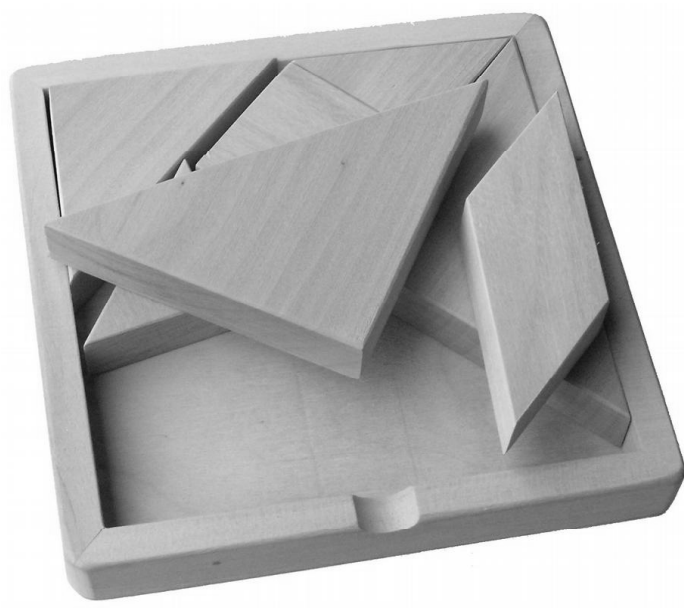
XOOTIC magazine

December 2005-Volume 11-Number 2

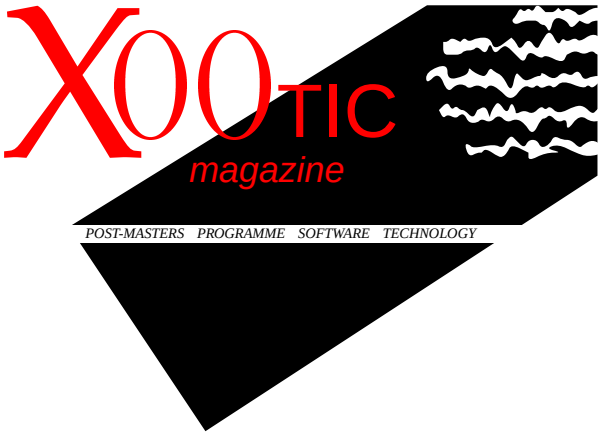
POST-MASTERS PROGRAMME SOFTWARE TECHNOLOGY

TANGRAM



Contents

Advertorials



Colofon

XOOTIC MAGAZINE
Volume 11, Number 2
December 2005

Editors
S. Estok
M.M. Lindwer
M.P.W.J. van Osch
L. Posta

Address
XOOTIC and XOOTIC MAGAZINE
P.O. Box 6122
5600 MB Eindhoven
The Netherlands
xootic@win.tue.nl
`http://www.win.tue.nl/xootic/`

Secretariat OOTI
Post-masters Programme
Software Technology
Eindhoven University of Technology, HG 6.57
P.O. Box 513
5600 MB Eindhoven
The Netherlands
tel. +31 40 2474334
fax. +31 40 2475895
ooti@win.tue.nl
`http://wwwooti.win.tue.nl/`

Printer
Offsetdrukkerij De Witte, Veldhoven

TANGRAM

editorial

The history of Tangram goes back hundreds of years. The time when the puzzle was invented is actually unknown. The earliest known Chinese book on the game dates back to 1813. The origin of the word Tangram is also unknown. Some stories suggest that it comes from the ”Tan” dynasty, others suggest it comes from Chinese river people called ”Tanka”, and others suggest it comes from the English word ”Tramgram” which means puzzle or trinket.

The history of TANGRAM goes back to 2003. The Embedded Systems Institute started a project with this name, with ASML and several university and industrial partners. The origin of the project name is: *”Test Approach based on iNtegrated product Generation and product Realization applied to Asml Machines”*. The goal of the TANGRAM project is to research and validate techniques for lead-time and cost reduction of embedded systems development. To achieve that, the TANGRAM project focusses on early test and integration, test automation, and diagnoses, all using models.

This issue of the XOOTIC magazine is entirely devoted to the TANGRAM project, of which I am also a member of. It contains a global introduction on the TANGRAM project and four in depth articles on several areas the project has been, and still is, working on.

We wish you a joyful reading of this issue of our magazine.

Michiel van Osch, editor

Embedded Systems
INSTITUTE

THE EMBEDDED SYSTEMS INSTITUTE IS LOOKING FOR

NEW *RESEARCH FELLOWS*

The Embedded Systems Institute (ESI) is a research center and a center of expertise for embedded systems. It does industrial research in the area of complex, software-controlled systems. The research projects at ESI are driven by problems from industry and are carried out in teams in which researchers from ESI, from industry, and from universities cooperate. To extend its own research staff ESI is now hiring *Research Fellows*.

Research fellows are candidates with industrial or academic backgrounds.
Candidates with industrial backgrounds must have experience in embedded systems design, and they preferably hold PhD degrees. Academic candidates must hold PhD degrees in one of the disciplines that are relevant to embedded systems, and they have actively shown interest in embedded systems. Examples of relevant disciplines are software engineering, control theory, systems engineering, and digital electronics.

Research Fellows are people with the ambition to become internationally recognized experts in (aspects of) embedded systems design.
Since this a discipline that is still in its infancy, the Research Fellows have the challenge and the opportunity to work actively in a new field that still has to be shaped. By participating in ESI research projects they keep in touch with the latest developments. In the projects they often use their expertise and experience to guide and coach other team members. Because ESI depends on these projects for the buildup of its expertise, the Research Fellows play an active role in the selection of new research projects.

Research Fellows transfer their expertise to others.
They give presentations at conferences, they publish in journals, and they give courses and seminars. It can be very attractive to combine a 4/5 position as ESI Research Fellow with a 1/5 position as part-time (associate) professor at a university, or a 1/5 corresponding senior position in industry.

ESI Research Fellows like to work in teams.
They especially enjoy working on problems that are at the cutting edge between industrial applications and scientific research. They like to work in multidisciplinary teams with experts from different technological disciplines. They like to listen to others, and they know that sharing of expertise and experience is essential in teamwork.

How to apply

- The ESI website (www.esi.nl) contains all kinds of information about the institute and its projects.
- The scientific director of the Embedded Systems Institute (ed.brinksma@esi.nl) may be contacted for more information about the job contents.
- The director of opererations of ESI (reinier.van.eck@esi.nl) can give information about benefits and conditions associated with the positions.
- Applications and inquiries can be e-mailed to office@esi.nl.
- The ESI is reachable by telephone at +31 (0)40 247 4720.

The Embedded Systems Institute is a foundation that is financially sponsored by its seven founders (Océ, ASML, Philips, TNO, the universities of Eindhoven, Delft, and Twente) and by the Netherlands Ministry of Economic Affairs. ESI is located at the university campus in Eindhoven. It was founded in 2002. Besides its own staff of around 20 people, some 70 guest researchers work at ESI. These numbers are expected to grow to 30 and 150 within a few years. ESI offers competitive salaries and is flexible in its modes of appointment. ESI employees can use the childcare facilities at the university campus.

An introduction to Tangram[1,2]

Edited by Michiel van Osch

The Tangram project aims at a significant reduction of lead time and cost in the integration and test phase of complex high-tech products. At the same time the product quality should be maintained or improved. In this paper we give a brief overview of the Tangram project.

Introduction

The performance demands on high-tech products keep on growing; they should be faster, more accurate, their uptime should be increased, etc. The business demands on these products keep on growing as well; the time to get such products to the market is getting shorter and the same goes for the period in which the return on investment can be obtained. So while engineers have to do their utmost to deliver technology that sometimes has not been invented yet, the market dictates them to do it faster, cheaper and better. This challenge is never more present than when system parts from different projects and from different disciplines (optics, mechanics, electronics and software) have to be integrated and tested. The combination of this 'faster, cheaper and better' issue and integrating multi-disciplinary state of the art technology, gives ample reason to want a breakthrough. It gives ample reason to want TANGRAM.

Project Organization

Tangram has teamed up the expertise and competence required to establish a breakthrough. Three universities: Delft University of Technology; Eindhoven University of Technology and Radboud University Nijmegen are involved. The institutes ESI and TNO-TPD are involved. The industrial partners involved are Science & Technology and finally ASML as carrying industrial partner. The project is partly subsidized by the Dutch Ministry of Economic Affairs.

TANGRAM foresees research and development along four Lines of Attention (LoA) that will be constantly challenged by a real life industrial case: a wafer scanner at ASML (See Figure 1).

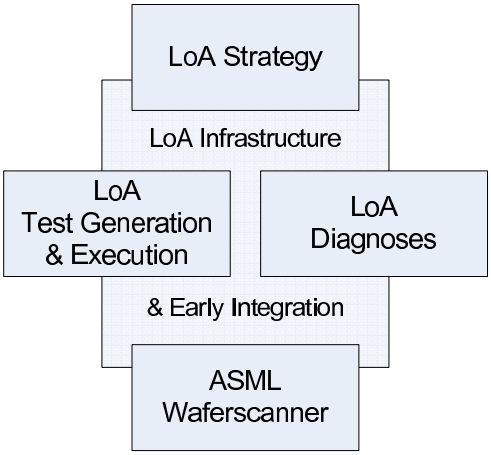


Figure 1: TANGRAM Organization

- The LoA with focus on strategy develops models of integration- and test processes featuring cost, parts to integrate, combinations to test, time to spend and product quality to achieve, as well as methods for test selection.

[1]This work has been carried out as part of the Tangram project under the responsibility of the Embedded Systems Institute. This project is partially sponsored by the Netherlands Ministry of Economic Affairs under grant TSIT 2026.

[2]Most of the content of this article is already published on the TANGRAM project site at `http://www.embeddedsystems.nl`.

- The LoA that concentrates on test generation & execution delivers model-based methods and tools for generating, executing and evaluating test cases as well as model-based simulations for parts that are not yet available.
- The LoA that aims at model-based diagnosis delivers methods and tools that are capable of diagnosing the internals of a system by monitoring its inputs and outputs.
- The infrastructure and early integration LoA delivers an environment that is capable of interconnecting the models and tools that result from the other LoA's, and develops methods and tools for early integration and test with models.

LoA Strategy

The goal of this LoA is to define the optimal test strategy for a certain product. An optimal test strategy in terms of total test duration and/or final product quality and cost is desired. The optimal test strategy for a product is influenced by the objectives of the test phase and the constraints that follow out of that. For instance a test strategy for a time-to-market driven environment (like ASML) is different from a test strategy for a quality driven environment (like aircraft industry). A good test strategy is therefore product dependent.

For complex systems an infinite number of test cases can be derived. Executing all possible test cases between the moment that a (sub-)system is ready and the system is released is therefore impossible. Selecting the optimal set of test cases is therefore a relevant question. This LoA investigates the possibility to solve this selection problem with test selection algorithms.

The third point of interest is the total duration of the test period and the resources required to do so. With total duration we mean the time it takes to execute all test cases successfully, so including fixing problems. Adding additional test resources is the common thing to do. This results in detecting more problems, which seems a good thing. But if your real bottleneck is in the fixing of problems, then adding test resources is not the best thing to do. So this means that the initial quality of the system, the available resources and the test cases to be executed determine the architecture of your 'Test Factory'. Different configurations of 'Test Factories' are modelled to investigate these effects and others on the end result, total test duration and end quality.

This magazine contains an article by Boumen et. al., in which they describe how to optimize test sequences such that it takes less cost or time and demonstrate this on an ASML TWINSCAN lithographic machine.

The current approach is to define the determination of the best test strategy into 3 phases: strategy selection, test selection and test scheduling. Optimal techniques for each phase are researched and developed. Additionally a simulator of the Test Factory has been developed to simulate the effect of the different developed test strategies on the total test duration and product quality.

LoA Test Generation and Execution

The objective of Line of Attention 3 is to improve the efficiency and effectiveness of the testing process by developing testing methodology, techniques, and tools using a model based approach.

In model based testing a model of the system under test (SUT) is developed. Models can be formal, such as those written in languages as Chi, Lotos, or Promela, or in semi-formal languages, such as state diagrams or UML models. A model is the basis for the automatic generation of test cases using a test derivation algorithm, and test results are automatically analyzed and evaluated with respect to the model. Moreover, a model can be used to simulate a part of the system under test during integration testing, if such a part is not yet available.

Starting points for Line of Attention 3 are models based on transition systems, the so-called ioco-test theory, and the prototype test tool TorX [3].

Via a case study driven approach we will work on extensions of these incorporating real-time testing, testing of complex data structures, testing of hybrid systems, compositional and integration testing, and testing of multi-disciplinary, non-software aspects.

There are several benefits of model based testing. First, a model can serve as a precise and unambiguous basis for testing, thus allowing formal validation of tests. Second, models make it possible to automatically derive test cases and evaluate test results, thus considerably reducing the manual effort of testing. In particular in case of modifications in

the system, a small adaptation in the model is sufficient to generate a complete new set of test cases. Although making models requires some effort, it is expected that this effort is more than compensated by the advantages of faster, more efficient, and higher quality testing.

The challenge of this Line of Attention is to extend the existing state of the art in model based testing in such a way that, on the one hand, there is a solid and well-founded theoretical basis, and on the other hand it leads to high applicability for testing the ASML systems.

This magazine contains an article by Braspenning et. al., about a case-study on automatic model-based testing with TorX using χ as specification language [2, 3]. As a result, they found interface discrepancies between the laser unit (3rd party) and controller of a lithography machine.

LoA Diagnosis

Throughout the design, integration, and operational phase, systems are plagued by faults. Finding the root cause of system malfunction typically consumes many resources that could be spent much more efficiently. This fault diagnosis process becomes even more problematic as system become more complex. Model-Based Diagnosis (MBD) is a computerized technique that considerably increases the efficiency and accuracy of fault diagnosis.

Current MBD techniques, however, are still not adequate to handle, complex, multidisciplinary systems as found in ASML. Given an adequate MBD technique, in turn, a subsequent problem is model specification, which is a labor-intensive and error-prone process. In this line-of-attention, we aim:

- to extend current MBD technology by including features such as state, time, and probabilities in order to provide the modelling capabilities required;
- to develop a technique to (semi-) automatically derive/integrate (partial) fault diagnostic system models from/within existing design specifications.

The MBD approach is based on decomposing the diagnostic system (software) in two major components:

- the system-specific reference model, describing normative and fault behaviors, and
- a generic, diagnostic inference engine that executes the search process (the 'diagnosis algorithm'), guided by comparing actual system measurements with predictions from the reference model.

This diagnostic algorithm includes both exclusion and deduction, reasoning probabilistically over time. Consequently, development of diagnosis software reduces to reference model specification, which acts as a source code from which the diagnostic (embedded) software is automatically generated.

