equal to H_0, the entropy of N independent stochastic variables,

$$H_0 = -N \sum_{c=1}^{|C|} \Pr(\neg h_c) \log_2 \Pr(\neg h_c) + \Pr(h_c) \log_2 \Pr(h_c)$$

For this model with a priori probabilities $\Pr(h_c = 0) = 0.01$ and $\Pr(h_c = 1) = 0.99$, $H_0 = 0.24$ bits.

Suppose the observed system variables are (x, y_1, y_2) and the observation $O = (x = 1) \wedge (y_1 = 0) \wedge (y_2 = 1)$ is made. The observation $y_1 = 0$ indicates a system health problem. Table 12.3 lists D for this case and shows the typical effect of weak fault models, i.e., many solutions. Although $\underline{d}_1 = (1, 0, 1)$ is the most probable candidate, the actual diagnosis might, e.g., also be $\underline{d}_2 = (0, 1, 0)$, i.e., inverter 2, is healthy but inverters 1 and 3 are broken. To prioritize the inference of $\underline{d}_k$ with the most probable faults, an a priori probability distribution for h_c is defined. Based on this probability distribution we compute the expected entropy after making the observations O. With the entropy,

$$H = -\sum_{D} \Pr(\underline{d}_k) \log_2 \Pr(\underline{d}_k | O)$$

the expected entropy becomes,

$$\mathrm{E}[H|O] = \sum_{\Omega} \Pr(O) H$$

where Ω is the set of all possible values for the set of observable variables O. Computation of the expected entropy, leads to $\mathrm{E}[H|O] = 0.13$ bits. As intuitively might be expected, observation of O reduces the expected uncertainty.

	$\underline{d}_k$	$\Pr(\underline{d}_k \mid O)$
1	$(1, 0, 1)$	0.970492
2	$(0, 0, 1)$	0.009803
3	$(1, 0, 0)$	0.009803
4	$(0, 1, 0)$	0.009803
5	$(0, 0, 0)$	0.000099

Table 12.3: Diagnosis set D, for three-inverters example, with $0 =$ false and $1 =$ true.

The expected entropy is an important parameter for diagnostic quality. The effectiveness of methods to increase diagnostic quality can be measured with this parameter. Methods that may decrease the expected entropy and improve the quality are adding behavioral modes to the model as discussed in Section 12.4, and increasing the observability of a model by either increasing the number of observed variables (in the real world this also increases the number of health variables), and measuring for a longer duration while assuming a constant health.

12.8 Automatic model derivation

Despite the run-time savings in diagnosis time, MBD still requires an investment in modeling time. This investment may still prohibit MBD to be accepted in an industrial setting. A possible solution to reduce the time investment in manual modeling is automatic model derivation from design. As part of the Tangram project we have investigated three possible model sources.

Initially we have chosen the model topology in the EPIN case study, as discussed in Chapter 13. In this particular case, part of the system structure, the connections between components (sensors, actuators, wiring, and logic) was available in Netlist format. This format provides a source for the component connections that are represented in the model topology. For this purpose the Netlist source was scanned for connections between components. These connections are instantiated as simple buffers connecting the different components. The components are instantiated as skeleton models for which the behavior can be added manually.

In the WS case study, also discussed in Chapter 13, we investigated the possibility to derive model behavior from VHDL[3] implementations. In practice, it is not possible to convert a model from the (partially) imperative VHDL language to a declarative LYDIA model. However, in a number of limited cases it is possible to derive a behaviorial description from the original VHDL code. This is true for simple Boolean expressions. We have also demonstrated that we are able to deal with state behavior following the approach as mentioned in Section 12.6.

In addition, for some systems it may already be known what conditions are unwanted. These conditions are expressions of non-nominal behavior. Healthy behavior can then be partially modeled as the negated unwanted behavior. For ASML this was true for the safety monitoring and emergency logic in the WS case study. The purpose of the logic is to monitor the state of the system and react with an emergency signal, followed by a full system stop in case of an emergency. The logic for the system was implemented in VHDL and used as the basis for the diagnosis model which was later on extended with more expert knowledge on nominal behavior.

12.9 Summarized results

In summary, the results of our research are as follows. We have investigated model complexity and its effect on diagnosis speed for different knowledge representations. Specifically we have investigated the effect of modeling in the finite integer domain and the effect of Boolean and direct encodings [46]. Furthermore, we have studied how to model systems with time-dependent behavior and proposed a method of limiting model complexity, thereby speeding up diagnosis [101]. Our approach makes use of delay calculus and delayed signals to get all time dependant behavior outside the health conditional equations. This allows the problem to be solved with a static model. This

[3] VHSIC Hardware Description Language, VHSIC stands for Very-High-Speed Integrated Circuit

approach is significantly more efficient than a discrete time approach were for each time step a model is instantiated and constraints are dependent on previous time steps.

Borrowing from Information Theory, we have used the existing entropy heuristic that was used in earlier work for measurement selection as a general quantifier for diagnostic quality. This was done by establishing a empirical relation between entropy and diagnostic accuracy [99]. Also the relations between observability and this entropy have been established [100], thereby making entropy an important feedback parameter in the model development process. The observability includes the quantity, quality, and duration of observations. As a result, it is now possible to express the diagnostic quality of a model in one entropy number. We have also extended the use of entropy as a heuristic for test selection and sequencing. We have shown that we can frame the test selection and sequencing problem by modeling the system together with the test setup [102]. Optimal tests and sequences are found by applying the entropy heuristic. Efficient approximations of entropy calculations have also been investigated.

Finally, in a of number case studies, we have demonstrated that MBD can be applied in an industrial environment, also by non-MBD experts. We have shown that modeling costs can be reduced by partly automatic model derivation from existing implementations expressed in Netlists and VHDL and by making practical abstractions. Also the earlier mentioned test selection and sequencing yields additional benefits of modeling for diagnosis.

12.10 Conclusions

We have discussed a number of issues related to MBD and the ‘art’ of modeling for fault diagnosis. We have discussed the importance of proper boundary and composition selection, the use of behaviorial modes, and the effect on system complexity. We have shown the importance of modeling interactions with the real world and its role in the diagnostic process and time-dependent behavior. Finally we have shown how to reduce investment in modeling time by deriving models from implementation and how to measure the diagnostic quality of models.

All-in-all, our research provides valuable lessons for practical modeling issues as encountered in the Tangram project. The actual application of MBD by ASML, resulting in the initiation of a transfer project (see Chapter 15), shows that we dealt successfully with these issues, and this provides a positive lookout for the further development of MBD.

Chapter 13

Costs and benefits of model-based diagnosis

Author: J. Pietersma, A.J.C. van Gemund

13.1 Introduction

In this chapter we discuss the costs and benefits of Model-based diagnosis (MBD) as applied within ASML, based on the results of the case studies performed in the Tangram project. At ASML, the effort and time consumed by fault diagnosis are mostly noticeable during integration and testing, and during operational life. Hence we discuss these two life cycle phases in detail. We will also briefly address the other life cycle phases. First, we illustrate the current way of working with two real-life examples that occurred during operational life, and give our general view on the way of working. Second, we present the results of the case studies. Next, we present how MBD can improve the current situation, i.e., diagnose faster and with less effort by investing time in building models beforehand.

13.2 Current way of working

Our first example of the current way of working is that of a failed initialization of the software that controls the wafer scanner air mounts. These air mounts support the metrology frame and protect it against vibrations. Lorentz motors provide these air mounts with force damping. Initially a first line service engineer dealt with this issue but was unable to solve it. Troubleshooting continued in telephonic contact with a second line engineer with more experience. Inspection of the motor temperature sensor values indicated that one was abnormally high, $\pm 100^{\circ}$ C compared to the expected

$\pm 20^{\circ}$ C. From cautious manual inspection it was suspected that this reading was wrong. This was corroborated by the fact that no power was consumed by this particular motor. Replacement of air mounts sensor board did not resolve this high temperature reading. A resistance measurement of the temperature sensor indicated an open contact of the relevant thermo couple. A spare part was ordered and installed after the weekend. This resolved the high reading and the initialization problem.

Our second example involves the wafer centering process. Before a wafer is measured and exposed, its position on the chuck has to be centered with sub millimeter precision. This precise placement is done by repeatedly spinning the wafer, measuring the discrepancy and replacing the wafer to decrease the discrepancy. In normal operations the discrepancy would converge in a number of steps to an acceptable low level. If this takes more than a specified number of steps the machine halts operations. After long uptime, one machine showed exactly this problem. Components that contribute to this problem are the actuator, sensors, the spinning table, cables, and sensor boards. One by one these components were replaced without resolving the problem. A third line service engineer was flown in. At a certain moment it was accidentally discovered that the actuator showed movement when the cables were moved. Apparently this was caused by bad wiring and the problem was solved by replacing the cable harness. This whole case took five days to solve. Note that in this case down-time impact was minimized by replacing the whole unit with one from an unused system, and testing the faulty unit on a system not critical for production. This partly mitigated the effects of the lengthy diagnosis process on system availability.

From these examples and our general experience with diagnosis at ASML we argue that the current fault diagnosis process at ASML is largely a manual task based on human knowledge mostly gained from experience. It is only *complemented* by (electronic) documentation, data analysis software, and results of earlier cases. We discern two types of knowledge derived from two types of experience:

(1) Engineering knowledge which captures the way the system works. This type of experience is usually obtained during development and augmented by training. During integration and testing, engineering knowledge of this type is close at hand because systems are initially put together at the development location.

(2) Symptomatic knowledge which represents the way the system usually fails and how to solve it. This type of experience is usually obtained in the operational phase and shared among peers.

Fault diagnosis that is based on both types of human knowledge and experience, encounters the following problems. First, the knowledge is volatile which means that when people leave, the knowledge disappears from the organization and has to be relearned or trained to other employees. Second, the knowledge does not cover unforeseen or never encountered faults. Besides these two general problems, engineering knowledge (type 1) is also scarce during the operational phase as most people with this type knowledge move on to the development of the next generation of systems.

The specific problem with symptomatic knowledge (type 2) is that it can lead to tunnel vision. We explain this mechanism with the following example. Assume that it is known from experience that error X is solved by replacing board A. This will be the reflexive reaction if error X occurs. However, in combination with error Y it is actually board B that is the root cause. In practice, this leads to a lot of unnecessary and costly component replacements.

For completeness we also identify the following issues in fault diagnosis which are not specific for the current approach:

- The system as such produces an enormous amount of information (e.g., real-time data dumps, event and error logs), only part of this is useful for fault diagnosis. From this information overload, service engineers have to select and process the appropriate data.
- The customer support organization needs to provide sufficient training, communication possibilities, and needs to allocate resources according to problem priority set by the customer.
- Relations with the customer are crucial. The customer is the first to be confronted with problem symptoms and also decides what information is accessible and what actions may be performed on the machine.
- The physical accessibility of the machine itself is limited due to space constraints imposed by the high costs of clean room space.

We also point out that, as with any engineering activity, troubleshooting and diagnosis are also affected by cultural aspects. This covers factors such as the proper 'tinkering' mentality and the ability and willingness to question decisions made by superiors.

13.3 Case studies

As part of the Tangram project five ASML subsystems have been modeled for diagnosis. For more information about the principles behind MBD, our implementation with the LYDIA language, and the different aspects of modeling for fault diagnosis, we refer to Chapter 12. Table 13.1 lists the following characteristics of these cases: engineering discipline, whether or not it involved dynamic system functionality, the LYDIA model size, the time spent on the modeling, and the (estimated) improvement in diagnosis time. We briefly describe these case studies.

The laser and dose control subsystem (LASER) provides light with an exact energy dose for exposure. The objective of the case study was to demonstrate the applicability of MBD for real-world ASML systems that show time-dependent (dynamic) behavior. In this case the diagnosis was performed on a simulation model of the real system. From this simulation model a diagnosis model was derived which was successfully tested with injected faults. This case showed how to use simulation models to validate

system	engineering discipline	LYDIA model size [LoC]	modeling time [days]	diagnosis time order of magnitude	
				current	*with MBD*
LASER	E, M, S, O, D	806	20	days	ms
EPIN	E, M	37	7	days	ms
POB	E, M, O	500	12	hours	s
ILS	E	82	8	minutes	ms
WS	E, M, S, H, D	2151	15	days	s

Table 13.1: Overview of modeling cases. E = electric, M = mechanic, S = software, O = optical, H = hydraulic, and D = dynamic.

diagnosis models and how these models can be used during design to analyze diagnostic quality. Furthermore it showed that the current LYDIA tooling which does not explicitly handle time can be used to model dynamic systems with work-arounds based on delayed signals and time conjunction as explained in Section 12.6.

EPIN is an electromechanical mechanism to lift wafers off the chuck. A system emergency occurs if certain safety constraints are violated. The case is a good example of how a relatively simple system consisting of three sensors, an actuator, and some safety monitoring logic, can still disrupt a diagnosis based on human reasoning. In one particular case it took two days to finally correctly identify the faulty sensor because of an initial mistake in the diagnostic reasoning. The objective of this case study was to demonstrate that LYDIA could work with real ASML board dump data and perform correct diagnoses. Besides meeting this objective, the case was extended to incorporate automatic model derivation from electronic layouts which are defined as Netlists. This automatic modeling step reduces modeling effort and decreases model maintenance as part of the model can be kept up-to-date to design automatically.

The Projection Optics Box (POB) is a system of adjustable mirrors. Common faults during integration and testing are wrongly connected mirror actuators and sign mistakes in the control software. The objective of the POB case study was to demonstrate the specific use of MBD during the integration phase and the combination of diagnosis models with existing (Matlab) models which greatly reduced the required modeling effort.

Scanners are equipped with an Interlock System (ILS) that detects whether the cover plates are in place to prevent humans from exposure to hazardous laser light. The objective of the ILS study was to demonstrate that it is possible to easily generate fault search trees from diagnosis models that guide engineers with fault finding in a system test. The derived trees are optimized by applying Information Theory and result in lower test costs in comparison to manually created test trees.

The wafer stage (WS) is used to move the wafer with nanometric precision during measurement and exposure. Similar to the EPIN mechanism safety logic monitors situations that may violate machine safety. The objective of the ongoing WS study is to demonstrate that MBD is useful for ASML on subsystem scale. To greatly reduce

modeling effort, part of the model is derived from VHDL implementations.