Aimed to provide proof-of-concept, in this line-of-attention we conduct case studies where we develop diagnostic models of relevant subsystems, apply them to realistic test data, and evaluate their diagnostic performance by comparing their diagnostic output with the diagnosis found in practice. Based on this feedback, we iteratively refine the diagnostic models and algorithms in order to determine a good trade-off between diagnosis effort (manual and computational) and diagnostic performance. The research is conducted by Delft University of Technology in cooperation with Science & Technology BV.

This magazine contains an article by Pietersma et. al., which describes the model-based diagnosis methodology as a solution for the fault diagnosis of an integrated system by inferring the health of a system from a compositional system model and real-world measurements

LoA Infrastructure and Early Integration

Infrastructure

The modelling, simulation, testing, and diagnoses techniques developed by the other LoA's need to be integrated in the ASML test and verification methodology and tools. It will be investigated how these modeling and simulation techniques can be integrated in the ASML test and verification methodology. For instance, in case the simulation models do not reflect the reality correctly (anymore), the models should be easily maintainable. Furthermore,

we also want to integrate other existing test and integration techniques, and tools, into the ASML methodology.

Therefore, the main objective is the conception of an integrated simulation and test environment that has the following features:

- support for real implementations (software, hardware) as well as simulation models;
- support for component integration;
- support for batch mode testing (e.g. for regression testing);
- support for automated test execution and pass/fail verdict;
- support for (remote) model based diagnostics, using same interfacing for models and real implementations;
- support for hybrid (discrete event, continuous time) models;
- support for real-time and simulation time execution.

Early Integration

In current practice testing is mainly performed after completion of the product development and prior to shipment. This implies that testing directly influences the shipment date. To test multi-disciplinary components (e.g. combining software with electronics, mechanics or optics), all components need to be available. For some (mechanical or optical) components this results in a significant investment to have the actual components available. When time-to-market concerns limit the amount of testing time, the rigor of testing is reduced. Consequently, the risk increases that certain malfunctions are not found prior to shipment. The resulting reduction in availability directly impacts market share. Given the above (three fold) problem statement, this results into the following observations:

Component dependencies (software, hardware) and availability of those components directly limits the test scheduling. This directly determines the time to market and predictability of shipment date. Components abstracting hardware cannot be tested without the actual underlying hardware. This directly results in the need to use expensive resources up to complete production quality systems. Currently, testing is mainly manual and the implementation, documentation, and evaluation of test procedures influences the product quality. The time to market pressure dictates the amount and rigor (coverage) of tests.

To address the above problems, the usage of modelling and simulation techniques will be investigated in this Line of Attention. With following this approach:

- Testing can be started before all components are completed,
- Testing of combined multi-disciplinary components can be done with simulated hardware components, and
- Testing can be made concurrent with system development, allowing an increase in the total testing rigor.

References

[1] TorX, `http://www.purl.org/net/TorX/`,2005

[2] R.R.H. Schiffelers, D.A. van Beek, K.L. Man, M.A. Reniers, J.E. Rooda, *A Hybrid Language for Modelling, Simulation and Verification*, `http://yp.wtb.tue.nl/pdfs/5281.pdf`, 2005

[3] D.A. van Beek, K.L. Man, M.A. Reniers, J.E. Rooda, R.R.H. Schiffelers, *Syntax and Consistent Semantics of Hybrid Chi*, CS-Report 04-37, `http://w3.wtb.tue.nl/nl/people_pages/?\&script=showabstract.php\&outid=4880`, November 2004

Test sequencing in a complex manufacturing system[1]

R. Boumen, I.S.M. de Jong, J.W.H. Vermunt, J.M. van de Mortel-Fronczak and J.E. Rooda

Testing complex manufacturing systems, like the ASML TWINSCAN [2] lithographic machine, takes a lot of time and costs. Within the Tangram project, methods are investigated to reduce this test costs. In this article, we describe a method which is used to optimize a test sequence such that it takes the least amount of costs, or time. With several cases we demonstrate that this method can be used to optimize test sequences within the manufacturing of a TWINSCAN lithographic machine such that cycle time is reduced.

Introduction

In today's industry, time to market is extremely important. In their drive to reduce systems time-to-market, many companies develop their systems concurrently. The final phase within concurrent development of systems is integration and test. This phase is on the critical path, and therefore has great influence on time-to-market (see [3]). The goal of the Tangram project is to reduce the time and cost spent on testing and integrating, and by that reduce time-to-market and cycle time of a system. Within the Tangram project, we look at test and integration strategy. A test and integration strategy defines a test and integration phase which is optimal in terms of time, costs and/or quality. In our work we are looking at methods that select or optimize test and integration strategies, taking into account time, costs and quality.

In this article, we describe a method to create optimal or near-optimal test sequences. A test sequence is a key element of the test and integration strategy. The basis of this method is described as Sequential Diagnosis by Pattipati [4], who used this method for the diagnosis of electronics. This method can also be used for test sequencing problems related to the manufacturing of complex systems.

System test problems are multidisciplinair (e.g. electronics, software and mechanics), large (hundreds of tests) and take a long time (up to several weeks/months). A test and integration strategy for systems is traditionally created by experts which have a good knowledge of the systems architecture, the risks and the test costs. Test sequencing and selection is traditionally a risk-based decision. That is, the elements with the highest risk are tested, until time is up and the system is shipped. At that moment, the quality of the system is often unknown.

The semiconductor industry is a typical example of a time-to-market driven industry. For companies such as ASML, shipping your system before competition is wanted, and thus dominates the test and integration phase. Several cases within the manufacturing process of a TWINSCAN machine are presented in this article.

The structure of the article is as follows: first an example test problem is introduced, then the test problem is formally described, then different solving algorithms are mentioned, then the results of the different cases are shown, and finally conclusions and future work are mentioned.

[1]This work has been carried out as part of the TANGRAM project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under grant TSIT2026.

Example test problem

To illustrate a system test problem, a telephone is taken as system under test. This telephone consists of three modules: the device, the receiver and the cable connecting the receiver and the device. The system is shown in Figure 1. There are two interfaces between the modules: one between the device and the cable and one between the cable and the receiver.

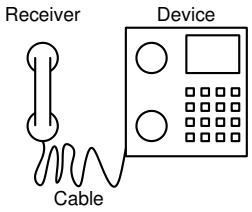


Figure 1: Telephone example

In this system under test, we can identify 5 possible faults:

1. The device is broken.
2. The cable is broken.
3. The receiver is broken.
4. The cable cannot be connected to the device.
5. The cable cannot be connected to the receiver.

The first three faults are logical, the last two may be less obvious. These two faults are interface faults, which are typical system faults that occur through concurrent engineering. All modules have been developed in parallel using interface specifications. If these specifications are ambiguous, the assembled system may not work as the specifications are interpreted differently for each module, which results in interface faults. Each fault has a certain probability that it exists. It is assumed that this fault probability is 10% for each fault.

The goal of testing the system is to find out which of the possible faults exists. 6 tests are available to test this system:

0. Test the complete telephone
1. Test the device
2. Test the cable
3. Test the receiver
4. Test the device and cable
5. Test the cable and receiver

The costs of each test are defined in uniform *cost units*. In real life, these costs can for example be defined in money or in time. Test 0 costs 3, while tests 1,2 and 3 each cost 1 and test 4 and 5 cost 2.

The objective is to create a test sequence with minimal expected test costs. This optimal sequence logically depends on the outcomes of tests applied, as illustrated in Figure 2. According to this test sequence, a tester starts with test 0. If this test passes, the tester knows no fault exists in the system and the system works. If this test fails, the tester knows that at least one fault exists and the tester has to perform more tests to identify this fault. This way of working results in a test tree, which contains several test sequences depending on the outcomes of tests. The objective of calculating the optimal test sequence actually means calculating the optimal test tree with minimal expected test costs, identifying each possible fault.

The test costs of a test tree can be calculated as described in the sequel for the example test tree of Figure 2.To start with, test 0 is performed. This test fails with a certain probability and if so test 3 is performed next. This probability depends on the covered faults and their probabilities. The expected test costs are therefore the test costs of test 0 plus the test costs of test 3 multiplied by the chance that test 0 fails, and so on. An optimal solution is a tree with the least expected test costs. An optimal solution for the telephone example is shown later in this article. We continue in the next section with a formal description of the test problem.

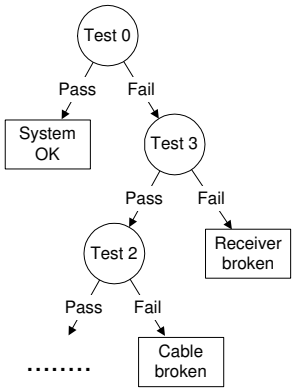


Figure 2: Test tree with multiple test sequences

Test problem formulation

Formally, a test problem $\mathcal{D}$ can be defined as a five-tuple: $\mathcal{D} = (\mathcal{T}, \mathcal{S}, \mathcal{T}_c, \mathcal{S}_p, \mathcal{R}_{ts})$, where:

- $\mathcal{T}$ is a finite set of *k* elements, called tests.
- $\mathcal{S}$ is a finite set of *l* elements, called fault states.
- $\mathcal{T}_c : \mathcal{T} \rightarrow \mathbb{R}$ gives for each test in $\mathcal{T}$ the associated costs of performing that test
- $\mathcal{S}_p : \mathcal{S} \rightarrow \mathbb{R}$ gives for each fault state in $\mathcal{S}$ the *a priori* probability that the fault state is present.
- $\mathcal{R}_{ts} : \mathcal{T} \rightarrow \mathcal{P}(\mathcal{S})$ gives the subset of fault states that are covered by a test.

The *a priori* probability is the absolute probability that a certain fault is present. The test problem can also be represented as a matrix A of dimensions $l \times k$, where $A_{ij} = 1$ if test t_j covers fault state s_i, otherwise $A_{ij} = 0$. The formal description is a model of the test problem and is therefore called the *system test model*. In Table 1, the system test model of the telephone example is shown, represented as a matrix.

Table 1: Telephone example system test model

$\mathcal{S}$ / $\mathcal{T}$	t_0	t_1	t_2	t_3	t_4	t_5	$\mathcal{S}_p$
s_1	1	1	0	0	1	0	10 %
s_2	1	0	1	0	1	1	10 %
s_3	1	0	0	1	0	1	10 %
s_4	1	0	0	0	1	0	10 %
s_5	1	0	0	0	0	1	10 %
$\mathcal{T}_c$	3	1	1	1	2	2	

In the following sections, different algorithms are discussed to solve the test problem and hence calculate the optimal test tree with minimal expected test costs.

Solving algorithms

Continuing on the work of Pattipati, many different solving algorithms using different heuristics have been developed. A good overview is given by Shakeri *et al* in [1]. The assumptions of the test problem solving algorithms are:

- binary outcome tests (only pass or fail),
- the fault states are independent of each other,
- the tests are 100% reliable,
- the tests are 100% sensitive and specific,
- a repair action 100% fixes the fault state.

The test problem solving algorithms consists of two types: single and multiple-fault algorithms. The single-fault algorithms have the assumption that at most one fault state is present. The multiple-fault algorithms do not have that assumption. Both types of algorithms are explained in the sequel.

Single-fault algorithms

The single-fault algorithm has the assumption that at most one fault state exists. This assumption results in some changes to the original test problem. The possibility that no fault state exists (the system is OK) must be modelled explicitly because the algorithm assumes that at least one fault is present. This is done by adding an extra state to $\mathcal{S}$, named s_0 which represents the system OK state. Element $\mathcal{S}$ of the basic test problem is denoted by $\underline{\mathcal{S}}$ for the single-fault problem. Also, because at most one fault state can be present, the sum of the fault state probabilities must be 100%. Therefore, the *a priori* fault probabilities $\mathcal{S}_p$ are converted to *conditional* fault probabilities $\underline{\mathcal{S}}_p$ using,

$$\underline{\mathcal{S}}_p(s_0) = \frac{1}{1 + \sum_{s \in \mathcal{S}} \frac{\mathcal{S}_p(s)}{1 - \mathcal{S}_p(s)}} \tag{1}$$

and

$$\underline{\mathcal{S}}_p(s_i) = \frac{\frac{\mathcal{S}_p(s_i)}{1 - \mathcal{S}_p(s_i)}}{1 + \sum_{s \in \mathcal{S}} \frac{\mathcal{S}_p(s)}{1 - \mathcal{S}_p(s)}} \text{for } i = 1, \cdots, l. \tag{2}$$

The single-fault system model of the telephone example is shown in Table 2.

Table 2: Telephone example single-fault system test model

$\underline{\mathcal{S}}$ / $\mathcal{T}$	t_0	t_1	t_2	t_3	t_4	t_5	$\mathcal{S}_p$	$\underline{\mathcal{S}}_p$
s_0	0	0	0	0	0	0	-	64.28%
s_1	1	1	0	0	1	0	10%	7.14%
s_2	1	0	1	0	1	1	10%	7.14%
s_3	1	0	0	1	0	1	10%	7.14%
s_4	1	0	0	0	1	0	10%	7.14%
s_5	1	0	0	0	0	1	10%	7.14%
$\mathcal{T}_c$	3	1	1	1	2	2	-	100%

A solution to the single-fault test problem is an AND/OR decision tree as shown in Figure 3. This tree consists of three types of nodes: AND, OR and leaf nodes. The OR nodes represent the suspected

set of fault states, the AND nodes represent tests applied to the OR nodes and the leaf nodes represent isolated faults states.

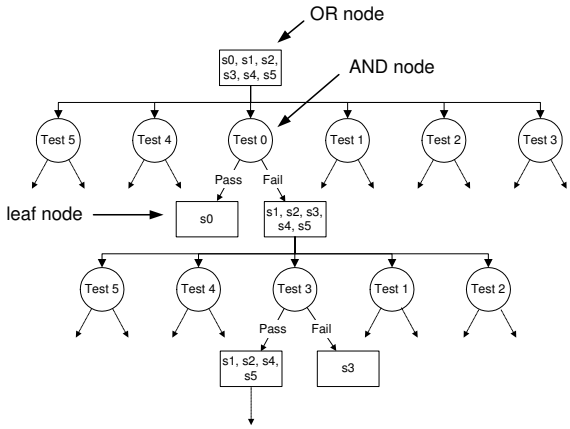


Figure 3: An AND/OR graph

Calculating an optimal AND/OR tree is NP-Hard [5]. Therefore in literature, two types of solving algorithms are described: optimal algorithms for small and near-optimal algorithms for large test problems.

To calculate an optimal AND/OR tree, two optimal algorithms can be used: Dynamic Programming and AND/OR graph search [5]. The Dynamic Programming technique is a recursive algorithm that constructs an optimal tree from the leave nodes up by identifying larger subtrees until the optimal tree is generated. The Dynamic Programming technique has storage and computational complexity of $\mathcal{O}(k^3)$ for the basic test problem. Therefore in this article, we use the more efficient top-down algorithm based on AND/OR graph search (AO^*).