Note that in Table 13.1, model size includes comment lines and modeling time includes the time spent on case familiarization. The EPIN and ILS studies have relatively low model size and long modeling time. For the EPIN case this is explained by the fact that the systems around the actual EPIN system had to be modeled on a high abstraction level. This abstraction level was required by the limited scope of this study and the high level of realism. The ILS study required additional research on and comparison with the original fault trees used for this system. So far the WS study has produced the largest model (2151 LoC) with relatively short modeling time (15 days). This is explained by a large degree of similarity between the different components.

In half of the cases the diagnosis time had to be estimated based on earlier cases and simulation experiments. Based on these estimates and actual diagnosis times we find that the investment in modeling time yields significant speed-ups in diagnosis time.

13.4 Evaluation

From our understanding of the current way of working and the results of our cases studies we conclude the following. Augmenting or replacing experience-based diagnosis with MBD offers the following advantages:

- Volatility is resolved. Once models have been developed they can be used forever. Maintenance of the models still requires expertise.
- Unforeseen or unknown faults are covered because models capture nominal behavior and *any* deviating behavior is diagnosed as faulty. This improves the diagnostic quality, while known fault modes can be explicitly added to the model. It is recommended modeling practice to always incorporate an undefined (implicit) failure mode that covers *any* observed symptoms.
- The MBD characteristics that resolve volatility also resolve scarcity. Once a model has been created the expertise it captures can be made available throughout the organization.
- Depending on the model quality tunnel vision is no longer a problem. In the case of wrong diagnosis, i.e., the symptom is not explained correctly by the model, the model needs to be adapted. A proper implementation of MBD needs to allow models to be maintained and to evolve throughout the product life cycle.
- MBD does not suffer from data overload. In this respect it can be seen as an extremely advanced filter which distills root causes from symptoms fully automatically.

The other non-specific problems listed in Section 13.2 are not solved by MBD. It may improve on the overall diagnosis situation and lessen the effect of organizational, customer-related, cultural, and maintenance problems. For example, MBD may lead to

Life cycle phase	Effect on diagnostic quality	Diagnosis problems	MBD
Concept Analysis of current and future customer demands.	System availability requirements lead to diagnostic quality requirements.	NA	Abstract models predict diagnostic quality and the effect of failure modes.
Detailed design The design needs to fulfill the (driving) requirements.	Detailed design determines system diagnostic quality.	Diagnostic quality leads to conflicts with other design budgets.	Detailed models allow for analysis of diagnostic quality and testability
Prototype Demonstrate that the system fulfills the (driving) requirements.	First results on the actual diagnostic quality are obtained. Minor design improvements are possible.	Many problems need to be solved but expertise is readily available.	Feedback may be used for model evolution.
Production (T&I) Manufacture machines that satisfy customer requirements.	More feedback on testability and diagnostic quality.	Available diagnosis time and available expertise decrease.	See bullets in Section 13.4.
Operation Machines should operate according to customer requirements.	Major feedback on diagnostic quality.	Long down-times, high man-to-machine ratio, and scarce expertise.	See bullets in Section 13.4.
Refurbishment Fulfill new requirements and special needs.	Opportunity for improvements	Expertise may no longer be available.	Models contain system expertise, may require extension.

Table 13.2: Diagnosis and the impact of MBD during all product life cycle phases.

less ambiguous diagnoses leading to reduced workload (man-to-machine ratio) on the organization, improving customer trust, lead to unambiguous action plans, and overall require less unnecessary component replacements.

13.5 Diagnosis in the development phases

The scope of the Tangram project is limited to the integration and testing phase and realization phase of the development process. Diagnosis, however, affects all development phases. Hence, MBD has a much broader impact beyond the integration and testing phase and the operational phase, as discussed in the previous. Table 13.2 lists all life cycle phases together with their priorities, effects on diagnostic quality, diagnosis problems, and the benefits of MBD. From this table and the benefits discussion it is clear that modeling is key to the success of MBD. To ensure that MBD is deployed in each life cycle phase and to gain maximum benefit of the investment in modeling, it is important to start modeling early on in the system life cycle. This also prevents costly

reverse engineering of models from existing implementations.

13.6 Conclusions

We have discussed the benefits and costs of MBD in general terms and in the case of our industrial research partner ASML in particular. This was done by an analysis of the effects of MBD on diagnosis as it is currently performed and by discussing the results of specific case studies. MBD provides an efficient automated method of diagnosis that is well suited for rapidly evolving technology. In the Tangram project team the benefits are widely accepted. However, it is also understood that modeling cost and effort may pose a possible bottleneck.

We have made initial progress in reducing modeling costs by automatic model generation and modeling by engineers that are not experts at MBD. These preliminary results look promising; see also the MBD transfer project in Chapter 15. On a higher level, we see MBD as part of a paradigm shift to model-based systems engineering which may very well prove vital for the commercial success of highly innovative industries such as ASML. Part of the ongoing effort is also aimed at education and training for this model-based approach.

Chapter 14

A multidisciplinary integration and test infrastructure

Author: W.J.A. Denissen

14.1 Introduction

This chapter will discuss some use cases, the requirements, the design and the implementation of an integration and test infrastructure. The integration and test infrastructure is intended to support the Tangram project in its research on model-based integration (Chapters 7 and 8), model-based testing (Chapters 9 and 10), and test-based modeling (Chapter 11). It shall also support ASML engineers in executing their manually created tests.

The integration and test infrastructure surrounds a system under test by intercepting data and control flow over its interfaces to and from its environment. These interfaces can be of a very different nature depending on the discipline from which they originate. For software engineering these can be software interfaces like functions and their parameters and their interaction to the system (e.g., blocking or nonblocking calls, callback, or events). For electrical engineering the interface can be a set of control and status registers on some printed circuit board, or an optical link.

The integration and test infrastructure must provide to its users full control over the stimuli to and observations from the system under test over all these interfaces. The user can specify for each stimulus its order with respect to other stimuli and observations, the data that flows with it, and its trigger moment. For observations, the data and time of observation are captured. The integration and test infrastructure must allow programs, third party simulators or rapid prototype implementations, to be plugged-in. Each program can observe one or several interfaces and generate stimuli over one or

more possibly different interfaces to allow early integration tests. These interfaces can exist among plug-ins, or between plug-ins and the system under test.

The system under test (SUT) within Tangram is the ASML Twinscan machine or parts of it as shown in Figure 2.1 and described in Chapter 2. To obtain an impression of the complexity of the system under test we present some figures. It contains roughly 10 subsystems, 50 processors, 400 sensors, 500 actuators, 12.5M lines of code written in either C, Java, Python, or Matlab.

The remaining sections of this chapter are organized as follows. Two use cases for the integration and test infrastructure are presented in Sections 14.2 and 14.3. Section 14.4 presents the requirements and their rationale for the integration and test infrastructure. Sections 14.5 and 14.6 elaborate on the design and implementation of the integration and test infrastructure, respectively. The last Section 14.7 presents some conclusions and future work.

14.2 Use case: early integration

Early integration is a methodology to detect interface problems early to make test and integration less costly. Model-based early integration is elaborated in Chapters 7 and 8. Here, we discuss it from the point of view of the requirements it puts on the integration and test infrastructure.

In figure 14.1 the principles of early model-based integration are shown. It shows how a system model (depicted as a sheet of paper) is decomposed into two subsystem models, how each subsystem gets designed and realized in several versions, and finally how the subsystem realizations (depicted as cube shaped objects) get composed into the final system.

During decomposition the interfaces will be identified and named. When errors are made early, either in its design or in its realization on either side of an interface, then these errors will only be discovered after the composition of the subsystem realizations into the system realization. As a result, repairs become very costly or may result in workarounds that will then make maintenance very costly.

Within a single engineering discipline a lot of tools will help you to guard the interoperability over these interfaces. But what will happen when interfaces cross different disciplines? There is currently no single tool that guards the interoperability over these *interdisciplinary interfaces*. There is not even a single way to express, or specify, the interoperability crossing a interdisciplinary interface. So the question is how we can check the interoperability over an interdisciplinary interface when the system is evolving. Suppose that for both disciplines the versions of models or realizations are growing moderately over time. The combination of them can grow dramatically. To check all these combinations tool support is needed.

There are also non-technical issues during development and integration. The brick wall in Figure 14.1 symbolizes the behavior that starts to arise when responsibilities are distributed over several projects and/or different disciplines. Either side of the wall might feel that he is the owner of the interface and starts to define one. The other party

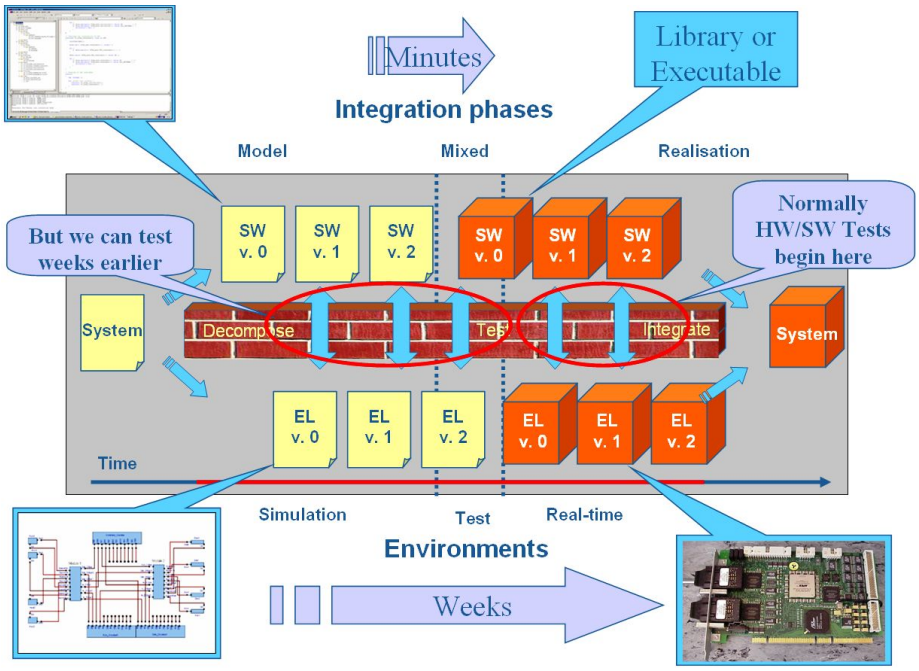


Figure 14.1: Early integration.

is hardly involved because they have not yet reached the point where they need to work with the interface. As a consequence they get much later confronted with an interface, which is defined from only one perspective. It might get even worse; both define an interface in the beginning, expressed in their own development environments and start to deviate from each other during development. Nobody guarantees that both interface descriptions are consistent. The brick wall can be used as the perfect excuse for neither project for feeling responsible for the interface. As a result the inter-disciplinary/inter-project interfaces between these subsystems become poorly managed. To tackle this, the integration and test infrastructure must support a new role: the interface manager. It must provide the interface manager with a means to define and guard the interfaces during the development process. Interfaces cannot be changed unilaterally or unnoticed.

When following the time line in Figure 14.1 different phases can be distinguished. Due to the fact that models and realizations reach completion at different moments in time there is no clear point when we cross to the design and realization phase. We therefore distinguish three integration phases, separated by vertical dotted lines.

In the *model integration phase* the integration and test infrastructure must provide a *simulation environment*, i.e., an execution platform in which several commercial of the

shelf and academic simulators can run executable models from different engineering disciplines in parallel and exchange information between each other. The data flow is typically small and of a simple type. Time can progress as simulated time and can be faster or slower than real-time.

In the *mixed integration phase* the integration and test infrastructure must provide a *test environment*, i.e., an execution platform in which simulators and prototypes can run in parallel with realizations. Realizations typically have much more dataflow, which is of a more complex type, and they have strict rules on the representation and availability in time. The integration and test infrastructure must be capable of converting information flowing from models to realizations and vice versa. When some of the dataflow has timing requirements, time must progress as real-time, and as a result a simulator must progress simulation time as fast as real-time. As a consequence, models can not be very detailed. When this is not sufficient a *prototype implementation* can be used that does communicate with the same dataflow as the realization does. In that case, more complex behavior can be expressed, because no dataflow conversions are needed. It, however, does come at the expense of a more detailed and elaborated prototype implementation. Because in this phase a complete system under test is not available, a *prototype environment* is needed to implement the missing parts of the system under test.

In the *realization integration phase* only realizations exist. The execution platform is called a *real-time environment*, and it must manage the interactions between realizations in real-time. It forms the environment in which the system under test runs, and the integration and test infrastructure must be able to interface with it.

14.3 Use case: test opportunities

In this section we consider the different phases in a V-model development process and we identify the opportunities for testing. These are depicted schematically in Figure 14.2 as two pointed arrows. Testing in this context is defined as detecting inconsistencies.

At each development level (system, sub-system, unit) a designer is involved that needs to come up with a design that fulfills the requirements for that level. He will typically use a design tool to build up a mental model of the structure and behavior of his design. By using a tool he is forced to express his design in a formal model. The mental model of the developer is kept aligned, synchronized, and consistent with this formal model. We will call these activities *mental* $\leftrightarrow$ *formal model testing*. This kind of testing is well supported by commercial tools, at least for disciplines other than software (more on this in Section 14.7), and is therefore not within the scope of Tangram.

When going down on the left side of the V-model a whole model is decomposed into part models. This is common practice within a single engineering discipline, and is the basic pattern to handle complexity. For instance, a system engineer can decompose his system budgets into subsystem budgets. A software engineer can decompose his

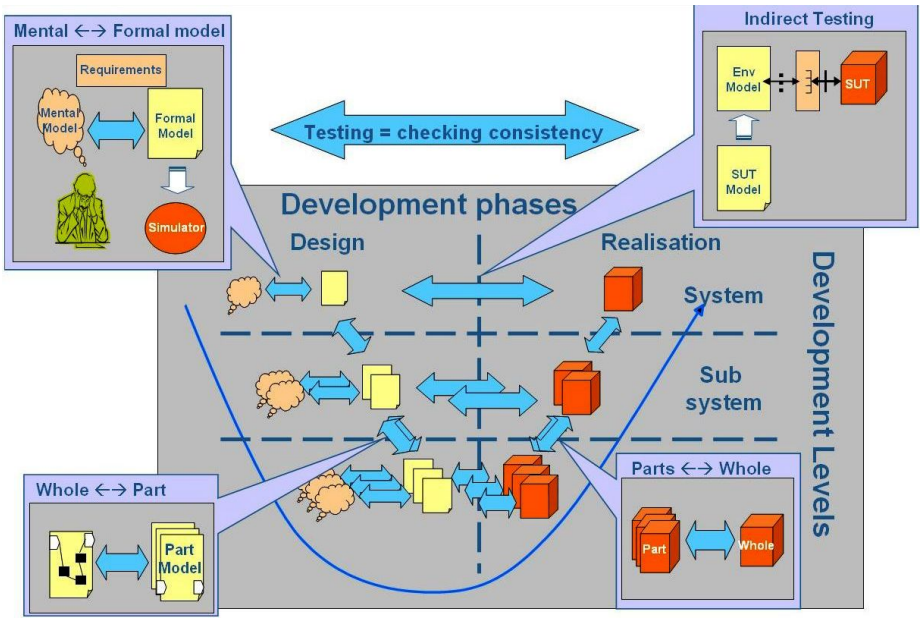


Figure 14.2: Test opportunities per level and phase.

software program into a set of subprograms. An electrical engineer can decompose his electrical model into a set of sub-models. The testing activity in *whole ↔ part model testing* consists of checking that the developer who will come up with the part models does not violate the requirements imposed on the whole model and vice versa. Once an inconsistency is detected either the whole or the part model needs to be modified such that they together are consistent again.

Part ↔ whole realization testing occurs the moment the different part realizations are assembled together. The kind of problems are typically related to resource conflicts or unknown interactions, for example, a memory footprint in software engineering, volume budgets in mechanical engineering, or the fan-in and fan-out of the active electrical parts in electrical engineering. Resources might be shared by different disciplines. For instance, a certain mechanical component might be blocking an optical light path, or the mechanical materials might outgas such that the optical lenses get polluted. Structural inconsistencies are typically detected while assembling. Behavioral inconsistencies are typically detected when executed the whole realization according to its use cases. Normally, most of the interactions are known by experience or from previous similar systems; these interactions are then also modeled in the design phase. Unforseen interactions are typically detected in this testing activity.

Model ↔ realization testing is normally known as conformance testing. Several

models can describe different aspects of a single realization. For each model the realization must conform to the model in their interfaces and their kind (software, electronics, physical) to their environment. Both the models and the realization interact with their environment over these interfaces. Typical usage of the system under test is modeled as a set of use cases, and can be seen as a 'good weather' test suite. A realization conforms to its model when the observations of the model and the realization are the same when the same set of use cases are applied to them. Testing a given aspect of a realization is typically done in an indirect way as depicted in the upper right part of Figure 14.2. Given a model of the system under test, an environment model, in the form of a test suite, is constructed against which the SUT is tested.

In *manual* model $\leftrightarrow$ realization testing the test designer manually derives the test suite from a model of the system under test. In *model-based* model $\leftrightarrow$ realization testing however, the test cases are automatically derived from the system under test model. A model-based test generator interprets the model of the system under test and derives on the fly test cases from it; see Chapters 9 and 10. The integration and test infrastructure executes the test suite, controls the system under test and observes its reactions. The integration and test infrastructure can judge, based on the observations, whether the system under test is reacting correctly or not.

14.4 Requirements and rationale

Now that we have described the use cases of the integration and test infrastructure, we can present the requirements and their rationale. For the integration and test infrastructure the following requirements apply.

- The same integration and test infrastructure must be used in each development phase and level, open for future extensions or unforeseen interactions between environments.

- The Tangram project is positioned at the right side of the V-model of ASML's development process as shown in Figure 14.2. This implies that all interfaces of the system under test have already been determined. For software interfaces, they are numerous (1000) and very volatile, they might change over night.

- For economical reasons there is only room to develop new test models with little detail and certainly no test models that contain almost as much (an order less) detail as the implementation. The models to test the system with, will therefore be as much as possible the existing models that where produced during the design phase of the development process.

- All realizations need to be accessed without any modification, to avoid pollution of the realization with test functionality, and interference with the development process.

- All newly designed parts of the integration and test infrastructure must be based on open standards, common-ware and commercial of the shelf tools, to reduce risks (proven technology, continuity) and development costs.

- The integration and test infrastructure must be applicable for other manufacturers of complex systems. Therefore, the ASML specific parts will be isolated as much as possible from the rest of the integration and test infrastructure.

The *simulation environment* must allow co-simulation of several models from different disciplines at the same time. The following aspects must be taken into consideration when designing the simulation environment.

- In Mental $\leftrightarrow$ Formal model testing, each discipline uses its own commercial of the shelf simulator, which has proven its usability within that discipline. The developers are familiar with these simulators and have invested considerable effort in building specific models. The simulation environment must therefore fully integrate and support these simulators. As a consequence these simulators must be open for extensions.

- In Whole $\leftrightarrow$ Part model testing, the whole model might run on a different simulator and/or platform than the part models. The simulation environment must therefore support a distributed simulation.

- The information describing the interfaces needs to be centralized and owned by an interface manager.

- To allow a modeler to stay within his own discipline, all interactions with the outside world go through a so called model connector. This can be a graphical/-textual representation that can be imported from a model library.

- Models containing simulation time need to be synchronized according to their semantics.

- The simulation environment must support addition of model animations that show, for instance, the state of the SUT at the proper design level.

The *prototype environment* must allow testing of a partly realized SUT. The following aspects must be taken into consideration when designing the prototype environment.

- The prototype environment must allow substitution of prototypes with realizations. For early integration, the developer must be capable to build prototypes in the most suitable (rapid prototype) programming language (e.g., C, Python, Java, and Matlab)

- The prototype environment must support different operating systems (e.g., Solaris, VxWorks, Linux, and Windows), and hardware platforms (e.g., Sun workstation, Powercore, PC).

The *test environment* allows a test designer to specify a test suite (a set of tests) that can be executed against a system under test. Each test can either pass or fail. The test environment must fulfill the following additional requirements:

- The test environment must allow automatic execution of tests.
- The test environment must have a notion of time to allow timed testing. Therefore the test environment must be able to control the actual moment of stimuli to the SUT and must also have access to time-stamped observations of the SUT's reactions. To test or diagnose the SUT in its real-time environment the test environment needs full control and observability over its interfaces.
- The test environment must be connected to the simulation environment to allow a partly simulated environment for the system under test while testing.
- The test environment must be connected to the prototype environment to allow a partly implemented system under test to be tested.
- The test environment must handle synchronous and asynchronous bi-directional interactions with the simulation environment, prototype environment and the system under test.

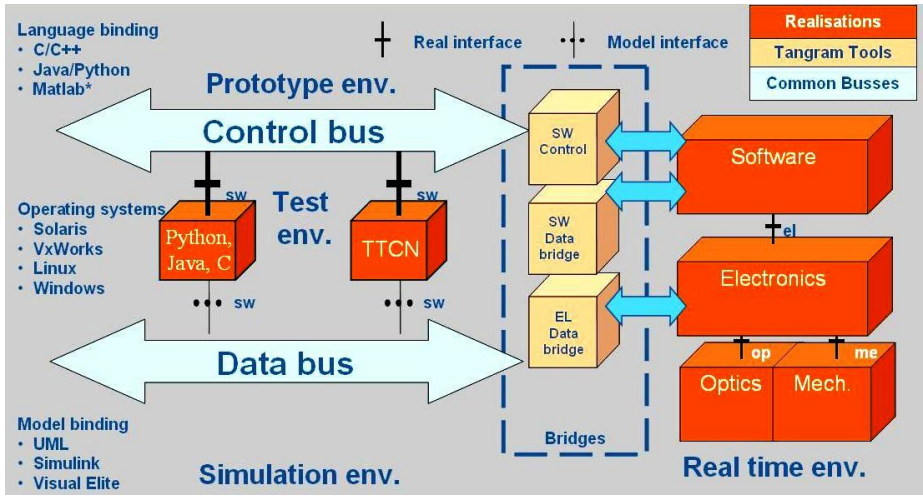


Figure 14.3: Integration and test infrastructure.

14.5 Design

Based on the previously described requirements a design of the integration and test infrastructure can be presented. Figure 14.3 shows the integration and test infrastructure. Four different environments can be identified: simulation, prototype, test, and real-time. For scalability reasons, the integration and test infrastructure is based on a bus topology. An open and standardized bus avoids vendor lock and assures interoperability between the participants. The prototype, test, and real-time environments are all concentrated around a standard *control* bus. This control bus is based on CORBA [92]. The rationale for selecting CORBA is:

- CORBA is based on decades of experience in driving reactive systems, is designed for and by software developers, and is an open OMG standard.
- CORBA hides all transport specific details, is based on the proxy pattern (i.e., allows uniform calling of services over different programming languages, operating systems, and communication hardware).
- Several tool vendors provide CORBA and its services. CORBA has a demanding user community: Defense, Aerospace, and Manufacturing companies.

Moreover, the simulation, test, and real-time environments use a standard *data* bus. This data bus is based on the CORBA data distribution service (DDS) [91]. The rationale for selecting DDS is:

- DDS is based on decades of experience in driving real-time reactive systems, is designed for and by software developers, and is an OMG standard.
- DDS is based on the publish/subscribe pattern and provides a simple API with lots of quality of service configurations and hides all transport specific details.
- Several tool vendors provide DDS tools and a vast demanding user community exists around DDS: Defense and Aerospace companies.
- The numerous ‘quality of service’ settings of DDS allow, amongst others, easy implementation of multicasting, super- or sub-sampling, and time stamping of value streams.

For the *simulation environment* at least two commercial of the shelf simulators need to be supported to start with: Matlab/Simulink and Labview The rationale for selecting these simulators is:

- Simulink is widely used within ASML to simulate physical behavior of (parts of) the system under test or its environment.
- Labview has been selected to demonstrate that two different simulators can be combined to co-simulate over a DDS data bus.

For the *prototype environment* a rapid prototyping language Python [104] has been selected to start with. Other CORBA supported languages like C, C++, or Java could also have been selected. The selection is based on the following rationale:

- Python is a extremely flexible (less type checking), object oriented, small, clean and easy to learn programming language.
- Python has powerful build-in container types like: lists, tuples and dictionaries.
- Python is modular, supports reflection, and can be embedded within C/C++ or embed C/C++ library functionality directly.
- Python is accompanied with a vast array of modules.
- The development of Python as a language and its tools is an open source project. A vast and enthusiastic user community exists that can provide feedback and bug-fixes quickly and easily.
- Serious companies are relying on the functionality and flexibility of python, like Google for instance.

For the *test environment* the standard test language TTCN3 (Test and Test Control Notation [42]) has been selected for the test designer to write his test suite. The selection is based on the following rationale:

- TTCN3 is based on decades of experience in testing reactive systems, is designed for and by test developers, and is an open ETSI standard.
- TTCN3 abstracts away all SUT specific details.
- TTCN3 allows uniform testing over different real interfaces.
- Robust and mature IDEs exist that help the test engineer in writing, debugging and managing his/her test specifications.
- Several Tool vendors provide TTCN3 tools and a vast user community exists around TTCN3: Automotive, Telecom companies.

For the *bridges* at least a software control bridge and an electronic control bridge need to be prototyped to demonstrate interdisciplinary testing. The selection for the bridges is based on the following rationale:

- For bridging ASML proprietary software interfaces, code generation techniques are used that will produce glue code. It will convert proprietary interface descriptions into standard interface descriptions. Control and data flow are redirected by generating redirectors in the form of shared object libraries with an equivalent software interface.

- For bridging a *standard* electronic interface, the commercial of the shelf tool Labview has been selected because of its vast library of supported electronic interfaces.

- For bridging a *proprietary* electronic interface, an optical link communication protocol that is widely used within ASML subsystems (High Speed Serial Link, or HSSL) has been selected, and has been implemented as a classical operating system device driver. That way the device driver can be easily used and combined with other data links. It allows the test engineer to redirect electronic/optical information between sub-systems to a test environment, thereby allowing testing of subsystems in isolation, without having to fall back on the expensive and heavily claimed prototype machines.

14.6 Implementation, pilot projects, and transfer

Based on the design as described in the previous section, we have implemented the following parts of the integration and test infrastructure as shown in Figure 14.3: Software control bridge, Electronic data bridge, TTCN to CORBA bridge, TTCN to DDS bridge, Simulink to DDS bridge/connector, and Labview to DDS bridge/connector. This simple set of bridges provides already great flexibility to combine existing tools to test the system under test, or to combine models with models. Concrete ASML pilot projects helped us assessing the applicability, usability, flexibility, and robustness of our integration and test infrastructure. Due to the restricted resources the pilot projects could not cover all testing opportunities, for each development phase and level, as we would like to do. Within the HSSL pilot project we investigated how to bring electronic simulations close to the system under test when the timing requirements vary from non real-time to soft real-time to hard real-time. In another pilot project we provided a proof of concept for testing the hardware software interface, where the interface is described as memory mapped I/O.

ASML has hosted the Tangram researchers to allow short communication lines to the problem owners. This not only resulted in a clear understanding of the problem but also a clear understanding of the challenges we might face when introducing our proof of concept tools to the users in the field. A proof of concept is fine to convince some early adapters, but the introduction of a new tool that needs to be used by several hundreds of people working within stringent delivery deadlines is something different. They demand full support, documentation, product quality tools, training, smooth migration and so on.

To fulfill these user expectations a separate ASML Tangram tranfer project (see Chapter 15) was defined. The software control bridge and the electronic data bridge where selected to be leveraged to product quality tools. Within six months a complete redesign of the software control bridge took place to minimize future maintenance, missing features where implemented, and a complete test suite and test system were built to test the quality of the delivered tools. Performance measurements of the soft-

ware control bridge product indicated an even better performance than the system under test's communication infrastructure. The software control bridge might therefore be used to smoothly migrate from a proprietary communication infrastructure to an open and standardized communication infrastructure.