The AO^* algorithm constructs an AND/OR graph as a directed graph with a root (or initial) node and a nonempty set of terminal leaf nodes. The initial node represents the given problem to be solved, while the terminal leaf nodes correspond to the subproblems with known solutions. An OR node is solved if any one of its successor nodes is solved, and an AND node is solved only when all of its immediately successors are solved. During the search within the AND/OR graph, the expected test costs of visited OR nodes are saved to reduce computational effort: these costs do not have to be calculated again.

For larger problems near-optimal algorithms are necessary. Several near-optimal search algorithms are known from literature [5], for example: the AO^*_ϵ algorithm, the limited search AO^*, and the AO^* algorithm combined with a multi-step information gain heuristics. The near-optimal one-step information gain heuristics can be used during the AO^* algorithm to solve the larger cases presented in this article.

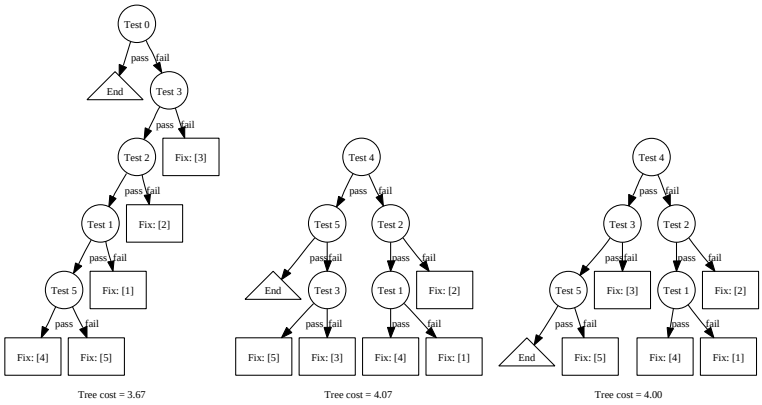
The test tree shown in Figure 4(b) is an optimal tree for the telephone example. The expected test cost of this tree are 4.07. This means that on average, 4.07 test cost are necessary to identify one fault state in the system.

To illustrate the different test sequences that can be found for different fault probabilities, we reduce the *a priori* fault chance of each fault state from 10% to 5%. The resulting tree can be seen in Figure 4(a). In the third situation, the *a priori* fault chance of each fault state is 50%. The resulting tree can be seen in Figure 4(c). In the 5% situation, only test 0 is necessary to check whether the system is OK, in the 10% situation both tests 4 and 5 are necessary to check whether the system is OK, while in the 50% situation tests 4, 3 and 5 are necessary to check whether the system is ok.

Multiple-fault algorithms

When fault probabilities are high, the assumption that at most one fault state is present in the system is questionable. In these cases, it is still possible to use the solution tree of the single-fault algorithm, over and over again until all fault states have been identified, but it is certainly not optimal. Therefore multiple-fault algorithms are necessary.

Multiple-fault algorithms construct AND/OR graphs in the same way as the single-fault algorithms. However, instead of considering one possible fault state, they consider all possible combinations of fault states. The OR node in an AND/OR graph represents all possible subsets of suspected fault states. Multiple-fault problems have a exponential complexity of $\mathcal{O}(2^l)$ (see [1]). The AO^* multiple-fault algorithm used in this article, is derived from the AO^* single-fault algorithm. Compared to the single-fault algorithm, the multiple-fault algorithm considers fix actions of fault states. If a fault state is isolated, it can be fixed immedi-



(a) 5% fault probability (b) 10% fault probability (c) 50% fault probability

Figure 4: Telephone example optimal single-fault test trees

ately. After the fix action, isolated fault states are removed and more tests are applied to find other faults. The algorithm terminates when all faults are excluded and the system is ok. The resulting graph has one root node and one leaf node. An example multiple fault tree is shown in Figure 5.

Besides test and fix actions, the algorithm also has diagnosis actions. If a number of fault states is under suspicion, but none have been isolated and additional testing does not give more information, a diagnosis action removes the suspected fault states. This diagnosis action has high costs, but is necessary to terminate the algorithm and solve the test problem.

To reduce computational complexity, the same information gain heuristic is implemented as in the single-fault algorithm. Most computational costs are spent during the calculation of the pass and fail probabilities of a test, as all subsets of fault states must be taken into account. Therefore, estimators are used to estimate the pass and fail probabilities and reduce this computational complexity. If a problem is still to large, it can be divided into subproblems that can be solved optimal or near-optimal. The subproblems by itself can then be sequenced with the same algorithm, or by hand. To reduce storage complexity of saved OR nodes, the implemented multiple-fault algorithm uses the compact set notation (see [1]). The compact set notation is a shorter notation for all possible subsets of fault states.

An optimal multiple-fault tree of the telephone example, shown in Figure 5, has been calculated with the optimal multiple-fault algorithm.

Tree simulation

Both single and multiple-fault algorithms can be used for system test problems. The advantage of a single-fault algorithm is that the resulting tree is smaller and better understandable. Also, the computational effort is less. However, the resulting test costs may be higher than in case of using solution from a multiple-fault algorithm. By using a simulation model of the test process, called the testFactory, the difference between the average test costs can be made clear. The testFactory is not discussed in this article. The testFactory simulates the testing of a number of predefined faulty systems either using a single-fault tree over and over again until all faults are found, or using the multiple-fault tree. In Figures 6(a) and 6(b) two histograms are shown of the simulation of the single and multiple-fault 10% fault probability trees. After 5000 simulation runs (number of systems tested), the average test costs of the single-fault tree were 5.7, while the average test costs of the multiple-fault tree were 5.3. If all tests would be performed, the test costs would be 10.

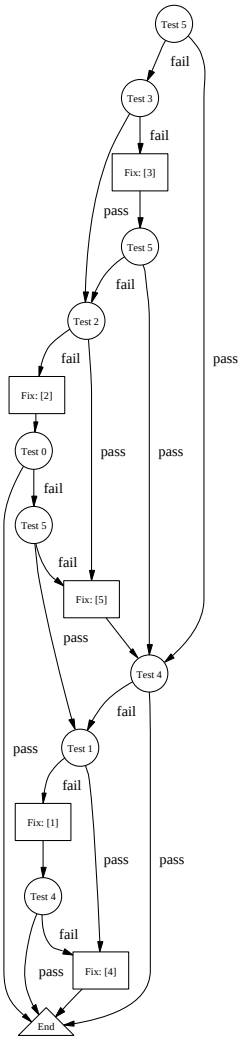


Figure 5: Telephone example optimal multiple-fault test tree

Cases

Within the manufacturing department at ASML, several test steps are performed during the production of a TWINSCAN lithographic machine. These test steps consist of performance, measurement and fault-detection tests, and calibrations. The presented test sequencing method is applied to three test steps, called job-steps, of different modules to reduce the cycle time of manufacturing a TWINSCAN machine.

The approach of the case is as follows:

1. Three models are created for 3 different job-steps.
2. For each model the optimal single and multiple-fault test trees are calculated.
3. The resulting test sequences are simulated using a test factory simulation model to show the expected test time.

In Table 3 the properties of the 3 created models are shown. The first column denotes the size of the matrices for job-steps A, B and C. The second column denotes the sum of all test costs, denoting the current situation. The cost of a test is for this case defined in time units. The third column shows average fault probability. The fourth column indicates the density of the A matrices, that is, how well 'filled' these matrices are.

Table 3: Case system test model properties

Case	$k \times l$	$\sum_{t \in \mathcal{T}} \mathcal{T}_c(t)$	Aver.($\mathcal{S}_p$)	Dens.(A)
A	15×15	815	71.3%	38.2%
B	33×60	33	46.0%	15.2%
C	39×73	730	15.8%	10.4%

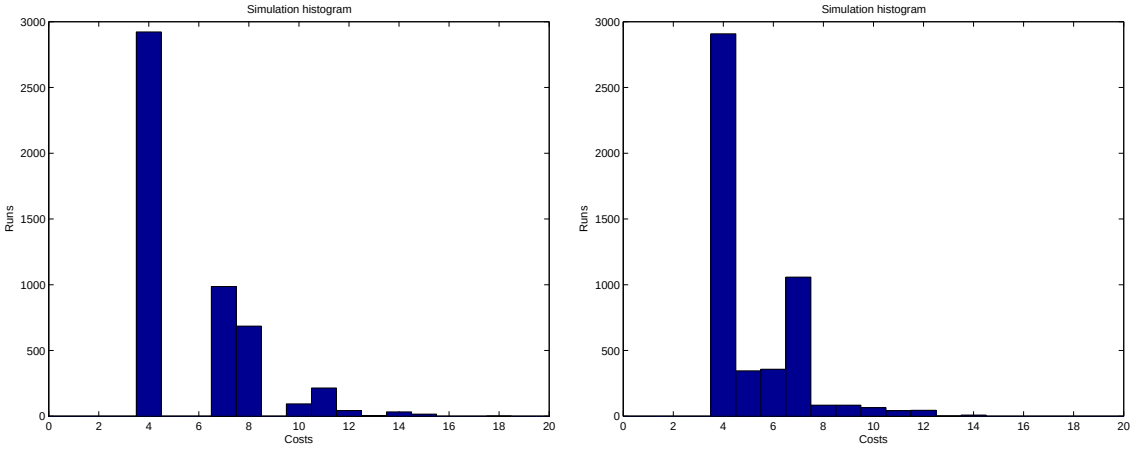
Now, the single and multiple-fault trees can be calculated. In Table 4, the properties of the trees and algorithms used are shown. The first single-fault column denotes which methods have been used to solve the single-fault problem: either the optimal calculation or using the information gain (IG) heuristic or by dividing (div) the problem in multiple problems. The second column shows the expected tree costs. The same columns are shown for the multiple-fault algorithm.

The costs of the single-fault trees are much lower then the multiple-fault trees. This due to the single-fault assumption and the conditional probabilities which are much lower in these cases then the *a priori* fault probabilities.

Table 4: Case test tree properties

	Single-fault		Multiple-fault	
Case	Method	Costs	Method	Costs
A	Optimal	202.9	IG	690.0
B	IG	5.0	div(4)	25.8
C	Optimal	144	div(4)	504

After the trees have been calculated, they are simulated in the simulation environment, as mentioned previously. In Table 5, the simulation results are shown. The first column shows the average single-fault tree costs and the second column shows the gain or loss in cycle time compared to the current situation. The third and fourth columns show the same for the multiple-fault test trees.



(a) Single-fault simulation (5.7 average)

(b) Multiple-fault simulation (5.3 average)

Figure 6: Telephone example tree simulation histograms

The average test costs of the single-fault tree are much higher then the test costs of the current situation. This results from the assumption that only one fault exists, while the average number of faults present is large (larger then 10). The single-fault tree is therefore only suitable when the average number of faults is small, in the range of 1 through 5. For a larger number of average faults, the multiple-fault algorithm performs better. The resulting test trees are even better then the test trees that are currently used.

Table 5: Case simulation results

	Single-fault		Multiple-fault	
Case	Sim.	Delta	Sim.	Delta
A	1848	+126%	689.0	−15.5%
B	60.8	+84.4%	25.9	−21.5%
C	1608	+120%	504.2	−30.9%

Conclusions

The presented method describes the test problem in a system test model. A single-fault algorithm calculates an optimal, with the least test costs, test tree, consisting of multiple test sequences, based on this system test model. This algorithm has the assumption that at most one fault exists. Besides this algorithm, a multiple-fault algorithm is described that creates a test tree with the assumption that multiple-faults can exist. This algorithm needs to take fixing and diagnosis of faults into account. The single-fault algorithm needs few computation to give an optimal solution, however it is recommended that this solution may only be used with test problems that have no more then 5 faults on average present in the system. The multiple-fault algorithm takes more computation effort, but the calculated solutions can also be used with problems that have more then 5 faults present.

We can conclude that the presented method is suitable for system test problems, as seen within ASML. There are two main benefits for using this method in the test and integration phase of systems. First, the test cost can be reduced by calculating the optimal test sequence as is shown in this case. Even test sequences that are judged to be quite good by experts, can be improved and cycle time can therefore be reduced.

Second, more insight in the test coverage of faults is gained when creating system test models. For large systems little knowledge exists about the relation between faults and tests: if a test fails it is difficult to indicate why. The presented system test model is a summary of these relations. Furthermore, the available test set can be made more explicit. New tests can be developed that cover faults which are not covered by the current test set. Also new tests can be developed that replace multiple other tests but cover the same or even more faults.

Future work

In the sequel of this project we will continue developing methods to optimize test and integration strategies. Test and integration sequences depend on each other. For example, the telephone consists of three modules. If the development of a certain module is delayed, tests using this module cannot be performed, while tests concerning the other two modules can be performed. Also, if the modules are separated, parallel testing would be possible, which probably reduces test time. In other words, the integration sequence of modules must be taken into account to determine the optimal test sequences. Or even further: the integration sequence must be optimized regarding time, costs and/or quality. Other aspects of the test and integration strategy relevant to our project are: scheduling tests over resources, strategy decisions regarding cost, time and quality and as already mentioned in this article, test and integration process simulations to determine the difference between certain test and integration strategies.

References

[1] M. Shakeri, V. Raghavan, K. R. Pattipati and A. Patterson-Hine, *Sequential Testing Algorithms for Multiple Fault Diagnosis* In *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, Volume 30, Number 1: 1-14, 2000

[2] ASML, `http://www.asml.com`

[3] M. Prins, *Testing Industrial Embedded Systems - An Overview* In *Proceedings of the 14th Annual International Symposium of INCOSE*, 2004

[4] K. R. Pattipati, S. Deb, R. W. Dontamsetty and A. Maitra, *START: System Testability Analysis and Research Tool* In *IEEE Aerosp. Electron. Syst. Mag.*: 13-20, 1991

[5] K. R. Pattipati and M. G. Alexandridis, *Application of heuristic search and information theory to sequential diagnosis* In *IEEE Trans. Syst. Man, Cybern.*, Volume 20: 872–887, 1990

Contact Information

Roel Boumen

Technische Universiteit Eindhoven
Department of Mechanical Engineering
P.O. Box 513, 5600 MB Eindhoven
The Netherlands
r.boumen@tue.nl

Ivo de Jong

Technische Universiteit Eindhoven
Department of Mechanical Engineering
P.O. Box 513, 5600 MB Eindhoven
The Netherlands
i.s.m.d.jong@tue.nl

Model-based testing with χ and TorX [1]

A case study of the ASML laser subsystem

Niels Braspenning, Asia van de Mortel-Fronczak, Koos Rooda

Within the TANGRAM *project, a case study on model-based testing of the* ASML *laser subsystem has been performed. The approach used in the case study is based on the proposed model-based testing framework, instantiated with state-of-the-art tooling from the* TANGRAM *project partners:* χ *as specification language and* TorX *as test tool. A* χ *specification model of the laser state behavior and communication interface has been developed. After verification and validation, the model has been used for automatic model-based testing with* TorX*. Using this approach, discrepancies between the implementation and specification of the laser subsystem have been found.*

One research topic of TANGRAM is model-based testing (MBT in short), that has already been a topic of the XOOTIC MAGAZINE [1]. In model-based testing, the behavior specification of a system under test is given by a formal model, which is a precise, complete, consistent, and unambiguous basis for testing. Using formal specifications for testing enables automatic processing by means of tools. Using a test derivation algorithm implemented in a test tool, test cases are automatically derived from the specification model and executed on the system.

One of the TANGRAM case studies concerns model-based testing of the ASML laser subsystem using the specification language χ and the test tool TorX. The objectives of this case study are to show the applicability of automated model-based testing using TorX within ASML, to show that χ models can be used for model-based testing, and to investigate the limitations and shortcomings of the approach used.

MBT framework

The proposed MBT framework is shown in Figure 1 and consists of the following elements:

- An *informal specification* of the correct behavior of the system under test expressed in a natural language (documentation) and present in the minds of the designers (mental model).
- A *(formal) specification model* of the correct behavior of the system under test expressed in an unambiguous specification language.
- A *formal test model* of the correct behavior of the system under test expressed in a test formalism that is suitable input for the test tool. Note that the specification model and the test model can be (but are not necessarily) the same.
- A *test tool* that is able to automatically derive tests from the test model, to execute these tests on the system under test, and to compare the test results with the test model behavior.
- A *test environment* that provides access to the interfaces of the system under test and enables stimulation and observation of these interfaces.
- A *system under test (SUT)*, which is the actual implementation that is tested together with the required context that is needed for testing.

[1] This work has been carried out as part of the TANGRAM project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under grant TSIT2026.

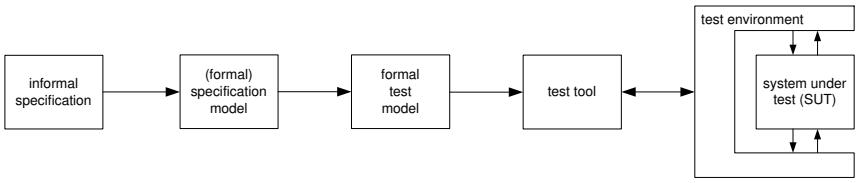


Figure 1: Model-based testing (MBT) framework

Tooling

When testing is to be performed automatically, some form of tooling is required. Looking at the MBT framework from Figure 1, the following tooling is needed:

- A specification language and test formalism in which the correct system behavior and the required test aspects can be expressed. The test formalism must be suitable input for the test tool.
- A test tool that is able to automatically derive tests from the test model using a test derivation algorithm and that is also able to automatically execute the derived tests on the SUT and compare the test results with the test model behavior.
- A test environment that connects the test tool to the SUT and enables stimulation and observation of interfaces of the SUT.

Looking at the TANGRAM project partners, good candidate tools for model-based testing would be χ [2] and TORX [3]. Within the Systems Engineering Group at the Eindhoven University of Technology, there is a lot of experience on the modeling, analysis, control, and optimization of manufacturing systems with the specification language χ, for both discrete-event and hybrid (i.e. including continuous behavior) systems. Using the high expressivity of χ for model-based testing will be beneficial in the future when the testing domain is extended towards time, data, and hybrid testing, because the current test formalisms are not expressive enough (discrete-event only) for specifying these other aspects.

The test tool TORX, developed at the Formal Methods and Tools research group at the University of Twente, is able to derive and execute tests on-the-fly, based on the *ioco* theory. Several case studies show successful application of the tool. Test formalisms that are currently supported by TORX are LOTOS and TROJKA, the latter one being a slightly adapted version of PROMELA [4]. The test domain of TORX is currently limited to the discrete-event domain, however extensions towards the data, time, and hybrid test domain are investigated within the TANGRAM project and other projects.

As it is a specific goal of the laser case study to investigate whether χ can be used for model-based testing, χ is chosen as specification language. However, χ cannot be used as a test formalism, as it is not a suitable input format for TORX. Because of this, and the fact that a direct connection between χ and TORX is considered as a future development, one of the supported test formalisms of TORX has to be selected to which the χ specification will be translated. Because of the resembling structures of χ and PROMELA and the existing experience in translating χ to PROMELA, TROJKA is chosen as test formalism for the laser case study. The usage of PROMELA also allows verification of certain properties of the model with the model checker SPIN.

Table 1: Properties of χ, PROMELA, TROJKA

Language property	χ	PROMELA	TROJKA
Simulation	√	√	√ (closed)
Verification	X	√	√ (closed)
Testing	X	X	√ (open)
Modeling expressivity	☺	😐	😐
Data	√	X	X
Functions	√	X	X
Time	√	X	X
Stochastics	√	X	X
Hybrid	√	X	X
Easy to modify	😐	☹	☹

The three specification languages mentioned above, χ, PROMELA, and TROJKA, are compared to each other according to certain properties in Table 1.

For the test environment, the current developments within TANGRAM on test infrastructure, also addressed in this XOOTIC MAGAZINE issue, are used, which provides easy access to the interfaces of ASML software components.

Case: ASML laser subsystem

For each exposure of an area (e.g. one chip) on a silicon wafer in a wafer scanner a beam of laser light is needed, that is provided by the laser subsystem. The laser subsystem is manufactured by another company than ASML, and has to operate together with the ASML wafer scanner to get good exposure results. To this end, a lot of communication is used between the scanner and laser, like commands, queries and responses, warnings and errors, control data, timing and synchronization triggers.

One condition of the case study is that only functional, untimed behavior is considered, so only the communication concerning commands, queries, and responses is taken into account. Although there are multiple (serial and parallel) communication interfaces between the wafer scanner and the laser, only the RS232 serial interface is used in the experiments, because this interface is easily accessible through the test environment.

Taking these limitations (functional behavior using the serial interface) into account and looking at the operational sequences in the laser subsystem documentation, the number of serial commands that can be tested is very limited. Many operational sequences use parallel commands (i.e. commands sent over the parallel communication interface) or can only be executed in the 'expose' state that requires parallel commands to reach. Therefore, only the laser state behavior (serial commands starting with 'LS') is considered, which limits the testable functionality to changing the laser state to standby and off, and to query the current state only. Nevertheless, this is still enough to show proof of concept.

Approach

From the informal specification in the form of documentation and mental models (revealed by talking with the ASML people involved), a χ model of the laser and an environment of the laser (necessary to get a *closed* system) is developed. To gain confidence in the model, the model is verified and validated against the informal specification by means of simulation.

Subsequently, the χ model is translated into PROMELA. Because PROMELA has several limitations concerning the modeling expressivity of χ, workarounds have to be found for the translation of certain χ model constructs. By means of simulation with SPIN, the PROMELA model is verified and validated against the informal specification and the simulation results of the χ model. Besides that, several model properties are verified using SPIN.

When there is enough confidence in the model, the PROMELA model is converted into the TROJKA test model, which involves only a few small modifications. It is important to mention that now only the laser part is converted, because testing is done using a system specification *without* environment (i.e. an *open* system). Finally, when TORX is connected to the TROJKA test model on one side and to the test environment that accesses the RS232 serial interface of the laser on the other side, the testing experiments are performed.

It is important to mention that for the first experiments, a hardware laser simulator (containing programmable electronics and cable connectors for the actual serial and parallel communication interfaces) is used instead of a real laser due to costs and safety issues. This hardware laser simulator is connected by cables to an ASML test rack, which is controlled by software. The approach of the laser case study, which is an instantiation of the MBT framework from Figure 1, is visualized in Figure 2.

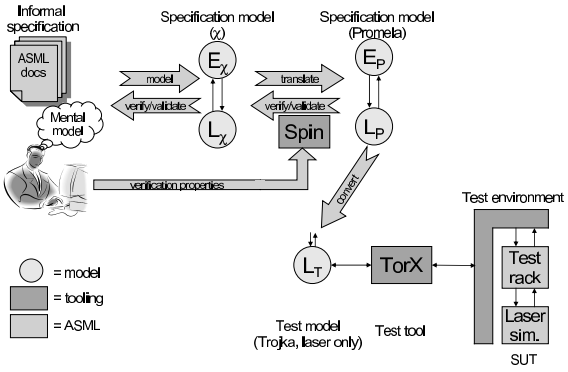


Figure 2: Case study approach

Modeling in χ

The χ specification model of the laser subsystem contains both the environment side and the laser side of the (serial and parallel) communication interface. The χ model, depicted in Figure 3, contains the following processes, which are interconnected by channels:

- The *environment* `Env` closes the system and can be configured (using an external configuration file) to generate specific command sequences for behavior validation, for example the operational sequences of the wafer scanner (as found in documentation).
- The *I/O interface* `IO` interfaces with the environment and passes through commands and responses to and from `LC` and `LS`.
- The *laser communication* `LC` process handles the commands from the environment (passed through by `IO`), performs the necessary actions (e.g. a state change), and creates the responses corresponding to the configuration that is loaded from an external file.
- The *laser state* `LS` process keeps track of the current laser state, in case the environment queries the current state.

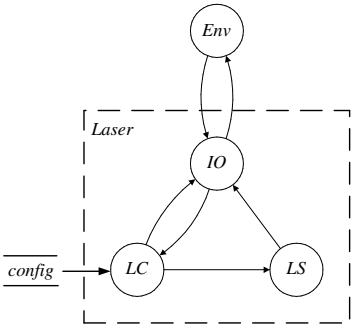


Figure 3: Processes and channels of the χ model

The χ model is *configurable* in a sense that the environment command sequences and the laser behavior can be changed easily in external files without changing and recompilation of the χ model itself. This easy changing of behavior already showed its advantage when it became clear that a certain laser type was not available in the laser simulator and another laser type had to be specified. Furthermore, the model contains *error handling* of 'unknown' commands (commands not understood by the laser) and 'bad context' commands (known commands that are not allowed in a certain state).

Figure 4 shows the behavior of the laser model that is to be tested by TORX. In this figure, the nodes depict the states of the model and the edges depict both commands/input (solid) and results/output (dashed). The central states at the top and bottom denote the actual laser states 'off' and 'standby' (numbered '00' and '03', respectively). The 'trans', 'error', and 'query' states are intermediate states between different LS (laser state) commands. Note that a state transition command to the current state results in a 'bad context' error ('??=02').

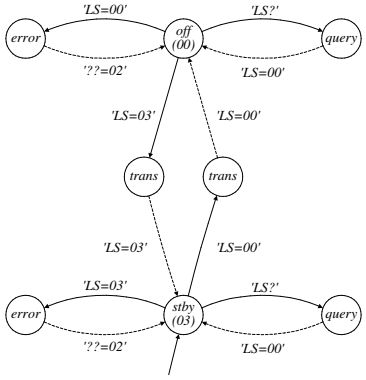


Figure 4: Laser behavior to be tested

The verification and validation of the χ model is performed by means of simulation. Several interesting scanner command sequences (e.g. operational sequences from documentation and bad weather (exceptional) behavior) are generated in process `Env` and the model is simulated. The simulation results show the same behavior as in the documentation and also the error handling functionality behaves as expected.

Translation to PROMELA

Because χ is not a suitable input for TORX, the χ specification model has been translated to PROMELA by hand, which is a laborious and error-prone task. For most of the χ constructs, a translation scheme from χ to PROMELA, developed in the TIPSY project [5], can be used. However, some specific χ constructs cannot be directly translated, for example lists, sets, (repetitive) selective waiting, and functions (e.g. the `pick` function to select one element from a set). For these cases, workarounds have been found and applied. As the translation is done manually according to some translation scheme, it is certainly not guaranteed that the translation is correct. Nevertheless, the resulting PROMELA model resembles the χ model as much as possible, which means that each statement in χ is translated into one PROMELA statement

or into one block of PROMELA statements that is preferably considered as one internal action (by using the `atomic` and `d_step` operators). The resulting PROMELA code is certainly not optimal and not the most efficient, which is due to the translation from χ.

Modeling the laser subsystem in PROMELA right away would probably result in a more elegant model, so in this case the benefits of using χ may not be really clear. However, one of the objectives of this case study was to investigate the possibility of using χ for MBT, and in this case the usable functionality of χ is limited to the functionality that is supported by PROMELA and TORX. So, the experience gained in this case study is beneficial when data, time and hybrid aspects are to be included, which are supported in χ, but not in PROMELA.

Table 2: Model properties of χ, PROMELA, and TROJKA

Model property	χ	PROMELA	TROJKA
Environment process Env	√	√	X
Laser processes IO/LC/LS	√	√	√
Serial interface	√	√	√
Parallel interface	√	√	X
Error handling	√	√	√
Configurable behavior	☺	☹	☹
#lines for model	350	800	350
Time to build	3 weeks	+3 weeks	+1 week

Verification and validation with SPIN

Just like the χ model, the translated PROMELA model is also verified and validated by performing simulation runs, in this case with the model checker SPIN. Again, several operational sequences and bad weather command sequences are generated in the environment and the results are as expected.

An advantage of having a specification model in PROMELA, is that SPIN can be used to verify certain model properties. Several generic properties like *deadlock freeness* and *no unreachable states* are successfully verified. Besides that, also some specific properties of translated χ constructs and of the laser behavior are verified and found to be correct, for example that:

- the PROMELA translation of the χ function `pick` (which takes an element from a set) always returns one set element;
- only certain state transition sequences are allowed;
- only certain replies are allowed to a command.

Conversion to TROJKA

With verification and validation, the confidence in a model grows. When there is enough confidence in the model, it is used for model-based testing. To this end, the PROMELA model needs to be slightly modified, which results in a TROJKA model that is suitable input for TORX. First of all, the TROJKA model is an *open* system, i.e. it does not contain the `Env` process from Figure 3. Another difference is that in a TROJKA model the channels that are *observable* to the outside world need to be defined, which is done by giving them the special attribute `OBSERVABLE`. Finally, the channel names have to conform to a certain naming convention to enable the connection of TORX to the system under test through the test environment.

Corresponding to Table 1 that shows properties of the specification languages χ, PROMELA, and TROJKA, a similar overview of laser model specific properties is given in Table 2.

Testing with TORX

Now that the specification side of the MBT framework (all elements on the left of the test tool in Figure 1) has been set up, the test tool has to be connected to the SUT. For the translation of the abstract commands from the TROJKA test model into the concrete commands of the SUT and vice versa, an adapter component (implemented in PYTHON) is used.

For each observable channel in the test model, a PYTHON adapter function has been created that handles the connection to the SUT, which involves translation from abstract commands into real commands, wrapping of specific command data (e.g. a left justified string of 128 characters). The other way around, also the real replies received from the SUT have to be unwrapped and translated back into the abstract replies as specified in the test model.

System under test: laser simulator

As already mentioned, a hardware laser simulator is used as system under test instead of the real laser due to safety and costs issues. This laser simulator is connected to a software controlled test rack and is developed by ASML to be able to test the wafer scanner software and electronics in the test

rack without a real laser connected to it. This saves a lot of expensive cleanroom time and is less dangerous. As the ASML wafer scanners are shipped to customers with different laser types, also the laser simulator can be configured for several (but unfortunately not all, as we experienced) laser types.