14.7 Conclusions and future work

The test methodology as implemented in the prototype integration and test infrastructure has proven to be usable within ASML. It does not interfere with ASML's tightly scheduled system development as the system under test does not need to be modified to get it tested. The integration and test infrastructure opens up new opportunities to test the system under test. For instance, intercepting required software interfaces allows testing against 'bad weather' conditions or using a partly build system under test. The ability to plugin separate software prototypes and simulators allows ASML to reuse them for other products in the same product family. The fact that standard busses are used allows ASML to remotely test their machines, either in their prototype or production buildings, but also at the customer's site if needed. Discussions on how to integrate this testing methodology in ASML's current way of working have just started, and this demonstrates their confidence in the methodology. The Tangram transfer project has shown that prototypes can be turned into a product that meets ASML's quality requirements.

Future work therefore includes extending the current integration and test infrastructure by adding more simulators (SystemC, Visual Elite), design management tools (interfaces, requirements, and versions), diagnostic tools (like model animators and code instrumentation), and test tools (test case generators and extensions for timed testing).

Two parts of the integration and test infrastructure that really need more attention, and therefore have higher priority, are is mental $\leftrightarrow$ formal model testing, and whole $\leftrightarrow$ part testing, especially for software. Although they were out of scope for the current project (see Section 14.3), they carry most of the currently perceived problems when developing and testing systems, like structured and controlled evolution, proper versioning, and low maintenance. Proper tool support for both of the above mentioned kinds of testing will also greatly reduce the above problems.

Chapter 15

The Tangram transfer projects: from research to practice

Authors: A.P.G.C. van Dongen, G.J. Tretmans

15.1 Introduction

The research projects that are coordinated by the Embedded Systems Institute, such as Tangram, are characterized as applied research and development. They are carried out in an *industry-as-laboratory* setting; see Chapters 1 and 2. For Tangram, this laboratory setting is provided by the main industrial partner ASML. The industry-as-laboratory setting provides the realistic industrial environment in which new research ideas and theories, mostly coming from the academic partners, can be validated and tested. The industrial relevance of new methods and techniques is demonstrated by a *proof-of-concept* showing that the principles work, and could be deployed in industry. The actual transfer of knowledge, including training, industrial strength tools, scaling to industrially sized problems, embedding in the industrial process, et cetera, are not part of the research project. Separate transfer projects were initiated for those research areas that demonstrated their industrial maturity by means of a successful proof-of-concept. This chapter discusses transfer projects in general, and those within Tangram in particular. The emphasis is on the managerial aspects of the transfer projects.

15.2 Phases of industrial evidence

Different levels of proof-of-concept, and different phases in demonstrating evidence and transferring knowledge can be recognized. The Embedded Systems Institute distinguishes between six phases, which range from pure academic research to full oper-

ational use in industry. In the successive phases, the scale and level of reality of the proof-of-concept case studies increases, while the involvement of the industrial partner grows from none to complete involvement in the operational phase; see Figure 15.1.

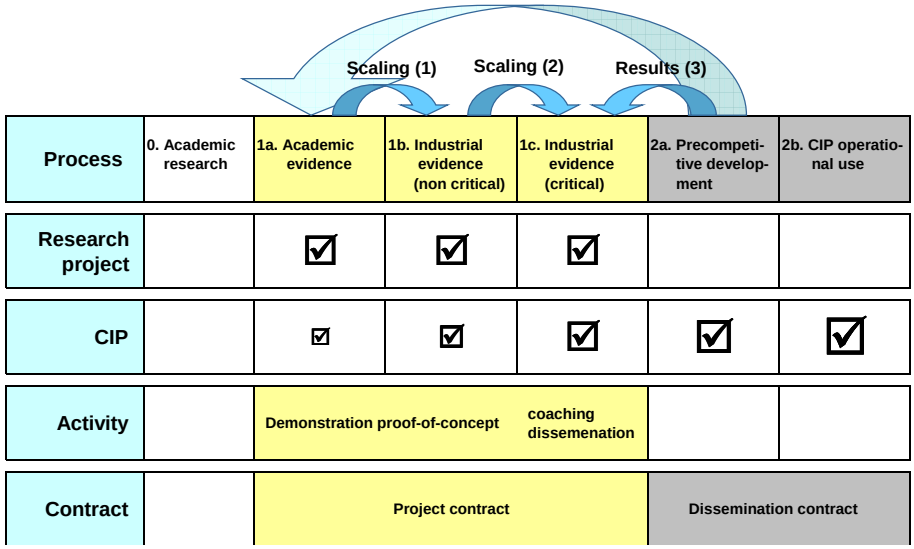


Figure 15.1: The transfer process.

Phase 0: Academic research. Universities and research institutes perform research in a more or less autonomous setting. This research may be triggered by industrial problems, but more often it is just curiosity driven academic research. Industry-as-laboratory projects of the Embedded Systems Institute are not directly involved here, nor are transfer projects. This research may lead to future industrial relevance. Related to Tangram, such research may involve the definition of new modeling languages and their formal semantics, or general complexity issues of planning algorithms.

Phase 1a: Academic evidence. Academic evidence refers to a proof-of-concept of newly developed methods, techniques and tools in an academic environment. Emphasis in this phase is on making the theory work, and showing that it may potentially solve an industrial problem. The methods and tools are applied to an artificial academic problem, or to a simplified industrial problem, in a well-defined and controlled environment. Aspects like scalability of the method, or usability of the tools do not play a role, yet. Typically, some evidence of this

kind is a starting point for an industry-as-laboratory project like Tangram. An example is the Tangram area of model-based testing, where experiments were performed in the university laboratory on testing relatively simple smart-card applications.

Phase 1b: Industrial evidence, non-critical. A realistic problem of the industrial partner with limited size and complexity is the starting point for showing industrial evidence. Project members and researchers are involved; the industrial partner may actively cooperate but his main task is to state the problem, provide a case for this problem, and provide the necessary domain knowledge so that the project members get a clear view on the problem and the solution area. The case study should show the proof-of-concept, and demonstrate the feasibility and benefits, when possible but not necessarily quantitatively, of the newly developed techniques and tools in a real industrial case. This case should be non-critical, in the sense that the daily business of the industrial partner may in no way depend on it. In this phase the case may concern an old problem for which a solution has already been developed with other means, e.g., applying new testing techniques to an already tested and released product to investigate the benefits of the new technique.

Phase 1c. Industrial evidence, critical. The newly developed techniques and tools are used in pilot projects in the real industrial context, where the outcome does matter. The focus is on scalability, embedding in, and impact on existing processes, and demonstrating quantitative evidence of their usefulness. Issues and aspects that do not directly relate to the main functionality of the new techniques and tools, such as usability, performance, reliability, availability of documentation and manuals, help desk, and training, are getting more important. Also the importance of knowledge consolidation and transfer increases. Since many of these issues are important for, and in addition specific to the industrial partner, whereas they are less interesting from a research perspective, the involvement of the industrial partner increases.

To organize the aspects mentioned above, a *transfer project* is initiated for each successful research area, which has clearly shown industrial evidence of benefits. The aim of a transfer project is, as the name suggests, to transfer knowledge, methodology, and tools from the research project to the industrial partner. Since this goes beyond the proof-of-concept goal of the research project, and since the major part of the manpower for this activity is provided by the industrial partner, transfer projects are decoupled from the research project.

Phase 2: Pre-competitive development. The main target of this phase is to prepare the methodology, techniques and tools so that they can be institutionalized within the industrial partner. This involves a seamless continuation of the activities of the transfer project, but with much less involvement of the research-project members; they may give support for specific requests, but they are generally not

involved anymore. The pilot projects are gradually taken over by real users, adapting and deploying the methodology and supporting tools in their daily development activities. The activities during this phase include refactoring of tools, documentation, user training, and configuration management, making the methodology and supporting tooling ready to be rolled out in the whole organization.

Phase 3: Operational use. The full roll out of the methodology, techniques and tooling into the organization of the industrial partner takes place in this phase. Parallel to the daily usage of the techniques and tools, the support aspects like training, knowledge consolidation, tool maintenance and support, and configuration management are institutionalized within the industrial partner. For the research project there is no role anymore in this phase.

Typically, and ideally, a research topic passes through these six phases from academic research to operational use in industry. But, of course, there are several feedback loops. The results in any phase can trigger new research questions, and also new problems may trigger new research subjects. And yet, many research ideas never make it to operational use.

15.3 The Tangram transfer projects

A transfer project is initiated to transfer the knowledge, methodology, and tools from the research project, i.c. Tangram, to the industrial partner, i.c. ASML. The activities of a transfer project focus on those aspects that go beyond the original project goal of showing proof-of-concept, such as industrial tool development, tool configuration management, training, embedding in the industrial organization, et cetera. A transfer project is only started for research areas that successfully showed evidence of industrial applicability. The outcomes of a transfer project are methods, techniques, and supporting tools, which are ready for institutionalization within the industrial organization.

The work in Tangram was structured into five research areas, see Section 1.4: integration and test planning, model-based integration, model-based testing, model-based diagnosis, and integration and test infrastructure. Three of these areas delivered a sufficiently mature proof-of-concept to start a transfer project:

- integration and test planning,
- model-based diagnosis, and
- integration and test infrastructure.

Moreover, the industrial partner ASML did recognize the value and benefits of these methodologies, and was willing to make investments in these areas.

15.4 Transfer project activities

Within the Tangram transfer projects the main activities were transfer of knowledge about the methodology, techniques and supporting tools to ASML, and the transfer of the tools themselves.

Since the research project Tangram aimed at a proof-of-concept, the main focus for tool development was on the technical and functional aspects of these tools to support the methodologies. Other aspects, such as scalability, usability, performance, reliability, documentation, and manuals, sometimes referred to as non-functional properties, were not really taken into account. Because these non-functional requirements are important for successful industrial usage, the transfer of tools included implementation activities aimed at adaptations and extensions to cope with these non-functional aspects. Below we elaborate on how these tool adaptations and extensions were organized.

The first activity of transfer of knowledge was mainly accomplished by means of training, workshops, and coaching. Moreover, pilot projects using the methodology and tools were started to have immediate feedback from the users to the transfer project team.

The transfer project teams were mainly staffed by ASML (80%), and supported by the Tangram team (20%). The transfer projects lasted between 8 months and 1 year.

Transfer of tools: incremental delivery

An important aspect of the transfer projects was to adapt and extend the prototype Tangram tools in order to make them applicable in an industrial context, and to embed them in the ASML environment. In contrast to the rather ad-hoc, research driven tool development in Tangram, these tool adaptations and extensions were strictly managed.

An incremental development approach with intermediate deliveries to the pilot projects using the tools was selected for this purpose. The objective of the incremental deliveries was to obtain immediate feedback, to deal with the lack of detailed requirements, and to cope with the limited initial knowledge about how the tools should be embedded within the ASML processes. In this way, the methodology and supporting tools are gradually adapted to the ASML context, so that consolidation is easier and more robust. For this approach the selection of appropriate pilots is essential. The application of the new methodology and tools should really have an impact on these pilots, and the pilots should be able to provide immediate and useful feedback.

Transfer of tools: requirements management

The incremental development and delivery approach imposed the need for strict, yet flexible requirements management. An initial requirements inventory was made, and based on the feedback from the pilot projects requirements were added, changed, or removed. A release plan, documenting which requirements would be implemented for which delivery, was continuously adapted and updated. This release plan was strongly

coupled to the needs of the pilot projects. For this purpose the requirements were classified into four classes according to their expected cost/benefit ratio: (1) high or low importance for the customer (the pilot project), and (2) high or low effort for the implementation (the transfer project team); see Figure 15.2.

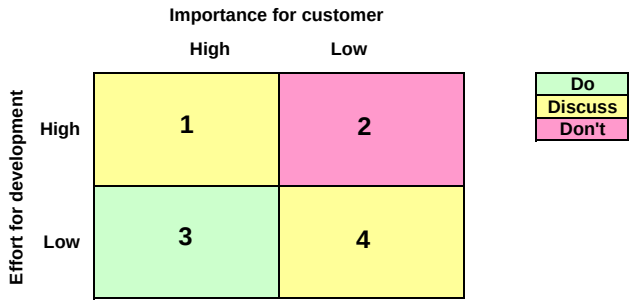


Figure 15.2: Requirements: importance and effort.

According to this classification the release planning was made. Requirements with high importance and low effort are, of course, immediately implemented, while low importance and high effort requirements are not considered at all. High importance and high effort requires detailed discussion and, when deemed necessary, planning, and low importance and low effort requirements are implemented when the implementers have some spare time left.

This requirements capturing, classification, and planning is repeated for every incremental delivery taking into account the experiences and feedback from the pilot projects.

15.5 Transfer project challenges

The transfer projects encountered a couple of challenges. Compared with the research phase of Tangram, there were more non-technical and management issues. And whereas technical challenges can usually be solved by clever engineers, sometimes supported by Tangram project members, the non-technical problems in general have less clear solutions. Since these non-technical problems are mostly of a generic nature, independent of the topic of Tangram being transferred, we briefly discuss some of them here.

Selection of appropriate pilot projects The selection of pilot projects is an activity that must be done with utmost care. On the one hand, the application of the new methodologies and tools in the pilot projects should be feasible and doable. On the

other hand, the pilots should be chosen such that they lead to observable and quantifiable results and benefits, not only at the end of the transfer project, but continuously at all intermediate steps in order to create continuous interest, support, and commitment at all levels of the ASML organization.

Prototype tool quality The technical baseline for the transfer of a tool is the prototype tool developed for the proof-of-concept. As explained above in Section 15.4, adaptations and extensions, mainly concerning non-functional aspects, are necessary to make such a tool applicable in an industrial environment. To obtain a realistic planning for these adaptations and extensions a precise evaluation of the current quality of the tool must be made first. This includes the status of the technical documentation and manuals. Realistic planning is important to keep commitment and support for the transfer project. The right adaptations and extensions should be incrementally delivered on the right moment to be of added value for the pilot projects. Commitment may be jeopardized by negative experiences with, or feelings about the tools. Such negative sentiments should as much as possible be kept under control by timely and consistent support solving the issues involved.