With a configured laser simulator, connected by cables to the ASML test rack where the software, electronics, and the test environment are up and running, the whole test setup as shown in the right bottom part of Figure 2 (consisting of the TROJKA test model, the TORX test tool, the test environment, and the laser simulator as SUT) is prepared for experimenting.

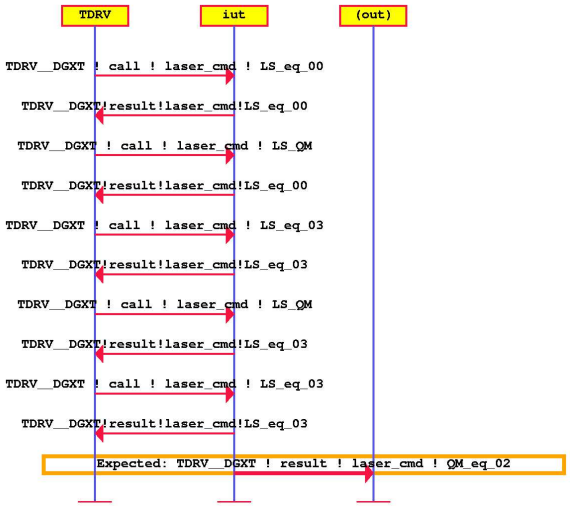


Figure 5: TORX has found a discrepancy!

Experiments and results

With the test setup as described above, the laser simulator has been tested automatically. Serial commands are selected from the TROJKA test model by TORX and sent to the laser simulator. The responses from the laser simulator are observed and compared with the behavior specified in the model. During the experiments two major discrepancies between the test model and SUT concerning state behavior have been found. One of these discrepancies is discussed in more detail below.

The specified laser behavior from Figure 4 shows that a state transition command to the current state (e.g. giving the command 'LS=03' in the 'standby (03)' state) should give a 'bad context' error ('??=02') as reply. However, the laser simulator replies with the current laser state instead ('LS=03' in this case). The TORX message sequence chart in Figure 5 shows the commands and replies leading to this discrepancy. Because the signs '=' and '?' are not allowed in PROMELA, they are replaced by 'eq' and 'QM', respectively.

Besides discrepancies in the implementation, also some errors and inconsistencies in the specification documents are found. Due to the general explanation in words, these specifications are incomplete, they can be interpreted in different ways, and sometimes they are even conflicting. Especially the specification of bad weather behavior (if it is specified at all) is not clear. For example, a lot of (operational) command sequences are specified separately, but nothing is explicitly stated about the remaining (e.g. bad weather) command sequences. Even if it is possible, it is very hard to extract this information from the informal specification. When making a specification model, the specification language explicitly forces a complete specification of all possible cases, for example in an if-elseif-else construct.

Conclusions

With the laser case study, a proof of concept is delivered that automatic model-based testing with TORX can be applied within ASML. Furthermore, it is also shown that χ models can be used for model-based testing. In this case the χ model is not directly used for model-based testing, however the structure of the χ model is maintained during the translation into the PROMELA and TROJKA models.

Developing a formal specification model starting from an informal specification is a difficult task, especially when a modeler is new to the system. The information is scattered over different documents, can be interpreted in different ways, is incomplete, and in some cases it is conflicting. Moreover, it is possible that parts of the informal specification are not documented, but stored in the minds of the designers (mental models). Therefore, talking to the people involved is very important to clear confusion, to reveal the mental models, and to validate your specification model.

The often heard argument that modeling a system takes a lot of time is not completely true. It is not the modeling (i.e. writing the specification down in some specification language) itself that takes a

lot of time, but the development of an *unambiguous specification*. The act of modeling itself forces the modeler to think harder about the system specification, which will result in a better understanding of the system and also in a more complete and less ambiguous specification.

Concerning the specification model of the laser subsystem in χ, the modeling is done according to the current way of working within the Systems Engineering Group at the TU/e. The configurability of the model can be considered as a new way of specifying behavior.

The translation from χ to PROMELA is a very laborious and error prone process that results in a loss of modeling expressivity, readability, and modifiability. Additionally, there is no certainty about the correctness of the translation, as it is done by hand.

One question that can be asked is whether it is beneficial to start modeling with χ instead of modeling directly in PROMELA. Currently the used functionality of χ is limited to what is possible with PROMELA and TORX, i.e. functional testing of discrete-event systems. The expressive modeling power of χ is yet untouched and all functionality that is used in the χ model is maintained in PROMELA and TROJKA (but certainly not in an optimal way). So for this case study, it would be reasonable to start modeling in PROMELA right away. However, when the data, time and hybrid test domain come into the picture (which will be the case in the near future), PROMELA will not suffice any more. Then the project can benefit from using χ and, therefore, this initial case study is useful and valuable for future research.

The approach described in this report enables automatic testing of the responses of the laser simulator, for both good and bad weather. The initial experiments concerning the laser state behavior tested limited functionality (because the interface accessibility was limited), however some discrepancies between implementation and specification of the laser simulator have been found.

Future work

A direct connection between χ and TORX is definitely required when χ specification models are to be used for model-based testing. Therefore, the first steps towards such a connection are being taken. To utilize TORX within ASML in a more easy way, the connection of TORX to the test environment (which now is done through the manually developed adapter component) will also be made more generic.

Besides that, more research is performed on model-based testing, especially regarding theory and tooling extensions towards the time, data and hybrid domain. First experiments show that a timed version of TORX is able to derive tests from a timed automata specification to test the functionality and some timing requirements (e.g. response time) of a system under test.

References

[1] XOOTIC MAGAZINE *'Testing' issue*, Volume 8, Number 2, November 2000.

[2] D.A. van Beek, K.L. Man, M.A. Reniers, J.E. Rooda, and R.R.H. Schiffelers, *Syntax and Consistent Semantics of Hybrid Chi*, Computer Science Reports 04-37, Technische Universiteit Eindhoven, November 2004.

[3] J. Tretmans and E. Brinksma, *TorX: Automated model based testing*, In 1st European Conference on Model-Driven Software Engineering, December 2003.

[4] Gerard J. Holzmann, *The model checker SPIN*, Software Engineering, 23(5):279–295, 1997.

[5] E. Bortnik, N. Trčka, A.J. Wijs, B. 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, November 2005.

Contact Information

Niels Braspenning

Technische Universiteit Eindhoven
Department of Mechanical Engineering
P.O. Box 513, 5600 MB Eindhoven
The Netherlands
n.c.w.m.braspenning@tue.nl

Verum specialises in the mathematical design and verification of software systems for the original equipment manufacturer (OEM), automotive, medical and telecoms markets.

Making Software Work

Telephone: +31 (40) 235 9090
Fax: +31 (40) 235 9099
E-mail: info@verum.com
Web: www.verum.com

VERUM

A model-based approach to fault diagnosis of embedded systems[1,2]

Jurryt Pietersma, Arjan J.C. van Gemund and Andre Bos

The problems that arise from the integration of subsystems into complex, multi-disciplinary embedded systems, are a potential obstruction for the expected, exponential growth in embedded systems applications. Faults that occur because of the dynamic behavior of the integrated system are difficult to trace back to individual subsystems or components. The Model-Based Diagnosis (MBD) methodology offers a solution for the fault diagnosis of the integrated system by inferring the health of a system from a compositional system model and real-world measurements. In this article we present the initial results of our MBD research as applied on the lithography systems of ASML. We explain our methodology based on a modelling language LYDIA which is specifically being developed for the purpose of MBD. Furthermore we discuss the results of our first diagnosis test case.

Introduction

As the exponential increase in hardware performance-per-cost ratio is expected to continue, the number of embedded systems is to increase accordingly. The associated complexity crisis is a potential show stopper for the continued pervasion of embedded systems in our society. This is particularly true for complex, multi-disciplinary systems that are integrated from multiple subsystems. While these subsystems might function well separately, integrating them can cause unexpected faults. Because of the dynamic interaction between these subsystems, these faults take a lot of time and effort to diagnose, let alone fix.

One of the solutions is to automate the fault diagnosis of these integrated embedded systems. The classical way of automated diagnosis e.g., by means of application-specific code or, more generically, by using expert systems, has disadvantages. The mapping from symptoms to diagnosis is explicitly coded in the software, which means that even a minor design change of the system may require a major redesign of the diagnosis software. It also means that while trying to decrease system complexity, we actually increase it by adding a lot of diagnosis software.

A promising way of overcoming these problems is to apply a *model-based* approach to diagnosis. In the Model-Based Diagnosis (MBD) approach [5], knowledge about the system is expressed in terms of a compositional model. A generic fault diagnosis engine, using AI search algorithms, consults this model during run-time, while tracking the system. Because information about the system design is separated from the fault finding method, a design change only requires a similar change in the model. This curbs the increase in complexity.

[1]This work has been carried out as part of the TANGRAM project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under grant TSIT2026.

[2]This article was originally presented at the ASCI Conference 2004.

Within the TANGRAM project [5], a multi-university research project aimed at model-based testing and diagnosis of multi-disciplinary embedded systems, the MBD approach is applied to lithography systems as produced by ASML. While the ever increasing performance of these chip manufacturing systems actually provides us with the aforementioned exponential increase of the hardware performance-per-cost ratio, these systems themselves are by no means free from the complexity crisis. Hence, MBD is seen as an important solution to decrease the cost of design, integration and operation of these systems.

Despite recent advances in MBD [6, 10, 11, 13] complex, multi-disciplinary systems as found in ASML are currently beyond the state-of-the-art. Furthermore, given an adequate MBD technique, a subsequent problem is model specification, which is a labor-intensive and error-prone process. Within the TANGRAM project MBD research focuses on extension of MBD technology with respect to time, state and probability.

Our MBD approach is based on the modelling language LYDIA (Language for sYstems DIAgnosis) [7]. LYDIA is model-based systems specification language aimed at systems fault diagnosis and simulation using the same model. In this article we present the initial results of our MBD research as pursued in the TANGRAM project. We demonstrate how LYDIA can be used for diagnosis in general. In addition, we describe how this methodology has been applied in terms of a case study within the TANGRAM context.

The article is organized as follows. In the first section we introduce the principles of MBD with an example. In the second section we present the LYDIA modelling language and accompanying tools, including two examples on how to use these tools for diagnosis. In the third section we present the case study and discuss the resulting model and its diagnosis. In the final section we draw our conclusions from this initial research.

Model-Based Diagnosis

Diagnosis is the process of finding differences between models and reality. Model-Based Diagnosis (MBD), first suggested by Reiter [12] and continued by de Kleer, Mackworth and Reiter [4], is the process of finding faults in a system on the basis of observations from reality and reasoning about a model of the system. Formally, model-based diagnosis can be seen as finding faulty components that explain the difference between behavior predicted by a model and behavior observed in reality.

For example, consider an example of MBD using a digital circuit, consisting of three inverters: A, B, and C (Figure 1). Let $w = 1$. Then y and z should be 1 as well. If observations indicate that $y = 0$ and $z = 1$ then the diagnosis could be that component B is faulty. Another option is that A and C are faulty, as this also explains the symptoms. The trivial solution, A, B, and C all faulty, also explains the observations but is of no added value, as any superset of $\{B\}$ or $\{AC\}$ explains the observations. A subset of $\{B\}$ or $\{AC\}$ does not. That is why $\{B\}$,$\{AC\}$ can be called the *minimal* fault set. This diagnosis can be formalized, using a logical model, as follows.

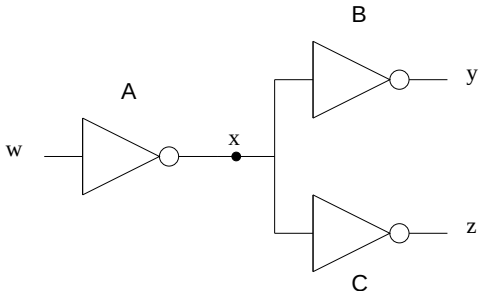


Figure 1: Three-inverters example

Let h indicate the *health* of a component. If $h = 1$ then the component is "healthy" and obeys certain behavioral rules. The three inverter example has three components (A,B,C), so it has three such rules:

$$h_A \rightarrow x = \overline{w}$$
$$h_B \rightarrow y = \overline{x}$$
$$h_C \rightarrow z = \overline{x}$$

As the observations are: $w = 1, y = 0, z = 1$, it follows (applying the rule $p \rightarrow q \Leftrightarrow \overline{p} + q$):

$$(\overline{h}_A + \overline{x}) \cdot (\overline{h}_B + x) \cdot (\overline{h}_C + \overline{x}) = 1 \qquad (3)$$

This can be rewritten to DNF-form:

$$\overline{h}_A\overline{h}_B\overline{h}_C + \overline{h}_A\overline{h}_B\overline{x} + \overline{h}_A\overline{h}_C x + \overline{h}_B\overline{h}_C\overline{x} + \overline{h}_B\overline{x} = 1$$

This formula reduces to the following prime implicants:

$$\overline{h}_A\overline{h}_C x + \overline{h}_B\overline{x} = 1$$

Thus $\overline{h}_A\overline{h}_C = 1$ (A and C are faulty when $x = 1$) or $\overline{h}_B = 1$ (B is faulty when $x = 0$).

Another possibility to calculate the faulty components is by using conflict sets. Applying the resolution rule $(p+q)\cdot(r+\overline{q}) \rightarrow (p+r)$ and De Morgan's Laws, from equation (3) it follows:

$$\begin{aligned} (\overline{h}_A + \overline{h}_B) \cdot (\overline{h}_B + \overline{h}_C) &= 1 \\ \overline{(\overline{h}_A + \overline{h}_B) \cdot (\overline{h}_B + \overline{h}_C)} &= 0 \\ \overline{(\overline{h}_A + \overline{h}_B)} + \overline{(\overline{h}_B + \overline{h}_C)} &= 0 \\ h_A h_B + h_B h_C &= 0 \end{aligned}$$

so $h_A h_B = 0$, and $h_B h_C = 0$ which means $\{$AB$\}$ and $\{$BC$\}$ are conflict sets. Finding the minimal fault set, or minimal conflicts, can be done using algorithms for the Hitting Set problem. This, of course, also results in the sets $\{$AC$\}$ and $\{$B$\}$. In summary, in MBD of combinational systems the model is solved for h using propositional logic. In the next section we describe our tool for MBD.

Model-Based Diagnosis with LYDIA

LYDIA

In the following, we briefly present some of the major features of LYDIA. Due to space constraints we only present those constructs that are used in the sequel. For a comprehensive introduction to LYDIA we refer to [7]. Each LYDIA statement is a Boolean equation (proposition), and all statements apply concurrently. Each variable, e.g., `x` is a function of (continuous) time, i.e., `x(t)`. The time argument is omitted. All operators are functions that operate on each time argument (i.e., element-wise data flow). Thus,

```
op(x)  <=> for all t: op(x(t)) = true
x op y <=> for all t: x(t) op y(t) = true
```

Roughly speaking, LYDIA can be placed in the "functional" category of the functional (equational) vs. imperative (state-transition) dichotomy. It resembles synchronous languages [2], such as Lustre [9] and Signal [8], with the major difference being the absence of synchronous time. Timed actions are asynchronous, i.e., signals (and events) are not sampled at regular time intervals. State transitions may also be timeless (cf. timed and immediate transitions in timed Petri nets [1]), with only the transitions that are enabled at the same time being synchronous. In this respect, LYDIA resembles a synchronous language with infinite clock resolution, which is implemented through a discrete-event propagation scheme. Although based on a functional approach, many of the LYDIA models are expressed in a state-transition style as syntactic sugar. The reason for this is that the description of some systems (e.g., state-machines) in a functional language sometimes proves awkward, where a more state-transition-oriented dialect offers a much more natural model.