People Having appropriate staffing of the transfer projects, both in number and in expertise, is important. Because the methodology and tools emanate from a research environment, the transfer project members should be able to communicate effectively with the research team to cope with this situation. The support from the Tangram research team members is available but limited, because of the ongoing research activities, and because transfer activities that go beyond the proof-of-concept are not in the scope of the research project. With respect to the number of people in the transfer projects, there will be a constant competition for scarce resources, e.g., when there is pressure because of critical deliveries elsewhere in the organization. These are not under control of the transfer project. To compete with such projects, it is even more important to have managerial commitment.

Commitment Creating and maintaining commitment for a transfer project within all levels of the ASML organization is one of the most important issues. Initial commitment is created by results of the proof-of-concept cases. During the execution of the transfer project, the commitment should be extended and increased, mainly by continuous and observable progress and benefits in the pilots. These pilot projects let people directly experience the advantages in their daily working activities, thus creating enthusiasm and bottom up commitment.

Early involvement of managers in the project communication and outcome evaluation is important to create top down commitment. This includes realistic expectation management. Commitment is also very important for the later consolidation and institutionalization of the methodology and tools within ASML.

Tool support for the pilot projects The tool support for the pilot projects is performed by the transfer project team itself. This is a perfect way to keep in touch with the practical use of the tool, and have continuous feedback for detecting and fixing problems and upgrading with new features. It is important to keep the balance between direct support for the pilots, and working on the planned tool adaptations and extensions, in particular when the number of pilots grows. This risk is mitigated by creating a so-called 'buddy' system in which the more experienced pilot users guide the novice users with initial training and first line support. This approach also facilitates the gradual institutionalization of the methodology and tools.

15.6 The Tangram transfer project results

Transfer projects were initiated for three research areas of Tangram, see Section 15.3: integration and test planning, model-based diagnosis, and integration and test infrastructure. We briefly describe their outcomes.

During the transfer project, the tool for automatic integration and test planning LONETTE was updated with features supporting its efficient and effective use, with emphasis on usability aspects. By having ASML developers perform this task, also knowledge about the internal implementation of LONETTE was indirectly transferred. An extensive training program including a workshop was developed and held for ASML engineers so that they have sufficient knowledge to use the method and its tool LONETTE. Currently, the method and LONETTE are used in planning the weekly software validation test, which is executed every week to check the changes made to the software baseline.

For model-based diagnosis, a training module was developed, including knowledge and experience about model-based diagnosis in general, and about the LYDIA system and language in particular. Workshops based on this training module were held, in which ASML engineers were taught how to develop diagnosis models in LYDIA, and how to use the tool for fast diagnosis. As part of the transfer project, a VHDL to LYDIA translator was developed, which enables automatic conversion of a VHDL description of safety logic into a LYDIA model. Moreover, the LYDIA system was extended with several features that increased its performance and usability. A pilot project where LYDIA was used for diagnosis at a customer site showed very good results.

The know-how about the integration and test infrastructure has been transferred by having ASML engineers doing a re-factoring of the proof-of-concept tool, supported by a Tangram member. Emphasis in this re-factoring was on increasing the robustness of the tools. Next to that, a user training was developed enabling ASML developers to effectively use the integration and test infrastructure in their daily working activities. Currently, it is mainly used as a test execution environment for manually developed tests.

For all three transfer projects, ASML has taken over the methodologies, tools, and accompanying training materials. Further consolidation and institutionalization is now

the responsibility of ASML, and is, among others, done by placing it on the respective road maps for future developments.

15.7 Concluding remarks

The goal of Tangram was to develop methods and techniques to reduce lead time, improve quality, and decrease costs of integration and testing of high-tech multidisciplinary systems; see Chapter 1. The outcomes are a couple of methods, techniques and tools at different levels of maturity. These outcomes are delivered as a proof-of-concept: a demonstration that the developed solution works and solves a problem of the industrial partner, i.c. ASML. Transfer of the proposed solution to the industry, including aspects like training, embedding in the organization, long term support, et cetera, are explicitly not considered within such a research project. Starting with the results of Tangram, three transfer projects were initiated to transform the proof-of-concept results into industrially applicable results, and to transfer them to ASML.

This chapter showed the different phases of proof-of-concept and industrial evidence of research project results, and why a transfer project is necessary, even if Tangram is called an 'applied' research project. The activities in such a transfer project were discussed, with emphasis on tool adaptations and extensions, and the necessity to organize these with incremental delivery of tool functionality and strict requirements management. Also a couple of issues and challenges for the organization of such transfer projects were discussed, of which creating and maintaining commitment within the organization is considered the most important one.

The three Tangram transfer projects finished successfully. ASML has adopted their results, and will now have to work on further consolidation, institutionalization, and roll-out in the organization. Another challenge is the further proliferation of these results outside ASML, and the question to what extent these results can be copied to other organizations. Moreover, it can be expected that, one day, the use of the Tangram methodologies and tools will lead to new problems and to new research questions in integration and testing, which then again will trigger new research projects. With respect to both, the further proliferation of Tangram results, and the identification of new research questions and the initiation of new projects, there is an important role for the Embedded Systems Institute.

Appendix A

Tangram publications

Scientific publications and Ph.D. theses

1. J.C.M. Baeten, D.A. van Beek, P.J.L. Cuijpers, M.A. Reniers, J.E. Rooda, R.R.H. Schiffelers, and R.J.M. Theunissen. *Model-Based Engineering of Embedded Systems Using the Hybrid Process Algebra* χ.
   In LIX Colloquium on Emerging Trends in Concurrency Theory, Paris, France, November 2006. To appear in Electronic Notes in Theoretical Computer Science.

2. J.C.M. Baeten, D.A. van Beek, and J.E. Rooda. *Process Algebra for Dynamic System Modeling*.
   In P.A. Fishwick, editor, CRC Handbook of Dynamic System Modeling. Taylor and Francis Group LLC, 2007. To appear.

3. H. Bohnenkamp and A. Belinfante. *Timed Testing with* TORX.
   In J. Fitzgerald, I.J. Hayes, and A. Tarlecki, editors, FM 2005: Formal Methods: Int. Symposium of Formal Methods Europe, number 3582 in Lecture Notes in Computer Science, pages 173–188. Springer-Verlag, 2005.

4. H. Bohnenkamp and A. Belinfante. *Timed Testing with* TORX.
   In W. Dulz, editor, ITG FA 6.2 Workshop on Model-Based Testing. VDE-Verlag, 2006. *Reworked version of publication no. 3*.

5. R. Boumen. *Integration and Test Plans for Complex Manufacturing Systems*.
   PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2007.

6. R. Boumen, I.S.M. de Jong, J.M.G. Mestrom, J.M. van de Mortel-Fronczak, and J.E. Rooda. *Integration Sequencing in Complex Manufacturing Systems*.
   SE Report 2006-02, Eindhoven University of Technology, Eindhoven, The Nether-

lands, 2006.

7. R. Boumen, I.S.M. de Jong, J.M. van de Mortel-Fronczak, and J.E. Rooda. *Test Time Reduction by Optimal Test Sequencing*.
In Proceedings of INCOSE 2006 – 16^{th} Int. Symposium on Systems Engineering, Orlando, FL, USA, July 9-13 2006.

8. R. Boumen, I.S.M. de Jong, J.M. van de Mortel-Fronczak, and J.E. Rooda. *Optimal Integration and Test Planning Applied to Lithographic Systems*.
In Proceedings of INCOSE 2007 – 17^{th} Int. Symposium on Systems Engineering, San Diego, CA, USA, June 24-28 2007.

9. R. Boumen, I.S.M. de Jong, J.M. van de Mortel-Fronczak, and J.E. Rooda. *Optimal Integration and Test Plans for Software Releases of Lithographic Systems*.
In Proceedings of the 5^{th} Annual Conference on Systems Engineering Research — CSER, 2007, Hoboken, NJ, USA, March 14-16 2007. Stevens Institute of Technology.

10. R. Boumen, I.S.M. de Jong, J.W.H. Vermunt, J.M. van de Mortel-Fronczak, and J.E. Rooda. *Test Sequencing in Complex Manufacturing Systems*.
IEEE Transactions on Systems, Man and Cybernetics – Part A: Systems and Humans, 2006. Accepted for publication.

11. N.C.W.M. Braspenning, E.M. Bortnik, J.M. van de Mortel-Fronczak, and J.E. Rooda. *Model-Based System Analysis using Chi and Uppaal: An Industrial Case Study*.
Computers in Industry, 2007. Accepted for publication.

12. N.C.W.M. Braspenning, J.M. van de Mortel-Fronczak, and J.E. Rooda. *A Model-Based Integration and Testing Method to Reduce System Development Effort*.
In Proceedings of the Second Workshop on Model Based Testing – MBT 2006, volume 164/4 of Electronic Notes in Theoretical Computer Science – ENTCS, pages 13–28. Elsevier, 2006.

13. N.C.W.M. Braspenning, J.M. van de Mortel-Fronczak, and J.E. Rooda. *Analysis and Implementation of Infrastructure for Model-Based Integration and Testing*.
In Proceedings of the 5^{th} Annual Conference on Systems Engineering Research – CSER 2007, Hoboken, NJ, USA, March 14-16 2007. Stevens Institute of Technology.

14. N.C.W.M. Braspenning, D. Kostić, J.M. van de Mortel-Fronczak, and J.E. Rooda. *Model-Based Support for Integration and Testing of an Industrial System*.
In Proceedings of the European Systems Engineering Conference – EuSEC 2006, Edinburgh, UK, September 18-20 2006.

15. N.C.W.M. Braspenning, D.O. van der Ploeg, J.M. van de Mortel-Fronczak, and

J.E. Rooda. *Model-Based Techniques for Intelligent Integration and Testing in Industry*.
In Proceedings of INCOSE 2007 – 17^{th} Int. Symposium on Systems Engineering, San Diego, CA, USA, June 24-28 2007.

16. W.J.A. Denissen. *A Multidisciplinary Model-Based Test and Integration Infrastructure*.
In Proceedings of the 2006 IEEE Int. Symposium on Intelligent Control, pages 1916–1921. IEEE, October 2006.

17. A. Feldman, J. Pietersma, and A. van Gemund. *A Multi-Valued SAT-Based Algorithm for Faster Model-Based Diagnosis*.
In C. Alonso Gonzáles, T. Escobert, and B. Pulido, editors, Seventeenth Int. Workshop on Principles of Diagnosis – DX-06, pages 93–100, Peñaranda de Duero, Burgos, Spain, June 2006.

18. A. Feldman, J. Pietersma, and A. van Gemund. *All Roads Lead to Fault Diagnosis: Model-Based Reasoning with* LYDIA.
In P.-Y. Schobbens, W. Vanhoof, and G. Schwanen, editors, Proceedings of the Eighteenth Belgium-Netherlands Conference on Artificial Intelligence – BNAIC-06, pages 123–131, Namur, Belgium, October 2006.

19. L. Frantzen and J. Tretmans. *Model-Based Testing of Environmental Conformance of Components*.
In F.S de Boer and M. Bosangue, editors, Formal Methods of Components and Objects – FMCO 2006, volume 4709 of Lecture Notes in Computer Science. Springer-Verlag, 2007.

20. L. Frantzen, J. Tretmans, and T. Willemse. *Test Generation Based on Symbolic Specifications*.
In J. Grabowski and B. Nielsen, editors, Formal Approaches to Software Testing – FATES 2004, volume 3395 of Lecture Notes in Computer Science, pages 1–15. Springer-Verlag, 2005.

21. L. Frantzen, J. Tretmans, and T.A.C. Willemse. *A Symbolic Framework for Model-Based Testing*.
In K. Havelund, M. Núñez, G. Roşu, and B. Wolff, editors, Formal Approaches to Software Testing and Runtime Verification – FATES/RV'06, volume 4262 of Lecture Notes in Computer Science, pages 40–54. Springer-Verlag, 2006.

22. M. Gromov and T.A.C. Willemse. *Testing and Model-Checking Techniques for Diagnosis*.
In A. Petrenko, M. Veanes, J. Tretmans, and W. Grieskamp, editors, Testing of Software and Communicating Systems – 19^{th} IFIP TC 6/WG 6.1 Int. Conference, TestCom 2007, 7^{th} Int. Workshop, FATES 2007, volume 4581 of Lecture Notes in Computer Science. Springer-Verlag, 2007.

23. I.S.M. de Jong, R. Boumen, J.M. van de Mortel-Fronczak, and J.E. Rooda. *Integration and Test Strategies for Semiconductor Manufacturing Equipment*.
In Proceedings of INCOSE 2006 – 16^{th} Int. Symposium on Systems Engineering, Orlando, FL, USA, July 9-13 2006.

24. I.S.M. de Jong, R. Boumen, J.M. van de Mortel-Fronczak, and J.E. Rooda. *An Overview of Integration and Test Plans in Organizations with Different Business Drivers*.
In Proceedings of the 5^{th} Annual Conference on Systems Engineering Research – CSER 2007, Hoboken, NJ, USA, March 14-16 2007. Stevens Institute of Technology.

25. I.S.M. de Jong, R. Boumen, J.M. van de Mortel-Fronczak, and J.E. Rooda. *Software Reliability Qualification for Semi-Conductor Manufacturing Systems*.
In the 18^{th} Annual IEEE/SEMI Advanced Semiconductor Manufacturing Conference – ASMC 2007, Stresa, Italy, June 11-12 2007. Semiconductor Equipment and Materials International – SEMI, San Jose, CA, USA.

26. M. van Osch. *Automated Model-Based Testing of χ Simulation Models with* TorX.
In R. Reussner, J. Mayer, J.A. Stafford, S. Overhage, S. Becker, and P.J. Schroeder, editors, Quality of Software Architectures and Software Quality, number 3712 in Lecture Notes in Computer Science, pages 227–241. Springer-Verlag, 2005.

27. M. van Osch. *Hybrid Input-Output Conformance and Test Generation*.
In K. Havelund, M. Núñez, G. Roşu, and B. Wolff, editors, Formal Approaches to Software Testing and Runtime Verification – FATES/RV'06, volume 4262 of Lecture Notes in Computer Science, pages 70–84. Springer-Verlag, 2006.

28. J. Pietersma, A. Feldman, and A.J.C. van Gemund. *Modeling and Compilation Aspects of Fault Diagnosis Complexity*.
In AUTOTESTCON 2006 – Proceedings IEEE Systems Readiness Technology Conference, pages 502–508, Anaheim, California, USA, September 2006. IEEE.

29. J. Pietersma and A.J.C. van Gemund. *Diagnostic Accuracy of Models*.
In H.Y. Zhang, editor, 6^{th} IFAC Symposium on Fault Detection, Supervision and Safety of Technical Processes – SAFEPROCESS 2006, pages 913–918, Beijing, China, August 2006. International Federation of Automatic Control – IFAC.

30. J. Pietersma and A.J.C. van Gemund. *Temporal versus Spatial Observability in Model-Based Diagnosis*.
In Proceedings of IEEE Int. Conf. on Systems, Man, and Cybernetics – SMC 2006, Taipei, Taiwan, October 2006. IEEE.

31. J. Pietersma and A.J.C. van Gemund. *Benefits and Costs of Model-Based Fault Diagnosis for Semiconductor Manufacturing Equipment*.

In Proceedings of INCOSE 2007 – 17^{th} Int. Symposium on Systems Engineering, San Diego, CA, USA, June 24-28 2007.

32. J. Pietersma and A.J.C. van Gemund. *Symbolic Factorization of Propagation Delays out of Diagnostic System Models*.
In G. Biswas, X. Koutsoukos, and S. Abdelwahed, editors, 18^{th} Int. Workshop on Principles of Diagnosis – DX'07, pages 170–177, Nashville, USA, 2007.

33. J. Pietersma, A.J.C. van Gemund, and A. Bos. *A Model-Based Approach to Fault Diagnosis of Embedded System*.
In J.J. van Wijk, J.W.J. Heijnsdijk, K.G. Langendoen, and R. Veltkamp, editors, Proceedings of the 10^{th} Annual Conference of the Advanced School for Computing and Imaging – ASCI 2004 – Port Zélande, Ouddorp, The Netherlands, pages 189–196, Delft University of Technology, The Netherlands, June 2004. Advanced School for Computing and Imaging (ASCI).

34. J. Pietersma, A.J.C. van Gemund, and A. Bos. *A Model-Based Approach to Sequential Fault Diagnosis*.
In AUTOTESTCON 2005 – Proceedings IEEE Systems Readiness Technology Conference, pages 621–627, Orlando, Florida, USA, September 2005. IEEE.

35. M. Prins. *Testing Industrial Embedded Systems - An Overview*.
In Proceedings of INCOSE 2004 – 14^{th} Int. Symposium on Systems Engineering, Toulouse, France, June 20-24 2004.

36. J. Tretmans. *Model Based Testing with Labelled Transition Systems*.
In R. Hierons, editor, Testing with Formal Methods, Springer-Verlag, 2007.
To appear.

37. T.A.C. Willemse. *Heuristics for ioco-Based Test-Based Modelling*.
In L. Brim, B. Haverkort, M. Leucker, and J. v.d. Pol, editors, Formal Methods Applications and Technology: 11^{th} Int. Workshop on Formal Methods for Industrial Critical Systems – FMICS 2006, and 5^{th} Int. Workshop on Parallel and Distributed Methods in Verification – PDMC 2006, volume 4346 of Lecture Notes in Computer Science, pages 123–147. Springer-Verlag, 2007.

Technical reports

1. A. Bos. *A Model-Based Fault Detection and Diagnosis System for the EUV-Projection Optics Box*.
Technical Report ASML-TAN-TN-0001, ASML N.V., Veldhoven, The Netherlands.

2. R. Boumen, I.S.M. de Jong, J.W.H. Vermunt, J.M. van de Mortel-Fronczak, and

J.E. Rooda. *A Risk-Based Stopping Criterion for Test Sequencing*.
SE Report 420460, Eindhoven University of Technology, Eindhoven, The Netherlands, 2006.

3. R. Boumen, I.S.M. de Jong, J.M.G. Mestrom, J.M. van de Mortel-Fronczak, and J.E. Rooda. *Integration and Test Sequencing for Complex Systems*.
SE Report 2007-07, Eindhoven University of Technology, Eindhoven, The Netherlands, 2007.

4. N.C.W.M. Braspenning, J.M. van de Mortel-Fronczak, and J.E. Rooda. *Modeling, Analysis and Implementation of Infrastructure for Model-Based Integration and Testing*.
SE Report 2007-08, Eindhoven University of Technology, Eindhoven, The Netherlands, 2007.

5. I.S.M. de Jong, R. Boumen, J.M. van de Mortel-Fronczak, J.E. Rooda. *Test strategy analysis for manufacturing systems*.
SE Report 2007-10, Eindhoven University of Technology, Eindhoven, The Netherlands, 2007.

6. I.S.M. de Jong, R. Boumen, J.M. van de Mortel-Fronczak, J.E. Rooda. *Parallelizing test phases using graph partitioning algorithms*.
SE Report 2007-11, Eindhoven University of Technology, Eindhoven, The Netherlands, 2007.

7. I.S.M. de Jong, R. Boumen, J.M. van de Mortel-Fronczak, J.E. Rooda. *Test set improvement using a next-best-test-case algorithm*.
SE Report 2007-12, Eindhoven University of Technology, Eindhoven, The Netherlands, 2007.

8. I.S.M. de Jong, R. Boumen, J.M. van de Mortel-Fronczak, J.E. Rooda. *Selecting a suitable system architecture for integration and testing*.
SE Report 2007-13, Eindhoven University of Technology, Eindhoven, The Netherlands, 2007.

Master theses, bachelor theses, and internship reports

1. J. Anggono. *Verification and Model-Based Testing of the ASML Laser Subsystem*.
SAI/2yr Thesis, SAI-IPPS, Eindhoven University of Technology, Eindhoven, The Netherlands, 2004.

2. A. Barve. *Model-Based Diagnosis – An ASML Case Study*.

Master thesis, Delft University of Technology, EEMCS, Delft, The Netherlands. 2005.

3. E.L.G. Bertens. *Translation of χ Models for Verification with* UPPAAL.
   Bachelor thesis. Report SE 420463, Eindhoven University of Technology, Dept. of Mechanical Engineering, Eindhoven, The Netherlands, 2006.

4. G. van Bokhoven. *Model-Based Integration of Manufacturing Machines: Concepts and Applications*.
   Master thesis, report SE 420448, Eindhoven University of Technology, Dept. of Mechanical Engineering, Eindhoven, The Netherlands, 2005.

5. G. van Bokhoven. *Model-Based Testing: A State-of-the-Art Overview*.
   Short Internship, report SE 420381, Eindhoven University of Technology, Dept. of Mechanical Engineering, Eindhoven, The Netherlands, 2004.

6. M. van Campenhout. LONETTE.
   Bachelor thesis, Fontys Hogeschool, Eindhoven, The Netherlands, 2007.

7. M.J.M. Dohmen. *Integration and Test Sequencing Applied to ASML Software Releases*.
   Traineeship report, Eindhoven University of Technology, Dept. of Mechanical Engineering, Eindhoven, The Netherlands, 2006.

8. J. Ekelmans. *Test Phase Partitioning for Lithographic Machines*.
   Master thesis, report SE 420502, Einhoven University of Technology, Dept. of Mechanical Engineering, Eindhoven, The Netherlands, 2007.

9. A.G.C.L. Geubbels. *Model-Based Integration and Testing of High-Tech Multi-Disciplinary Systems*.
   Master thesis, report SE 420476, Eindhoven University of Technology, Dept. of Mechanical Engineering, Eindhoven, The Netherlands, 2006.

10. M. van der Heijden. LONETTE *1.0: A Tool for Test and Integration Strategies*.
    Bachelor thesis, Fontys Hogeschool, Eindhoven, The Netherlands, 2006.

11. R.M.P.J. Hendrikx. *Model-Based Testing of Complex Manufacturing Systems: A Case Study*.
    Master thesis, report SE 420386, Eindhoven University of Technology, Dept. of Mechanical Engineering, Eindhoven, The Netherlands, 2004.

12. J.H.M. van Lierop and F.A.G. van der Sterren. TWINSCAN *Hardware Simulator, Efficient Integreren*.
    Bachelor thesis, Fontys Hogeschool, Eindhoven, The Netherlands, 2006.

13. J.M.G. Mestrom. *Integration and Test Sequencing for Complex Manufacturing Systems*.

Master thesis, report SE 420484, Eindhoven University of Technology, Dept. of Mechanical Engineering, Eindhoven, The Netherlands, 2006.

14. V. Niculescu-Dinca. *Infrastructure for Early Integration, Connecting Multidisciplinary Simulators in a Distributed Environment*.
TWAIO thesis, Stan Ackermans Instituut, Eindhoven, The Netherlands, 2005.

15. L. Oosterhof. *High Speed Serial Link (HSSL) Test Device*.
Bachelor thesis, Fontys Hogeschool, Eindhoven, The Netherlands, 2006.

16. K.M. Peplowska. *Models of the* TWINSCAN *Laser Subsystem for Model-Based Testing*.
SAI/2yr Thesis, SAI-IPPS, Eindhoven University of Technology, Eindhoven, The Netherlands, 2004.

17. D. van de Pol. *Data Distribution Service (DDS) Connectors for LabView and TTCN*.
Bachelor thesis, Fontys Hogeschool, Eindhoven, The Netherlands, 2006.

18. M.H. Schonenberg. *Timed Modelling & Verification of the DO/DG Component – A Case Study in the Tangram project*.
Internship assignment, University of Twente, EWI Faculty, Enschede, The Netherlands, 2005.

19. F. Stappers. *Simulation, Verification, and Model Based Integration for a Hybrid System in an ASML Case Study*.
Master thesis, Einhoven University of Technology, Eindhoven, The Netherlands, 2007.

20. R. Theunissen. *The Test and Integration Simulation Model*.
Traineeship report SE 420477, Eindhoven University of Technology, Dept. of Mechanical Engineering, Eindhoven, The Netherlands, 2005.

21. P. Verduin. LONETTE*: A Tool and Infrastructure for LoA1*.
Bachelor thesis, Avans Hogeschool, Breda, 's-Hertogenbosch, Tilburg, The Netherlands, 2005.

22. J.W.H. Vermunt. *Test Sequencing of Complex Manufacturing Systems*.
Master thesis, report SE 420435, Eindhoven University of Technology, Dept. of Mechanical Engineering, Eindhoven, The Netherlands, 2005.

Professional articles

1. R. Boumen and I.S.M. de Jong. *Doorlooptijd Verkorting door het Gebruik van*

Optimale Test Volgordes.
In Dutch; Translated title: Reduction of lead times by using optimal test sequences.
Bits & Chips, August 2005.

2. R. Boumen and I.S.M. de Jong. *Wiskundige Teststrategie Belooft Weken Winst in Doorloop*.
In Dutch; translated title: Mathematical test strategy promises a lead time gain of weeks.
Bits & Chips, 7(14):16–19, 2005.

3. R. Boumen, I.S.M. de Jong, J.W.H. Vermunt, J.M. van de Mortel-Fronczak, and J.E. Rooda. *Test Sequencing in a Complex Manufacturing System*.
XOOTIC Magazine, 11(2):9–16, December 2005.

4. R. Boumen, N.C.W.M. Braspenning, I.S.M. de Jong, J.M. van de Mortel-Fronczak, and J.E. Rooda. *Methods and Algorithms for Model-Based Integration and Testing*.
SPIder Koerier, 2007(1):4–10, April 2007.

5. N.C.W.M. Braspenning, J.M. van de Mortel-Fronczak, and J.E. Rooda. *Model-Based Testing with Chi and TorX*.
XOOTIC Magazine, 11(2):17–23, December 2005.

6. W.J.A. Denissen. *A Multidisciplinary Model-Based Test and Integration Infrastructure*.
XOOTIC Magazine, 11(2):35–46, December 2005.

7. L. Engels, I.S.M. de Jong, and J. Tretmans. *Tangram met Modellen naar Snellere Integratie en Test*.
Bits & Chips, 9(11):32–35, 2007.

8. M. van Osch. *An Introduction to Tangram*.
XOOTIC Magazine, 11(2):5–8, December 2005.

9. J. Pietersma, A.J.C. van Gemund, and A. Bos. *A Model-Based Approach to Sequential Fault Diagnosis*.
IEEE Instrumentation and Measurement Magazine, 10(2):46–52, 2007.

10. J. Pietersma, A.J.C. van Gemund, and A. Bos. *A Model-Based Approach to Fault Diagnosis of Embedded System*.
XOOTIC Magazine, 11(2):25–33, December 2005.

Appendix B

List of authors

Ir. A. Belinfante
University of Twente
Axel.Belinfante@cs.utwente.nl

Dr. rer. nat. H. Bohnenkamp
Rheinisch-Westfälische Technische Hochschule Aachen
henrik@cs.rwth-aachen.de

Dr. ir. R. Boumen
ASML
Eindhoven University of Technology
r.boumen@tue.nl

Ir. N.C.W.M. Braspenning
Eindhoven University of Technology
n.c.w.m.braspenning@tue.nl

Ir. T. Brugman
ASML
Tom.Brugman@asml.com

Dr. ir. W.J.A Denissen
TNO Science and Industry
Will.Denissen@tno.nl

Ing. A.P.G.C. van Dongen
ICT Embedded B.V.
ad.van.dongen@esi.nl

Ir. L. Engels
ASML
Luud.Engels@asml.com

Prof. dr. ir. A.J.C. van Gemund
Delft University of Technology
a.j.c.vangemund@tudelft.nl

Ing. I.S.M. de Jong
ASML
Eindhoven University of Technology
ivo.de.jong@asml.com

Dr. ir. J.M. van de Mortel-Fronczak
Eindhoven University of Technology
j.m.v.d.mortel@tue.nl

Ir. H.A.J. Neerhof
ASML
johan.neerhof@asml.com

Ir. M.P.W.J. van Osch
Eindhoven University of Technology
M.P.W.J.van.Osch@tue.nl

Ir. D.O. van der Ploeg
ASML
durk.van.der.ploeg@asml.com

Dr. ir. G.J. Tretmans
Embedded Systems Institute
Radboud University Nijmegen
jan.tretmans@esi.nl

Ir. J. Pietersma
Delft University of Technology
j.pietersma@tudelft.nl

Prof. dr. ir. J.E. Rooda
Eindhoven University of Technology
j.e.rooda@tue.nl

Dr. ir. T.A.C. Willemse
Eindhoven University of Technology
t.a.c.willemse@tue.nl

Tangram partners (institutions and companies):

ASML	Veldhoven, The Netherlands
Science and Technology	Delft, The Netherlands
TNO Science and Industry	Delft, The Netherlands
Delft University of Technology	Delft, The Netherlands
Eindhoven University of Technology	Eindhoven, The Netherlands
Radboud University Nijmegen	Nijmegen, The Netherlands
University of Twente	Enschede, The Netherlands
Embedded Systems Institute	Eindhoven, The Netherlands

For more information: office@esi.nl

Bibliography

[1] R. Alur, T. Dang, J. Esposito, R. Fierro, Y. Hur, F. Ivancic, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky. Hierarchical Hybrid Modeling of Embedded Systems. In *Proceedings of the 1st International Workshop on Embedded Software – EMSOFT-01, Tahoe City, CA, USA*, volume 2211 of *Lecture Notes in Computer Science*, pages 14–31. Springer-Verlag, 2001.