Combinational Operators

Apart from the usual operators, such as `=`, `+`, `-`, `/`, `*`, `and`, `or`, `not`, `>`, `<`, `>=`, `<=`, `sin`, `cos`, `tan`, `sqrt`, `pow`, `log`, `exp`, `max`, `min`, `abs`, etc., the derived operators include `!=`, `if`, `if-else`, defined as:

```
a != b <=> ! (a = b)

if (c) x <=> (! c) + x

if (c) x else y <=> (c * x) + ((! c) * y)
```

where `!`, `+`, `*`, are equivalent to `not`, `or`, `and`, respectively.

Time Operator

Time delay is described by the `after` function:

```
y = (x after delta default x0)
```

that defines a signal (variable) `y` that lags behind the signal `x` according to

$$y(t) = \begin{cases} x(t-\delta), & t \geq \delta; \\ x0, & 0 \leq t < \delta. \end{cases}$$

The default clause is optional.

Apart from the above constructs, LYDIA also features state transition operators, the treatment of which, however, is beyond the scope of this article.

LYDIA tools

Currently we have developed a number of tools that operate on LYDIA models. There is a LYDIA compiler called `lydia` that translates LYDIA models into C source code for the purpose of simulation, or into symptom-diagnosis lookup tables for the purpose of diagnosis. The latter tables are generated using propositional SAT solving and are consulted by a diagnostic engine, called `scotty`, that monitors the system's input and output, and returns a list of possible diagnoses, in order of probability. Currently the C compilation mode only works for models that operate in the discrete time domain. A second simulator `lsim` has been developed which interprets and simulates continuous-time LYDIA models.

Examples

This section describes some basic LYDIA examples. The first LYDIA system models an electronic inverter with a 10ns propagation delay, after which `y` becomes the inverted of `x`. The second example produces a clock signal `c` with a period of 1.0s. The last example simulates a bouncing ball with height `h` and velocity `v`. The velocity is reversed when the velocity and height are less than zero. The velocity and height are calculated using explicit first order Euler integration as specified by the function `integrate`.

Example 1:

```
system inverter (x: bool, y: bool)
{
  t_p = 1e-08
  y = ( not x after t_p )
}
```

Example 2:

```
system clock (c: bool)
{
  period = 1.0
  c = ( ( not c ) after period / 2 )
}
```

Example 3:

```
system ball (h: float,
             v: float,
             g: float,
             d_t: float,
             c: float)
{
  h = (integrate(h,v,dt)
      after dt default 5.0)
  v = ((if (b) (-c * v)
      else
      integrate(v,-g,dt))
      after dt default 0.0)
  b = ((v < 0.0) and (h < 0.0))
  exit = (time > 10.0)
}

function integrate (y: float,
                    f: float,
                    dt: float) : float =
{
  integrate(y,f,dt) = (y + f * dt)
}
```

Diagnosis of inverter model

Consider the inverter of the previous section, which this time either inverts a Boolean signal if healthy, or is stuck-at-zero, if at fault. The LYDIA model is given by:

```
system inverter (x: bool,
                 h: bool,
                 y: bool)
{
  t_p = 1e-08
  y = if ( h )
      ( !x after t_p default false )
      else false
}
```

where x, y denote input, output respectively, and h denotes the so-called health variable. We can run this model with lsim and a data input file, which results in the following output:

```
time:      x: h: y:
0.00000000 1  1  0
1.00000000 0  1  0
1.00000001 0  1  1
2.00000000 1  0  0
3.00000000 0  0  0
```

The first column indicates the simulation time, the second and third column are the input variables which are repeated from the input file. The result of the simulation is shown in the last column and corresponds to the expected output, y is only true, 10ns after the moment the inverter is healthy and the input is false.

It is instructive to note that the functional character of LYDIA allows us to use lsim simulator as a limited diagnostic engine. Instead of providing lsim with x and h we provide it with the observations x and y from which it deduces h, as shown below. The U symbol indicates an unknown value.

```
time:      x: y: h:
0.00000000 1  0  U
1.00000000 0  0  U
1.00000001 0  1  1
2.00000000 1  0  0
2.00000001 1  0  U
3.00000000 0  0  U
3.00000001 0  0  0
```

We observe that a diagnosis for this system is only possible in two out of four cases, namely only when the output of a healthy system, with a delay of 10ns, does not coincide with the output of an unhealthy system. Thus, for this system, only when the output is true can we distinguish between an h that is true or false.

Diagnosis of three-inverters model

Of course, the real goals for using LYDIA for MBD is diagnosis of far more complex, real-life systems than the one mentioned in the previous section. To perform diagnosis of these systems we can compose models out of simpler components. To illustrate this, we expand our initial model of one inverter to a model of the three-inverters example mentioned in the first section:

```
#include inverter.sys
system inverter3 (w: bool,
                  hA: bool,
                  hB: bool,
                  hC: bool,
                  y: bool,
                  z: bool)
{
  probability ( hA = false ) = 0.01
  probability ( hB = false ) = 0.01
  probability ( hC = false ) = 0.01

  inverter ( w, hA, x)
  inverter ( x, hB, y)
  inverter ( x, hC, z)
}
```

A diagnostic approach based on mere simulation can no longer be used to diagnose this system because, as explained earlier, one combination of input and outputs can be caused by different types of failures. The simulator can only solve single equations for only one solution variable. To solve this general combinational problem we use the specialized diagnostic engine scotty, mentioned in Section 2, which can handle these combinatorics. At this point, our diagnosis algorithm does not allow time delay. Consequently in the following we consider the inverter model without the after statement. To make the model more generic and compliant with our logical three-inverters model, we also leave out the specific stuck-at-zero fault mode. To allow LYDIA to work with failure probabilities, we introduce the keyword probability, to indicate a health variable that has a certain probability of being false or true. As an example, we run the diagnostic engine with the input/output combination mentioned in the first section: w=1, y=0 and z=1. The result of the diagnostic engine is given by:

```
(0.97049200)  hA=true  hB=false hC=true
(0.00980295)  hA=false hB=false hC=true
(0.00980295)  hA=false hB=true  hC=false
(0.00980295)  hA=true  hB=false hC=false
(9.90197e-05) hA=false hB=false hC=false
```

The results correspond to the fault cases that can be derived from the minimal fault set {B},{AC} as calculated in the first section. The cases with two faulty inverters all have the same probability because all three inverters have the same individual failure probability. From the results it is also clear

that the trivial case of three failing inverters is extremely unlikely.

A current disadvantage of using `scotty` instead of `lsim` is the lack of support for time and state. As mentioned in the introduction, extending the diagnostic engine to incorporate this, is one of the goals of our ongoing research.

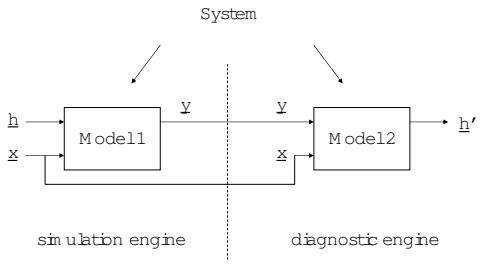


Figure 2: Connection between the simulation (M1) and diagnosis (M2) model of target system.

Modelling case study

Methodology

While the ultimate goal of our research is to diagnose lithography systems in the real world, our current goal is to gain experience in the specification of real-world models and our diagnosis algorithms. For this we need as few uncertainties as possible, which is why currently we only apply our diagnosis on the simulation models and not on the real system. Consequently, we proceed according to the following approach. We derive a simulation model M1 of the system under study. Its purpose is to:

1. document our understanding of the ASML systems including the possible failure modes of each component;
2. serve as a starting point for the derivation of a diagnosis model M2.

Our current experimental setup is shown in Figure 2. In this figure our simulation model M1 is on the left. We can insert failures (h) in this model, which we can then diagnose (h') using our diagnostic model M2. Ideally, h' should equal h for all (fault) scenarios.

In the current early stage of our research these models are generally not equal, because, as mentioned in the third section, while we have no problem *simulating* models with time and state, we are only able to *diagnose* combinational models. As we make progress, our diagnostic model M2 will evolve in the direction of M1. In the following we describe M1 and the subsequent derivation of M2.

Simulation model

At present, a laser sub-system is chosen as a case study for the TANGRAM project. The purpose of this system is to provide the lithography scanner with an exact dose of light energy to expose the wafer. The dose is provided in the form of laser pulses. Besides the laser, the model for this system also includes the interface with the scanner and the laser control software located at the scanner side.

To build this model of the laser system both a top-down and bottom-up approach is followed. In the top-down approach we model the entire structure of the whole system. We start out by interfacing with empty LYDIA systems and gradually add functionality and fault modes. In the bottom-up approach we choose a specific sub-system, of which the basic functionality is implemented in a LYDIA model. Furthermore, we also investigate known or interesting failure modes of this sub-system and introduce health variables to simulate this behavior. An example of this approach is the shutter module. The shutter can be thought of as part of the optical interface that blocks or passes on the light emitted by the laser. Beside this nominal functionality we also implemented the following faulty behavior. A nominal shutter would start opening when the "open" command is given, and would only report that it is fully opened when done. A fault mode of this shutter, which has been known to exist in an earlier design, is that it would not wait to be fully opened,

but would immediately return the "open" status after the command has been given. The following LYDIA code implements both the nominal and fault behavior.

```
% common.sys contains the clip and
% latch functions
#include common.sys

system shutter_M1 (
  % commands
  cmd_open: bool, cmd_close: bool,

  % health parameters
  h_open: bool, h_close: bool,

  %light coming in and going out
  light_in: float, light_out: float,

  % status
  sts_open: bool, sts_close: bool)
{
  % latch the mode based on the command
  latch (cmd_close, cmd_open, mode_open)
  latch (cmd_open, cmd_close, mode_close)

  sts_open = (h_open and (pos = 0.0))
           or (!h_open and mode_open)

  sts_close = (h_close and (pos = SHUT))
           or (!h_close and mode_close)

  step = if (mode_close) (CONST_STEP)
         else
           (if (mode_open) (-CONST_STEP)
           else (0.0))

  % integrate and clip position
  % between 0.0 and SHUT
  pos = clip ( 0.0, integrate (SHUT,
        pos, step, TIME_STEP ), SHUT )

  % calculate beam attenuation
  light_out = ((SHUT - pos) * light_in)
}
```

In this model the shutter latches the open or close command (pulse) to an internal mode (level). Depending on this mode the shutter position is either decreased (opened) or increased (closed). The LYDIA systems `latch`, `clip` and `integrate` are defined in the included LYDIA file `common.sys`. The `sts_open` and `sts_close` status signals are based on the shutter position if the sensors are healthy, and otherwise simply by the internal mode. The latter corresponds to the non-nominal behavior of the shutter.

Diagnostic model

As explained earlier, due to the limitation of our diagnostic algorithm, the diagnostic model for the current experiments is a simplified version of our simulation model. Again, we will use the shutter model as an example. The shutter model makes use of time, as it takes time to open or close, and uses state, as it has internal modes, `pos`, `mode_open` and `mode_close`, which determine the shutter position and whether it is opening or closing. The associated time and state variables prohibit our combinational diagnosis approach and therefore we have to convert M1 to a model M2 specifically suited for diagnosis.

In our conversion from M1 to M2 we take the following approach:

1. isolate the equations with health parameters, on the condition that they are combinational. For each health parameter we also introduce its probability of being false or true;
2. re-use those (auxiliary) equations from M1 that are required to solve the isolated, health equations.

Thus our diagnostic approach includes simulation next to diagnosis. The result of applying these two steps on our shutter model is as follows:

```
system shutter_M2
{
  % combinational health equations
  probability (h_open = false) =0.01
  probability (h_close = false) =0.01

  sts_open = (h_open and (pos = 0.0))
         or (!h_open and mode_open)

  sts_close = (h_close and (pos = SHUT)
         or (!h_close and mode_close)

  % auxiliary equations
  latch (cmd_close, cmd_open, mode_open)
  latch (cmd_open, cmd_close, mode_close)

  step = if (mode_close) (CONST_STEP)
         else
           (if (mode_open) (-CONST_STEP)
           else (0.0))

  pos = clip ( 0.0, integrate (SHUT,
        pos, step, TIME_STEP ), SHUT )
}
```

We use the M2 model to diagnose our M1 model with the setup shown in Figure 2. In this setup `lsim` simulates M1 as well as the auxiliary equations of M2. The combinational health equations of M2 are compiled into a symptom-diagnosis lookup table and used by `scotty` for the actual diagnosis, as explained in the second section.

Diagnostic test results

In the next experiment we use the following values for the constants: `SHUT`=1.0, `SHUTTER_STEP`=0.1 and `TIME_STEP`=0.01. As our models have a symmetric description for the open and close sensor, the simulation and diagnosis results for both sensors are also symmetric. Therefore we limit our discussion to the open sensor. In the first 6.51s we simulate a healthy open sensor. The first test starts at 1.00s and we allow the shutter to fully open, after which we close it again at 2.0s. The second run starts at 3.00s but now we interrupt the shutter at 3.01s, before it can open completely. At 5.0 we do the same but after the interrupt we open it again. In the second half (t $\geq$ 7.00s) we perform the same tests, only now with an unhealthy sensor. The experiment yields the following results:

```
time:
|    h_open_M1:
|    | cmd_open:
|    | | mode_open:
|    | | | (pos=0.0):
|    | | | | sts_open:
|    | | | | | h_open_M2:
|    | | | | | | probability:
|    | | | | | | |
1    2 3 4 5 6 7 8

0.00 1 0 0 0 0 1 0.9801
1.00 1 1 1 0 0 1 0.9900
1.11 1 1 1 1 1 1 0.9801
2.00 1 0 0 1 1 1 0.9900
2.01 1 0 0 0 0 1 0.9801
3.00 1 1 1 0 0 1 0.9900
3.01 1 0 0 0 0 1 0.9801
5.00 1 1 1 0 0 1 0.9900
5.01 1 0 0 0 0 1 0.9801
5.02 1 1 1 0 0 1 0.9900
5.13 1 1 1 1 1 1 0.9801
6.00 1 0 1 1 1 1 0.9801
6.50 1 0 0 1 1 1 0.9900
6.51 1 0 0 0 0 1 0.9801

8.00 0 1 1 0 1 0 0.9900
 8.11 0 1 1 1 1 1 0.9801
 9.00 0 0 0 1 0 0 0.9900
 9.01 0 0 0 0 0 1 0.9801
10.00 0 1 1 0 1 0 0.9900
10.01 0 0 0 0 0 1 0.9801
12.00 0 1 1 0 1 0 0.9900
12.01 0 0 0 0 0 1 0.9801
12.02 0 1 1 0 1 0 0.9900
12.13 0 1 1 1 1 1 0.9801
13.00 0 0 1 1 1 1 0.9801
```

The second column `h_open_M1` gives the inserted sensor health of our simulation model. The seventh column gives the diagnosed health `h_open_M2` as inferred from M2 and the last column the probability of this diagnosis. From the first part of the results we can see that `scotty` correctly predicts that the sensor is healthy. The second part shows that a correct diagnosis is only performed when the `(pos=0.0)` expression in the fifth column is unequal to the `sts_open` variable in the sixth column. In other words, when the output of the healthy shutter, for which `sts_open` is only true if `pos`=0.0, does not coincide with that of the unhealthy sensor, for which `sts_open` is only true if `mode_open` is true. This corresponds with the results from the diagnosis of the single inverter example in the first section.