[2] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 2(75):87–106, 1987.

[3] M.M. Arenthoft, J.J. Fuchs, Y. Parrod, A. Gasquet, J. Stader, and I. Stokes. OPTIMUM-AIV: A Planning and scheduling system for spacecraft AIV. *Future generation Computer Systems*, 7:403–412, 1991.

[4] D.A. van Beek, K.L. Man, M.A. Reniers, J.E. Rooda, and R.R.H. Schiffelers. Syntax and Semantics of timed Chi. Computer Science report 05–09, Eindhoven University of Technology, 2005.

[5] D.A. van Beek, K.L. Man, M.A. Reniers, J.E. Rooda, and R.R.H. Schiffelers. Syntax and Consistent Equation Semantics of Hybrid Chi. *Journal of Logic and Algebraic Programming, special issue on hybrid systems*, 68(1 - 2):129 – 210, 2006.

[6] D.A. van Beek, A. van der Ham, and J.E. Rooda. Modelling and Control of Process Industry Batch Production Systems. In *Proceedings of the 15th Triennial World Congress of the International Federation of Automatic Control – IFAC-02, Barcelona, Spain*, 2002.

[7] A. Belinfante. Timed Testing with TorX: The Oosterschelde Storm Surge Barrier. In M. Gijsen, editor, *Handout 8e Nederlandse Testdag*, Rotterdam, 2002. CMG.

[8] A. Belinfante. Torx test tool information. `http://fmt.cs.utwente.nl/tools/torx`, 2007.

[9] A. Belinfante, J. Feenstra, L. Heerink, and R.G. de Vries. Specification Based Formal Testing: The EasyLink Case Study. In *2nd Workshop on Embedded Systems – PROGRESS-01*, pages 73–82. STW, Utrecht, The Netherlands., 2001.

[10] A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal Test Automation: A Simple Experiment. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, 12^{th} *Int. Workshop on Testing of Communicating Systems*, pages 179–196. Kluwer, 1999.

[11] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In W. Reisig and G. Rozenberg, editors, *Lecture Notes on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer-Verlag, 2004.

[12] T. Berg, B. Jonsson, M. Leucker, and M. Saksena. Insights to Angluin's Learning. In S. Etalle, S. Mukhopadhyay, and A. Roychoudhury, editors, *Proceedings of SVV 2003*, volume 118 of *Electronic Notes in Theoretical Computer Science*, pages 3–18. Elsevier, 2005.

[13] M. Boasson. Control systems software. *IEEE Transactions on Automatic Control*, 38(7):1094–1106, 1993.

[14] B.W. Boehm and V.R. Basili. Software Defect Reduction Top 10 List. *IEEE Computer*, 34(1):135–137, 2001.

[15] H. Bohnenkamp and A. Belinfante. Timed Testing with TorX. In *Formal Methods Europe 2005*, volume 3582 of *Lecture Notes in Computer Science*, pages 173 – 188. Springer-Verlag, 2005.

[16] E.M. Bortnik, N. Trčka, A.J. Wijs, S.P. Luttik, J.M. van de Mortel-Fronczak, J.C.M. Baeten, W.J. Fokkink, and J.E. Rooda. Analyzing a χ model of a turntable system using SPIN, CADP and UPPAAL. *Journal of Logic and Algebraic Programming*, 65(2):51–104, 2005.

[17] E.M. Bortnik, D.A. van Beek, J.M. van de Mortel-Fronczak, and J.E. Rooda. Verification of timed Chi models using UPPAAL. In *Proceedings of the 2nd International conference on Informatics in Control, Automation and Robotics – ICINCO-05, Barcelona, Spain*, pages 486–492. INSTICC Press, 2005.

[18] E.M. Bortnik, J.M. van de Mortel-Fronczak, and J.E. Rooda. Verifying Chi models in UPPAAL. Systems Engineering report 2007–06, Eindhoven University of Technology, 2007.

[19] R. Boumen, I.S.M. de Jong, J.M. van de Mortel-Fronczak, and J.E. Rooda. Test time reduction by optimal test sequencing. *Proceedings of the 2006 INCOSE International Symposium*, 2006.

[20] R. Boumen, I.S.M. de Jong, J.M. van de Mortel-Fronczak, and J.E. Rooda. Optimal integration and test planning applied to lithographic systems. *Proceedings of the 2007 INCOSE International Symposium*, 2007.

[21] R. Boumen, I.S.M. de Jong, J.M. van de Mortel-Fronczak, and J.E. Rooda. Optimal integration and test strategies for software releases of lithographic systems. *Proceedings of the 5th Annual conference on Systems Engineering Research – CSER*, 2007.

[22] R. Boumen, I.S.M. de Jong, J.W.H. Vermunt, J.M. van de Mortel-Fronczak, and J.E. Rooda. A Risk-Based Stopping Criterion for Test Sequencing. SE Report 420460, Eindhoven University of Technology, Eindhoven, The Netherlands, 2006.

[23] R. Boumen, I.S.M. de Jong, J.W.H. Vermunt, J.M. van de Mortel-Fronczak, and J.E. Rooda. Test sequencing in complex manufacturing systems. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, 2006. Accepted for publication.

[24] R. Boumen, I.S.M. de Jong, J.M.G. Mestrom, J.M. van de Mortel-Fronczak, and J.E. Rooda. Integration sequencing in complex manufacturing systems. SE Report 2006-02, Eindhoven University of Technology, Systems Engineering Group, Department of Mechanical Engineering, Eindhoven, The Netherlands, 2006.

[25] J.B. Bowles. The new SAE FMECA standard. In *Proceedings of the Annual Reliability and Maintainability Symposium*, pages 48 – 53, 1998.

[26] L. Brandán Briones and Ed Brinksma. A test generation framework for quiescent real-time systems. In J. Grabowski and B. Nielsen, editors, *FATES04. Formal Approaches to Testing of Software (4th International Workshop)*, volume 3395 of *Lecture Notes in Computer Science*, pages 64–78. Springer-Verlag, 2004.

[27] N.C.W.M. Braspenning, E. Bortnik, J.M. van de Mortel-Fronczak, and J.E. Rooda. Analysis and implementation of infrastructure for model-based integration and testing. In *Proceedings of the 5th Annual Conference on Systems Engineering Research – CSER*, 2007.

[28] N.C.W.M. Braspenning, E.M. Bortnik, J.M. van de Mortel-Fronczak, and J.E. Rooda. Model-based system analysis using Chi and Uppaal: an industrial case study. *Computers in Industry*, 2007. Accepted for publication.

[29] N.C.W.M. Braspenning, J.M. van de Mortel-Fronczak, and J.E. Rooda. A model-based integration and testing method to reduce system development effort. *Electronic Notes in Theoretical Computer Science – Proceedings of the 2nd workshop on Model-Based Testing – MBT-06*, 164(4):13–28, 2006.

[30] N.C.W.M. Braspenning, J.M. van de Mortel-Fronczak, and J.E. Rooda. Modeling, analysis, and implementation of infrastructure for model-based integration and testing. Systems Engineering report 2007–08, Eindhoven University of Technology, 2007.

[31] N.C.W.M. Braspenning, D.O. van der Ploeg, J.M. van de Mortel-Fronczak, and J.E. Rooda. Model-based techniques for intelligent integration and testing in industry. In *Proceedings of the 17th International Symposium of INCOSE – INCOSE-07, USA*, 2007.

[32] L.G. Bratthall, P. Runeson, K. Ädelsward, and W. Eriksson. A Survey of Lead-time Challenges in the Development and Evolution of Distributed Real-time Systems. *Information and Software Technology*, 42(13):947–958, 2000.

[33] E. Brinksma and J. Tretmans. Testing Transition Systems: An Annotated Bibliography. In *Revised tutorial lectures of the 4th Summer School on Modelling and Verification of Parallel Processes – MOVEP-00, Nantes, France*, volume 2067 of *Lecture Notes in Computer Science*, pages 187–195. Springer-Verlag, 2001.

[34] M. Broy and O. Slotosch. From Requirements to Validated Embedded Systems. In *Proceedings of the 1st International Workshop on Embedded Software – EMSOFT-01, Tahoe City, CA, USA*, volume 2211 of *Lecture Notes in Computer Science*, pages 51–65. Springer-Verlag, 2001.

[35] H. Buus, R. McLees, M. Orgun, E. Pasztor, and L. Schultz. 777 Flight Controls Validation Process. *IEEE Transactions on Aerospace and electronic systems*, 33(2):656–666, April 1997.

[36] C. Campbell, Grieskamp W., L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Testing Concurrent Object-Oriented Systems with SPEC EXPLORER – Extended Abstract. In J.S. Fitzgerald, I.J. Hayes, and A Tarlecki, editors, *FM 2005: Int. Symposium of Formal Methods Europe*, volume 3582 of *Lecture Notes in Computer Science*, pages 542–547. Springer-Verlag, 2005.

[37] Ü.V. Çatalyürek and C. Aykanat. Patoh: Partitioning tools for hypergraphs. Technical report, Bilkent University, 2002.

[38] H. Chuma. Increasing complexity and limits of organization in the microlithography industry: implications for science-based industries. *Research Policy*, 35(3):394–411, April 2006.

[39] M.A. Cusumano and R.W. Selby. How Microsoft Builds Software. *Communications of the ACM*, 40(6):53–61, June 1997.

[40] Embedded Systems Institute. The Trader project. `http://www.esi.nl/trader`.

[41] A. Engel, I. Bogomolni, S. Shacher, and A. Grinman. Gathering historical life-cycle quality costs to support optimizing the VVT process. *Proceedings of the 14th Annual International Symposium of INCOSE*, 2004.

[42] ETSI. Testing and Test Control Notation: TTCN-3. `http://www.etsi.org/WebSite/Technologies/ttcn3.aspx`.

[43] ETSI. Testing standards for GSM/GPRS/3G telecommunication devices and infrastructure. `http://www.etsi.org`, 1999-2007.

[44] C.F. Eubanks, S. Kmenta, and K. Ishii. System Behavior Modeling as a Basis for Advanced Failure Modes and Effects Analysis. In *Proceedings of the 1996 ASME Design Engineering Technical conferences and Computers in Engineering conference*, August 1996.

[45] P.Th. Eugster, P.A. Felber, R. Guerraoui, and A. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[46] A. Feldman, J. Pietersma, and A.J.C. van Gemund. A Multi-Valued SAT-Based Algorithm for Faster Model-Based Diagnosis. In *Proc. 17th International Workshop on Principles of Diagnosis – DX-06*, June 2006.

[47] A. Feldman, J. Pietersma, and A.J.C. van Gemund. All roads lead to fault diagnosis: Model-based reasoning with LYDIA. In P.Y. Schobbens, W. Vanhoof, and G. Schwanen, editors, *Proceedings of the Eighteenth Belgium-Netherlands conference on Artificial Intelligence (BNAIC-06) Namur, Belgium*, pages 123–131, October 2006.

[48] A. Feldman, A.J.C. van Gemund, and A. Bos. A Hybrid Approach to Hierarchical Fault Diagnosis. In *Proc. 16th International Workshop on Principles of Diagnosis – DX-05*, pages 101–106, 2005.

[49] J.C. Fernandez, C. Jard, T. Jeron, and C. Viho. An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology. In J.F. Groote and M. Rem, editors, *Special Issue of Industrially Relevant Applications fo Formal Analysis Techniques*. Elsevier, 1996.

[50] L. Frantzen, J. Tretmans, and T. Willemse. Test Generation Based on Symbolic Specifications. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Software Testing – FATES 2004*, volume 3395 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2005.

[51] L. Frantzen, J. Tretmans, and T.A.C. Willemse. A Symbolic Framework for Model-Based Testing. In K. Havelund, M. Núñez, G. Roşu, and B. Wolff, editors, *Formal Approaches to Software Testing and Runtime Verification – FATES/RV-06*, volume 4262 of *Lecture Notes in Computer Science*, pages 40–54. Springer-Verlag, 2006.

[52] P. Giordano and P. Messidoro. European and international verification and testing standards. In *Proceedings 4th International Symposium on Environmental Testing for Space Programmes, Liege, Belgium*. ESA, ESA SP-467, 2001.

[53] P. Godefroid. VeriSoft: A Tool for the Automatic Analysis of Concurrent Reactive Software. In *Proceedings of CAV'97*, volume 1254 of *Lecture Notes in Computer Science*, pages 476–479. Springer-Verlag, 1997.

[54] P.A.M. Haagh, A.U. Wilkens, H.J.A. Rulkens, E.J.J. van Campen, and J.E. Rooda. Application of a layout design method to the dielectric decomposition area in a 300 mm wafer fab. In *Proceedings of the 7th International Symposium on Semiconductor Manufacturing – ISSM-98, Tokyo, Japan*, pages 69–72. Ultra Clean Society, 1998.

[55] V.L. Hanh, K. Akif, Y.L. Traon, and J.M. Jézéquel. Selecting an efficient OO integration testing strategy: An experimental comparison of actual strategies. *ECOOP*, pages 381–401, 2001.

[56] M.J. Harrold. Testing: A Roadmap. In A. Finkelstein, editor, *ICSE - Future of SE Track*, pages 61–72. ACM, 2000.

[57] A. Hartman and K. Nagin. The AGEDIS Tools for Model Based Testing. In *Int. Symposium on Software Testing and Analysis – ISSTA 2004*, pages 129–132, New York, USA, 2004. ACM Press.

[58] M. Heemels and G. Muller. *Boderc: Model-Based Design of High-Tech Systems*. Embedded Systems Institute, Eindhoven, The Netherlands, 2006.

[59] J.W. Horch. *Practical guide to software quality management*. Artech House, 2nd edition, 2003.

[60] J. Huang, J. Voeten, and H. Corporaal. Correctness-preserving synthesis for real-time control software. In *Proceedings of the 6th International Conference on Quality Software – QSIC-06, Beijing, China*, pages 65–73. IEEE Computer Society Press, 2006.

[61] H. Hungar, T. Margaria, and B. Steffen. Domain-specific optimization in automata learning. In W.A. Hunt Jr. and F. Somenzi, editors, *Proceedings of CAV'03*, volume 2725 of *Lecture Notes in Computer Science*, pages 315–327. Springer-Verlag, 2003.

[62] H. Hungar, T. Margaria, and B. Steffen. Test-based model generation for legacy systems. In *IEEE international test conference – ITC*, pages 971–980, 2003.

[63] Informa Telecoms and Media. *Future Mobile Handsets*. Informa Telecoms and Media, 2006.

[64] International Electrotechnical Commission. IEC 61508: Functional safety of electronical/electronic/programmable electronic safety-related systems. Part 7: Overview of techniques and measures. IEC Standard, 2005.

[65] C. Jard and T. Jéron. TGV: Theory, Principles and Algorithms: A Tool for the Automatic Synthesis of Conformance Test Cases for Non-Deterministic Reactive Systems. *Software Tools for Technology Transfer*, 7(4):297–315, 2005.

[66] I.S.M. de Jong, R. Boumen, J.M. van de Mortel-Fronczak, and J.E. Rooda. An overview of integration and test plans in organizations with different business drivers. In B. Sauser and G.M. Muller, editors, *Proceedings of the 5th Annual conference on Systems Engineering Research – CSER*, volume 1. Stevens Institute of Technology, March 2007.

[67] I.S.M. de Jong, R. Boumen, J.M. van de Mortel-Fronczak, and J.E. Rooda. Parallelizing test phases using graph partitioning algorithms. SE Report 2007-11, Eindhoven University of Technology, Systems Engineering Group, Department of Mechanical Engineering, Eindhoven, The Netherlands, 2007.

[68] I.S.M. de Jong, R. Boumen, J.M. van de Mortel-Fronczak, and J.E. Rooda. Selecting a suitable system architecture for integration and testing. SE Report 2007-13, Eindhoven university of technology, Systems Engineering Group, Department of Mechanical Engineering, Eindhoven, The Netherlands, 2007.

[69] I.S.M. de Jong, R. Boumen, J.M. van de Mortel-Fronczak, and J.E. Rooda. Test set improvement using a next-best-test-case algorithm. SE Report 2007-12, Eindhoven University of Technology, Systems Engineering Group, Department of Mechanical Engineering, Eindhoven, The Netherlands, 2007.

[70] I.S.M de Jong, R. Boumen, J.M. van de Mortel-Fronczak, and J.E. Rooda. Test strategy analysis for semi-conductor manufacturing systems. SE report 2007-10, Eindhoven University of Technology, 2007.

[71] J. de Kleer and B.C. Williams. Diagnosing multiple faults. In Matthew L. Ginsberg, editor, *Readings in Nonmonotonic Reasoning*, pages 372–388, Los Altos, California, 1987. Morgan Kaufmann.

[72] A. Kleppe, W. Bast, and J. Warmer. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Professional, 1st edition, 2003.

[73] M. Krichen. The Timed Test Generator tool. `http://www-verimag.imag.fr/~krichen/ttg/index.html`, 2007.

[74] M. Krichen and S. Tripakis. Black-Box Conformance Testing for Real-Time Systems. In S. Graf and L. Mounier, editors, *Proc. 11th Int. SPIN Workshop – SPIN-2004*, volume 2989 of *Lecture Notes in Computer Science*, pages 109–126. Springer-Verlag, 2004.

[75] K. G. Larsen, M Mikucionis, B. Nielsen, and A Skou. Testing Real-Time Embedded Software using UPPAAL-TRON. In *The 5th ACM International conference on Embedded Software*, 2006.

[76] B.T. Laugen, N. Acur, H. Boer, and J. Frick. Best manufacturing practices - What do the best-performing companies do? *International Journal of Operations and Production Management*, 25(2):131–150, 2005.

[77] H.K.N. Leung and L.J. White. Insights Into Regression Testing. *Journal of Software Maintenance: Research and Practice*, 2:209–222, 1990.

[78] X. Liu, J. Liu, J. Eker, and E.A. Lee. Heterogeneous Modeling and Design of Control Systems. In *Software-Enabled Control: Information Technology for Dynamical Systems*, pages 105–122. Wiley-IEEE Press, 2003.

[79] K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. PhD thesis, Eindhoven University of Technology, 2006.

[80] Maplesoft. Maple. `http://www.maplesoft.com`, 2007.

[81] T. Margaria, H. Raffelt, and B. Steffen. Knowledge-based relevance filtering for efficient system-level test-based model generation. *Innovations in Systems and Software Engineering – A Nasa Journal*, 1(2):147–156, 2005.

[82] L.S.H. de Mello and A.C. Sanderson. A correct and complete algorithm for the generation of mechanical assembly sequences. *IEEE Transactions on Robotics and Automation*, 7(2):228–240, 1991.

[83] L.S.H. de Mello and A.C. Sanderson. Representations of mechanical assembly sequences. *IEEE Transactions on Robotics and Automation*, 7(2):211–227, 1991.

[84] Microsoft Research. Spec Explorer. `http://research.microsoft.com/specexplorer`.

[85] M. Mikucionis, B. Nielsen, and K. G. Larsen. Real-time System Testing On-the-Fly. In K. Sere and M. Waldén, editors, *15th Nordic Workshop on Programming Theory*, pages 36–38. Abo Akademi, Department of Computer Science, Finland, 2003.

[86] P. Millard, P. Saint-Andre, and R. Meijer. XEP–0060: Publish-Subscribe. `http://www.xmpp.org/extensions/xep-0060.html`, 2006. Jabber Software Foundation.

[87] J.M. van de Mortel-Fronczak, J. Vervoort, and J.E. Rooda. Simulation-based design of machine control systems. In *Proceedings of the 15th European Simulation Multiconference, Prague, Czech Republic*, 2001.

[88] National Instruments. Compact Fieldpoint product information. `http://www.ni.com/compactfieldpoint`, 2007.

[89] I. Ogren. On Principles for Model-Based Systems Engineering. *Systems Engineering*, 3(1):38–49, 2000.

[90] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus: an architecture for extensible distributed systems. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 58–68, New York, NY, USA, 1993. ACM Press.

[91] OMG. Data Distribution Service for Real-time Systems. `http://www.omg.org/technology/documents/formal/data_distribution.htm`.

[92] OMG. Common Object Request Broker Architecture – Corba. `http://www.corba.org`, 2007.

[93] Opto 22. SNAP product information. `http://www.opto22.com/ad/pac.aspx`, 2007.

[94] M. van Osch. Hybrid Input-output Conformance and Test Generation. In K. Havelund, M. Nunez, G. Rosu, and B. Wolff, editors, *Proceedings of FATES/RV-2006*, volume 4262 of *Lecture Notes in Computer Science*, pages 70 – 84. Springer Verlag, 2006.

[95] G. Pardo-Castellote. OMG Data-Distribution Service: Architectural Overview. In *Proceedings of the 23 rd International conference on Distributed Computing Systems Workshops – ICDCSW03*, pages 200–206. IEEE Computer organization, IEEE, May 2003.

[96] K.R. Pattipati and M.G. Alexandridis. Application of heuristic search and information theory to sequential diagnosis. *IEEE Trans. Syst. Man, Cybern.*, 20:872–887, 1990.

[97] K.S. Pawar, U. Menon, and J.C.K.H. Riedel. Time to Market. *Integrated manufacturing systems*, 5(1):14–22, 1994.

[98] D. Peled, M.Y. Vardi, and M. Yannakakis. Black Box Checking. *Journal of Automata, Languages, and Combinatorics*, 7(2):225–246, 2002.

[99] J. Pietersma and A.J.C. van Gemund. Diagnostic Accuracy of Models. In H.Y. Zhang, editor, 6^{th} *IFAC Symposium on Fault Detection, Supervision and Safety of Technical Processes – SAFEPROCESS-2006, Beijing, China*, pages 913–918, August 2006.

[100] J. Pietersma and A.J.C. van Gemund. Temporal versus Spatial Observability in Model-Based Diagnosis. In *Proceedings of IEEE Int. Conf. on Systems, Man, and Cybernetics – SMC, Taipei, Taiwan. 2006*, October 2006.

[101] J. Pietersma and A.J.C. van Gemund. Symbolic factorization of propagation delays out of diagnostic system models. In *Proc. 18th International Workshop on Principles of Diagnosis – DX-07*, May 2007.

[102] J. Pietersma, A.J.C. van Gemund, and A. Bos. A model-based approach to sequential fault diagnosis. *Proc. of IEEE AUTOTESTCON 2005, Orlando, Florida, USA*, pages 621–627, September 2005.

[103] C. Potts. Software-Engineering Research Revisited. *IEEE Software*, 10(5):19–28, September/October 1993.

[104] Python.org. Python. `http://www.python.org`, 2007.

[105] A. Raven. *Consider it Pure Joy... An introduction to clinical trials*. Cambridge Healthcare Research Ltd., 1997.

[106] A. Raven. *Beyond What is Written. A researchers guide to good clinical practice*. Cambridge Healthcare research Ltd., 1998.

[107] R. Reiter. A Theory of Diagnosis from First Principles. In Matthew. L. Ginsberg, editor, *Readings in Nonmonotonic Reasoning*, pages 352–371, Los Altos, California, 1987. Kaufmann.

[108] J.A. Rowson. Hardware/software co-simulation. In *Proceedings of the 31st Design Automation conference – DAC-94, San Diego, CA, USA*, pages 439–440. ACM Press, 1994.

[109] A. Rozinat, I.S.M. de Jong, C.W. Günther, and W.M.P. van der Aalst. Process Mining of Test Processes: A Case Study. BETA Working Paper Series WP 220, Eindhoven University of Technology, Eindhoven, 2007.

[110] M. Shakeri, V. Raghavan, K.R. Pattipati, and A. Patterson-Hine. Sequential testing algorithms for multiple fault diagnosis. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, 30(1):1–14, 2000.

[111] C.E. Shannon. A mathematical theory of communication. *Bell Sys. Tech. Journal*, 27:379–623, 1948.

[112] J.G. Springintveld, F.W. Vaandrager, and P.R. D'Argenio. Testing timed automata. *Theoretical Computer Science*, 254(1–2):225–257, 2001.

[113] V. Stavridou. Integration Standards for Critical Software Intensive Systems. In *ISESS '97: Proceedings of the 3rd International Software Engineering Standards Symposium – ISESS-97*, page 99, Washington, DC, USA, 1997. IEEE Computer Society.

[114] Systems Engineering Group, Mechanical Engineering Department, Eindhoven University of Technology. Chi language and tools. `http://se.wtb.tue.nl/sewiki/chi`, 2007.

[115] L. Tan. CharonTester. `http://www.cis.upenn.edu/%7Etanli/tools/charontester.html`, 2007.

[116] L. Tan, J. Kim, I. Lee, and O. Sokolsky. Model-based Testing and Monitoring for Hybrid Embedded Systems. In *Proceedings of IEEE Internation conference on Information Reuse and Integration – IRI-03*. IEEE Society, 2003.

[117] J. Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.

[118] J. Tretmans. Model Based Testing with Labelled Transition Systems. In R. Hierons, editor, *Testing with Formal Methods*. Springer-Verlag, 2007. To appear.

[119] J. Tretmans and E. Brinksma. TorX: Automated model based testing. In A. Hartman and K. Dussa-Ziegler, editors, *Proceedings of the 1st European conference on Model-Driven Software Engineering*, Nürnberg, Germany, 2003.

[120] UPPAAL. `http://www.uppaal.com`, 2007.

[121] B. Vastenhouw and R.H. Bisseling. A Two-Dimensional Data Distribution Method For Parallel Sparse Matrix-Vector Multiplication. *SIAM Review*, 47(1):67–95, 2005.

[122] R.G. de Vries, A. Belinfante, and J. Feenstra. Automated Testing in Practice: The Highway Tolling System. In I. Schieferdecker, H. König, and A. Wolisz, editors, *Testing of Communicating Systems XIV*, pages 219–234. Kluwer Academic Publishers, 2002.

[123] A. van Weelden, M. Oostdijk, L. Frantzen, P. Koopman, and J. Tretmans. On-the-Fly Formal Testing of a Smart Card Applet. In R. Sasaki, S. Qing, E. Okamoto, and H. Yoshiura, editors, *Security and Privacy in the Age of Ubiquitous Computing – Procs. of the* 20^{th} *IFIP TC11 Int. Information Security Conference*, volume 181 of *IFIP Series*, pages 565–576. Springer-Verlag, 2005.

[124] T.A.C. Willemse. Heuristics for **ioco**-Based Test-Based Modelling (extended abstract). In L. Brim, B. Haverkort, M. Leucker, and J. van de Pol, editors, *Proceedings of FMICS and PDMC 2006*, volume 4346 of *Lecture Notes in Computer Science*, pages 123–147. Springer-Verlag, 2007.

[125] B.C. Williams and R. Ragno. Conflict-directed A^* and its role in model-based embedded systems. *Journal of Discrete Applied Mathematics*, 2004.