Conclusions

In this article we have presented our MBD approach and research objectives as pursued in the TANGRAM project. We have also demonstrated how to use the modelling language LYDIA in this approach. The examples show that we can already model and simulate the basic functionality of a realistic subsystem. Furthermore we have shown how we can make these models suited for combinational diagnosis. In the coming period we will put more emphasis on the diagnosis of existing fault scenarios. From this we expect to learn more about how to deal with the occurrence of time and state behavior in our diagnosis models.

Acknowledgements

We gratefully acknowledge the feedback from the discussions with our TANGRAM project partners from ASML, Eindhoven University of Technol-

ogy, Embedded Systems Institute, NLR, TNO-TPD, Twente University and the University of Nijmegen.

References

[1] M. Ajmone Marsan, G. Balbo and G. Conte, "A class of Generalized Stochastic Petri Nets for the performance analysis of multiprocessor systems," *ACM Tr. on Comp. Syst.*, vol. 2, May 1984, pp. 93–122.

[2] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic and R. De Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, January 2003, pp. 64–82.

[3] Tom Brugman and Frans Beenker, "Project plan for the TANGRAM project on model-based testing," Tech. Rep. Doc. Nr. 2002-10060 version 09, Embedded Systems Institute, Nov. 2002.

[4] Johan de Kleer, A. K. Mackworth and R. Reiter, "Characterizing diagnoses and systems," *Artificial Intelligence*, vol. 56, 1992, pp. 197–222.

[5] Johan de Kleer and Brian C. Williams, "Diagnosing multiple faults," in *Readings in Nonmonotonic Reasoning* (Matthew L. Ginsberg, ed.), Los Altos, California: Morgan Kaufmann, 1987, pp. 372–388.

[6] A. Fijany, F. Vatan, A. Barrett and R. Mackey, "New approaches for solving the diagnosis problem," 2002.

[7] A.J.C. van Gemund, "LYDIA Version 1.1 Tutorial," Tech. Rep. PDS-2003-001, Delft University of Technology, Nov. 2003.

[8] P.L. Guernic, M.L. Borgne, T. Gautier and C.L. Maire, "Programming real time applications with Signal," *Proceedings of the IEEE*, vol. 79, Sept. 1991, pp. 1321–1336.

[9] N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud, "The synchronous data-flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, September 1991, pp. 1305–1320.

[10] James Kurien, "Model-based monitoring, diagnosis and control." Ph. D. Thesis Proposal, 2000.

[11] Sriram Narasimhan and Gautam Biswas, "An approach to model-based diagnosis of hybrid systems," in *Hybrid Systems: Computation and Control HSCC* (C. J. Tomlin and M. R. Greenstreet, eds.), vol. 2289 of *LNCS*, Springer, Mar. 2002, pp. 465–478.

[12] R. Reiter, "A theory of diagnosis from first principles," in *Readings in Nonmonotonic Reasoning* (Matthew. L. Ginsberg, ed.), Los Altos, California: Kaufmann, 1987, pp. 352–371.

[13] Brian C. Williams and Robert J. Ragno, "Conflict-directed A* and its role in model-based embedded systems." To appear in Journal of Discrete Applied Math.

Contact Information

Jurryt Pietersma

Parallel and Distributed Systems Group
Faculty of Electrical Engineering
Mathematics and Computer Science
Delft University of Technology
P.O. Box 5031, NL-2600 GA Delft
The Netherlands
j.pietersma@ewi.tudelft.nl

Arjan J.C. van Gemund

Parallel and Distributed Systems Group
Faculty of Electrical Engineering
Mathematics and Computer Science
Delft University of Technology
P.O. Box 5031, NL-2600 GA Delft
The Netherlands
a.j.c.vangemund@ewi.tudelft.nl

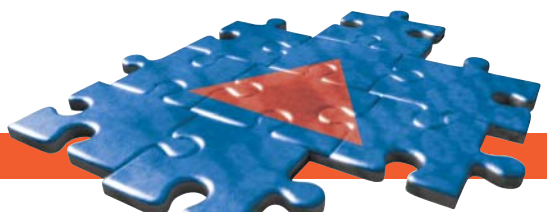
Andre Bos

Science & Technology BV
P.O. Box 608, NL-2600 AP Delft
The Netherlands
bos@science-and-technology.nl

Uitdaging als secundaire arbeidsvoorwaarde

Ruim 100 gedreven technische software specialisten, die behoren tot de besten in hun vakgebied, werken bij softwarehuis TOPIC in Best aan de meest uitdagende en uiteenlopende projecten in de consumentenelektronica, medische systemen en professionele systemen. Met succes. TOPIC blijft groeien. Daarom zijn we permanent op zoek naar **(embedded) software specialisten** met minimaal 2 jaar werkervaring die hun kick halen uit uitdagende opdrachten. Vakspecialisten die zich thuis voelen bij TOPIC. En die zich willen blijven ontwikkelen. In projecten en via ons Personal Improvement Program. Zin in een kick? En heb je de persoonlijkheid, ervaring en kwaliteit die past bij TOPIC? Mail dan snel je motivatie met CV naar recruitment@topic.nl of bel eerst met Frank de Roo, Manager Recruitment, op nummer (0499) 336979.

TOPIC
SOFTWARE GROUP
EMBEDDED IN YOUR FUTURE



Voor meer informatie: WWW.TOPIC.NL

A multidisciplinary model-based test and integration infrastructure[1]

Will Denissen

Current market trends like shorter time to market, faster return on investment, flexible product families, first time right etc., will put strong requirements on the development process of manufacturing companies. In this article we will present a test and integration infrastructure that supports the development process in these changing markets.

Introduction

ASML[4] is the carrying industrial partner within the Tangram[5] project and needs support for their test and integration challenges. Because no single solution to this problem exists a broad approach is taken in the form of four different lines of attentions, each defined to tackle a different part of the test and integration problem. These lines of attention are: test strategy, model based testing, model based diagnostics, and test and integration infrastructure.

In this article we will concentrate on the last line of attention and present a multidisciplinary model-based test and integration infrastructure. It is developed and used within the Tangram project and must support the other lines of attention.

The article is organized as follows. In the first section terminology is introduced that will help the communication between the different disciplines for which the test and integration infrastructure is developed. In the second section, different kinds of testing are presented which serve as use cases for the test and integration infrastructure. The third section describes the early integration concept for multiple disciplines. Then the design of the test and integration infrastructure is given.

Terminology

An extra challenge in multi-disciplinary testing w.r.t. mono-disciplinary testing is that each discipline uses its own terminology and some terms overlap and therefore might be misinterpreted. The disciplines we distinguish are: **system, software, electrical, mechanical, and optical engineering**.

To identify when and where **testing activities** can take place we have to concentrate on the **development process** (the classical v-model) as used within ASML. Figure 1 shows the different **development levels** and different **development phases** that can be identified in the ASML product development process.

For a new product a typical sequence of activities will follow the curved arrow representing the time axis. Going from a single system design **decomposing** it into several sub-systems up until an array of unit level designs. For each unit level design a realisation is constructed. Unit realisations are **composed** into subsystems and finally into a single complete system realisation.

The two sided arrows depict for each development level and development phase that a **testing activity** can occur. From our perspective a testing activity is no more than checking the consistency between two entities. These entities are either **designs** (in the

[1]This work has been carried out as part of the TANGRAM project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under grant TSIT2026.

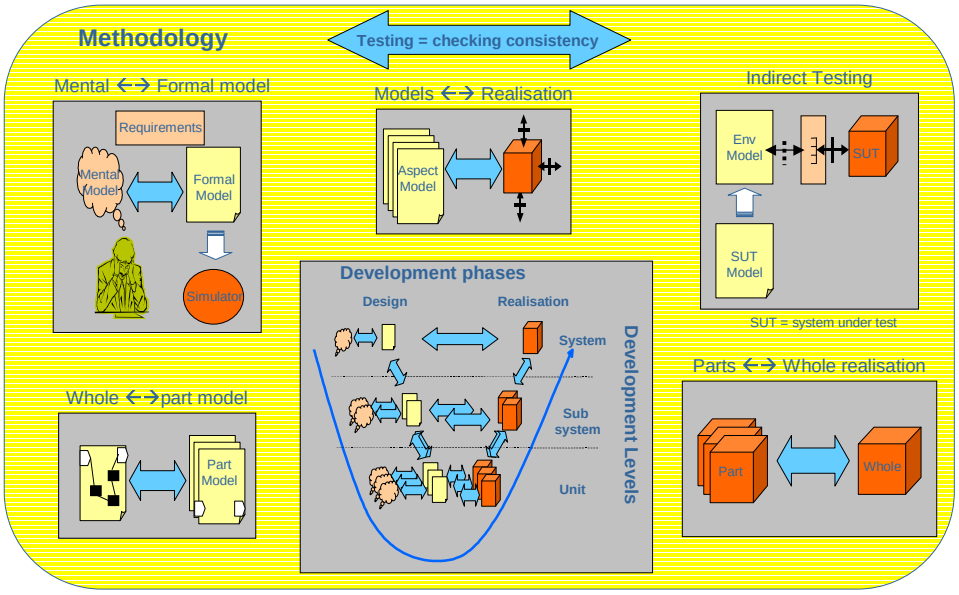


Figure 1: Different kinds of testing at different development levels and phases

form of documents, models, executable models) or **realisations** (in the form of libraries, executables, hardware or a combination of both). We will deliberately not talk about validation and verification because both approaches assume that one of the entities is correct and the other is incorrect. In practice both entities might need a correction. That's the reason why testing is depicted as a two-sided arrow. The definition and characteristics of each testing activity together with some examples is given in a separate section.

Development levels

At each development level a different level of abstraction is used to describe the system. The different development levels range from the high-level **system level**, via one or more **subsystem levels** up to the most low-level **unit level**. Going from high-level to low-level development levels the amount of information increases, describing more details of the system. The same holds for the amount of designs, realisations, and people involved.

At each level, possibly different groups of experts, playing a different role, cooperate in making a design for that level. At each level the design contains as much detail as relevant for that level. To cope with complexity the design at a certain level (the **whole-design**) is **decomposed** into a set of designs (the **part-designs**) at the next lower level. The experts at a certain level expect that the experts who are filling in the part-designs do not violate their whole-design. Each expert will develop his or her own mental model of the design they work on as a group. The **whole-realisation**, which is **composed** from the **parts-realisation** will be tested at the same level of abstraction as the whole-design. Both the **design** and the **realisation** fulfill the same set of requirements.

System level

At system level there can, by definition, be only one design and one realisation. There is no level above the system level. The group of people involved is typically small and their **role** is that of a **system engineer**. Based on their skills, experience, and common practice they will create or select a proper design. The design will typically deal with identifying and naming the subsystems and identifying and naming their interactions as **interfaces** and allocating budgets over these subsystems, without filling in the details of these sub-systems.

Sub-system level

A subsystem level is, by definition not the system level and not the unit level. There can be zero or

more subsystem levels. At subsystem level each part-design of its next higher-level whole-design is filled in. The group of people involved is typically of medium size and originate from different disciplines.

Unit level

A unit level is, by definition, the most detailed level of design and realizations. There are no levels below a unit level. Over all units, a lot of people are involved from different disciplines. For a given unit the experts originate from a single discipline. The designs are typically so detailed and complete that subcontractors can make (e.g. Electronics: PCB manufacturers) or tools can generate (e.g. Software: compilers, Mechanics: CNC-machines) realisations out of it .

Development phases

Two development phases can be distinguished, a **design phase** and a **realization phase**.

Design phase

In the design phase all kinds of information about the system's structure, behavior or operating constraints are collected and archived. Some information will end up in documents and others in models.

A design can contain several **models**. There are two types of models: **structural models** (e.g. a class diagram in UML[10], or a mechanical model in Unigraphics) and **behavior models** (e.g. an activity diagram in UML, or a 3D kinematic model). A behavior model is called an **executable model** when a **simulator** can execute it. The simulator simulates the executable model according to a certain **paradigm** (e.g. discrete event DE, communicating sequential processes CSP, Continuous time CT, Hybrid (DE + CT)). A simulator has a notion of logical time that can either run faster or slower than wall clock time. Every simulator is based on the same implementation pattern. A modeler can specify a model as relations between modeling entities forming a set of equations. The simulator will solve this set of equations for the current logical time, calculates the logical timestep, and advances the logical time with this timestep. This sequence is repeated until the end of logical time is reached.

Each model will only model a specific **aspect** (e.g. temperature distribution, resource scheduling) of the system. In the design phase interactions between models will be identified. An **interface** describes and names such an interaction.

Realisation phase

In the realisation phase the different realisations come to completion, part realisation will be assembled, tested and integrated into whole realisations.

A **realisation** is something that consumes resources (materials, space, time, memory footprint etc.). Realisations have commercial value; they are costly to build and/or to maintain. Realisations have identity. Two realisations can be identified by their product numbers, but both can be build from the same design.

In a realisation all kinds of different aspects are intrinsically combined and will influence each other in the form of **interactions**. Some interactions are known at design time and can have a model counterpart in the form of model interfaces and might be realised as **real interfaces** (e.g. electronic connectors, software function calls, optical light paths) but others might yet still be undetected (**hidden interactions**) (e.g. physical aspects due to the small nanometer scale of operation).

The discipline interfaces within a realisation are typically layered as shown on the right side in Figure 3. The interactions between disciplines occur only at the given interfaces. A interaction, for instance, between an optical lens and a software statement is hard to imagine without an electronic interface in between.

Kinds of Interfaces

In both the design phase and the realisation phase, interfaces between sub-systems or units exist but their nature of interaction are quite different. Therefore, two kinds of interfaces can be distinguished; **model interfaces** and **real interfaces**. The characteristics of each of them will be described below.

Model interfaces

Model interfaces model the flow of abstract information between models. The information flow

between models are a kind of data streams. At each logical clock increment, which are discrete moments in time a simulator will send/receive a datum to/from one or several other simulators. The logical clock of each involved simulator needs to be synchronised with other logical clocks. This can be done directly by a separate logical time manager or indirectly by configuring all participating simulators such that all logical clocks start at the same logical time with the same logical increments. Model interfaces are visualised in Figure 3 as a line crossed by a dotted line.

Real interfaces

Real interfaces incorporate both data and control flow of an interaction at unit level. Real interfaces are visualised in Figure 3 as a line crossed by a bold line, and can be annotated with its type. Currently we distinguish only real software, electronical, and physical interfaces. Both the real software and real electronic interface has a notion of direction. The flow of information takes time to travel from the producer to the consumer.

For **real software interfaces** the information flowing through the interface is the exact function call with all its parameters properly filled in, in the expected order, at **discrete moments in time** without knowing when the actions will actually take place.

For **real electronical interfaces** the information flows through electrical wires. The interface describes the signals, their shapes, duration, and connector with the proper mechanical dimensions. Digital interactions can take place only at discrete events (at the clock ticks). Analog interactions take place in continuous time.

For **real physical interfaces** there is no notion of information flow or causality, the interface just identifies an interaction between two or more entities and take place in **continuous time**. Some interaction might exist in real life but not been detected/known by the developers. An example of a real physical interface is a collision between two mechanical entities which occurs instantly and continuously.

Different kinds of testing

Now that we have introduced our terminology we can concentrate on describing the different kinds of testing that can take place in the development process (i.e the two-sided arrows in Figure 1). The testing process needs to be described because it provides the use cases and requirements for the test and integration infrastructure that we have designed and build. In the following subsections we will describe each kind of testing as depicted in Figure 1 around the development process.

Mental ↔ Formal model testing

At each development level a **designer** is involved that needs to come up with a **design** that fulfills the **requirements**. Given the requirements a lot of designs can be found that all fulfill the same requirements. This set of designs, is called the **design space** for the given requirements. While making a design new parts will identified and their relations. Some parts might be designed as common/commercial off the shelf (COTS) parts. Others might be a commonly used interface in the form of a design pattern. But whatever the design will be it will impose new/more detailed requirements on its parts (i.e the next lower development level). It is the designer's role to find such a design that fulfills the requirements at his level and minimises the lower level requirements and maximises their **designability**.

Because a designer has a freedom of selecting a design from a design space, he/she needs to get some feeling of how his design will look like (structure) or behaves. Preferably a designer will use a computer added design (CAD) tool to support his design activities. With such a CAD tool the designer builds up a **mental model** on the structure and behavior of his design. In order to use his CAD tool he needs to express his design in a **formal model**. The formal model that expresses the design is communicatable among other developers because of its unambiguous semantics. His mental model however is not transferable because, it will never be as complete, accurate, or unambiguous as a formal model. A developer can also never cope with the different versions of designs that might pop up and all the implications that the combinations of these designs might have.

The mental model of the developer is kept aligned/synchronised/consistent with the formal model. The developer will learn from the formal

model and adjusts his mental model accordingly. The formal model becomes more detailed until it mimics the behavior from the mental model.

Whole ↔ Part model testing

Testing whole models with part models all have to do with decomposing a design in a set of sub designs. This decomposition of whole designs into parts designs is typically aligned with the whole realistion and its parts realisation. There is typically a one to one relation between whole and parts models and realisations at each design level.

Decomposing a whole model into parts models is nothing new within a single discipline, and is the basic pattern to handle complexity. For instance, a system engineer can decompose his budgets in a hierarchical manner. A software engineer can decompose his software program in a set of subprograms. An electrical engineer can decompose his electrical model into a set of sub models. Some sub models might be standardised into a library of models (e.g. software: mathematical library, electronics: counters, clock dividers, mechanics: robot arm, gearbox).

The testing activity in **whole ↔ part testing** consists of checking that the developers who will come up with the part designs do not violate the requirements imposed on the whole design and vice versa. Once a discrepancy is detected either the whole or the part models need to be modified such that they together are consistent again.

Part ↔ Whole realization testing

Part ↔ whole realisation testing occurs the moment the different part realisations are assembled together (a.k.a. integration phase). The kind of problems you observe, are typically related to resource conflicts or unknown interactions. For instance, assembling together different software realisations (e.g. libraries, executables) might show that the memory footprint of the whole exceeds the available memory.

Assembling mechanical parts might uncover incompatibilities. The shared resource could be the space the parts may occupy at a given moment in time. Something similar occurs in electrical engineering. The fan-in and fan-out of the active electrical parts must match when assembled into a whole otherwise the quality of the electrical signals will degrade.

Resources might be shared by different disciplines. For instance, a certain volume might be blocking an optical light path by a mechanical component. Or the mechanical materials used might outgass such that the optical lenses get polluted. Typically structural interactions (without a time dependency) are directly detected while assembling. Behavioral interactions can only be detected when the whole realisation can be executed/used/employed according to its use cases.

Normally most of the interactions are expected because the were already known by experience or from previous similar systems, these interactions are then also modeled in the design phase. Unknown interactions are typically detected in this testing activity. Judging whether the whole realisation is functioning correctly is done indirectly. First the part realisations are tested with their part models on conformity, then the whole realisation is tested on its conformity with its whole model.

Model ↔ Realization testing

Model ↔ realisation testing is normally known as **conformance testing**. The to be build realisation is described by models, each capturing a different aspect. For each model the realisation must conform in structure and behavior. Both the models and the realisation are **open**, i.e. the interaction with their **environment** is modeled. The interfaces and their kind (software, electronics, physical) are identified. The behavior of the environment is modeled as a set of use cases. A realisation conforms to its models when both the observations of the model and the realisation are identical when the same set of use case are applied to them.

Testing a given aspect of a realisation is typically done in an indirect way as depicted in the upper right part of Figure 1. Given a model an environment model (in the form of a **test suite**, a set of tests or use cases) is constructed against which the (**system under test (SUT)** is tested.

In **manual Model ↔ Realization testing** the test designer derives manually, the test suite from a model of the SUT. The test developer than implements an autotester that hard codes this test suite, that the SUT must pass.

In **model based Model ↔ Realization testing** however, the test cases are automatically derived from the SUT model. The model based autotester interprets the model of the SUT and derives on the fly test cases from it. The model based autotester controls the SUT and observes its reactions. The model based autotester can judge, based on the observations of the SUT, whether the SUT is reacting correctly or not.

Early integration

Looking at Figure 2, we can see how a system is decomposed into two subsystems, how each subsystem gets designed and implemented in several versions. Due to the fact that models and realisations reach completion at different moments in time there is no clear point in time where we cross the design and realisation phase.

We therefore distinguish three integration phases, indicated by vertical dotted lines. The **model integration phase** starts a soon as there are part-designs of the system design available, which share at least one design interface. It stops as soon as the first unit realisation is available. The **mixed integration phase** starts a soon as the first unit realisation is available and stops as soon as the last unit realisation is available. The **realization integration phase** starts a soon as the last unit realisation is available and stops as soon as the system realisation is available.

Although the system architects are fully aware of the **interdiscipline/interproject interfaces** between the subsystems (they have identified them in the first place), they become poorly managed during the red marked time interval. **Errors** made, either **design errors** (detailing designs that violate the interdiscipline/interproject design interfaces) or **realisation errors** (realisations that violate the interdiscipline/interproject realisation interfaces) in each of those **swimlanes** will only be discovered after the composition of the subsystem realisations into the system realisation.

Because of the possibility to introduce interdiscipline/interproject interface violations very early (i.e. after decomposition) and the fact that these can only be detected very late (i.e after composition) in the development process, together with the fact that late detection results in costly repairs, we think tooling can perfectly help in managing these interdiscipline/interproject interfaces, especially when a lot of subsystemss and versions are flowing around.

The brick wall in Figure 2 symbolises the behaviour that occures when responsibilities are distributed over serveral 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 is hardly involved because they have not yet reached the point where they need to work with the interface. As a consequence they get in the end confronted with an interface which is defined from only one perspective.
Another scenario might be that both define an interface in the beginning but this interface is expressed in there own development environments and start to deviate from each other during both developments. Nobody guarantees that both interface descriptions are equal. Better would it be when there is only one interface description owned by a system architect from which specific interface descriptions are derived.
The fact that there is such a brick wall makes it easy to export your problems to someone else by just throwing it over the wall. Both parties might even insist on having such a brick wall just because of this. We think that especially tooling might help in solving these kinds of problems.

In the next subsections we will elaborate on the different integration phases because they impose different requirements on our test and integration infrastructure.

Model integration phase

In the model integration phase only model interfaces exist. The integration environment that is needed during the ***model integration phase*** is one that can support model interfaces between different structural and behaviour models and is called a **simulation environment**. The simulation environment can manage the dependencies between models by facilitating communication between simulators that run these models.

Mixed integration phase

In the mixed integration phase a mixture of model interfaces, real interfacs exist. The integration envi-

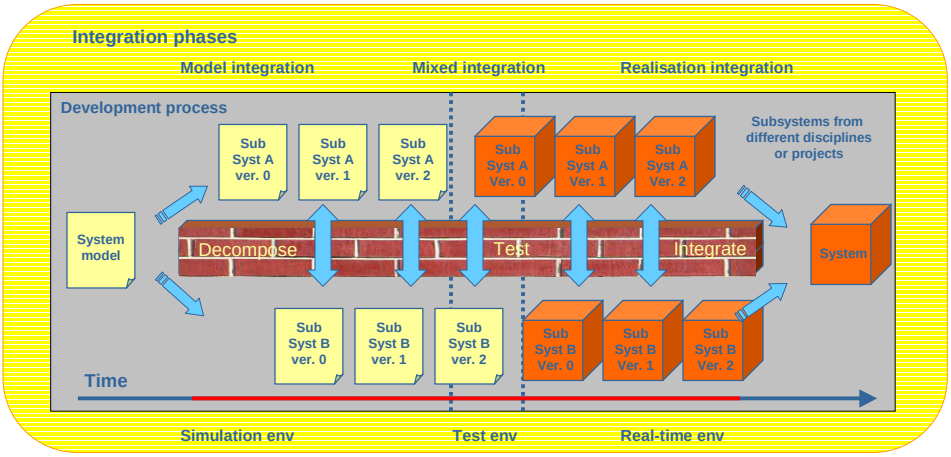


Figure 2: Early integration phases

ronment that is needed during the ***mixed integration phase*** is one that can manage both model interfaces between different (structural and behaviour) models and real interfaces between realisations and is called a **test environment**. It must be capable of bridging information flowing through model interfaces into information flowing through real interfaces.

Realisation integration phase

In the realisation integration phase only real interfaces exist. The integration environment that is needed during the ***realisation integration phase*** is one that can manage the real interfaces between different realisations and is called an **real-time environment**. A real-time environment is part of the system and is as such developed in the development process. The real-time environment must manage the control dependencies between realisations in real time.

Test and integration infrastructure

Figure 3 shows the test and integration infrastructure. Four different environments can be identified: Simulation, Prototype, Test, and Real-time. Each environment will be described in the following subsections.

For the complete test and integration infrastructure the following requirements must hold.

- The same test and integration infrastructure must be used: In each development phase, for each development level, for each discipline.
- All existing parts (simulators and realisations) need to be integrated as is, without any modification.
- All newly designed parts of the test and integration infrastructure must be based on open standards, commonware or COTS tools, to avoid vendor locks.
- The test and integration infrastructure must be open for future extensions or unforeseen interactions between environments.
- The test and integration infrastructure must be applicable for other High Precision Equipment Manufacturers. Therefore the ASML specific parts will be isolated as much as possible from the rest of the integration and test infrastructure.

Simulation environment

A simulation environment allows 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 ↔ Formal model testing, each discipline uses their own simulators, which have proven their usability within that discipline. Commonly used simulators are: Simulink[7], Visual Elite[11], LabView[2], Unigraphics[13], and SystemC[12]. 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 they are.

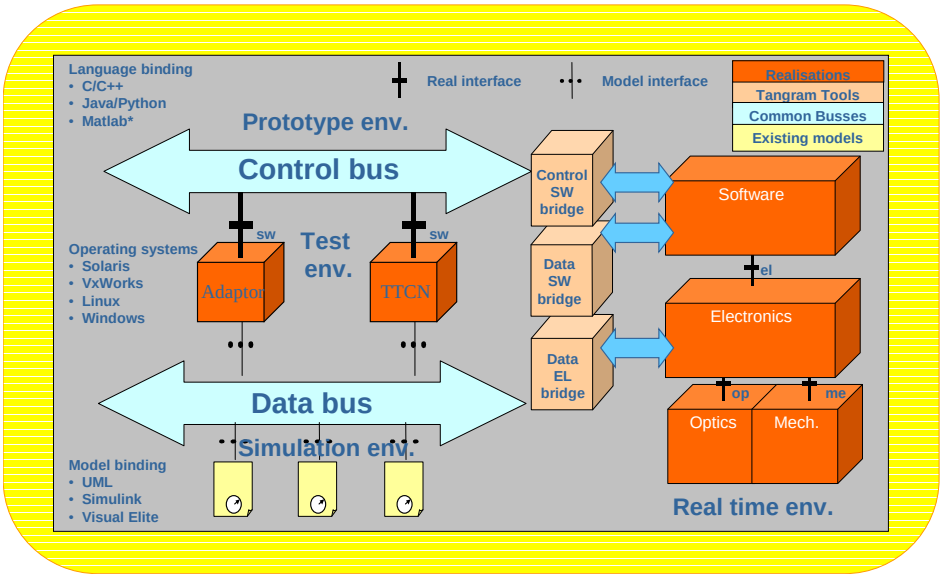


Figure 3: test and integration infrastructure

- In Whole ↔ 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.
- To facilitate the interface management, the information describing the model interfaces need to be centralised and owned by a system architect.
- To allow a modeler to stay within his/her own discipline, all interaction 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 logical time need to be synchronised 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.

Prototype environment

A prototyping environment allows execution of prototype realisations. **Prototype realisations** are realisations that implement real interfaces but their behaviour is only rudimentary implemented. The following aspects must be taken into consideration when designing the prototype environment.

- The prototyping environment must allow substitution of prototype implementations with realisations.
- For early integration, the developer must be capable to build prototype implementation in the most suitable (rapid prototype) progamming language. Commonly used languages are: C, Matlab, Python, and Java
- The prototyping environment must support different operating systems (e.g. Solaris, VxWorks, Linux and Windows). The prototyping environment must support different hardware platforms (e.g. PC, Sun workstation, IBM).

Test environment

A test environment allows a test designer to specify a test suite (a set of tests) that can be executed agains a SUT. Each test can either pass or fail. The test environment must fulfill the following aditional requirements:

- For test generation purposes and to save manhours, 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 stimulus to the SUT and must also have access to time-stamped observations of the

SUTs reactions.

- To test or diagnose the SUT in its real time environment the test environment needs full control and observability over its interfaces. Currently the SUT must be controllable and observable over three types of interfaces: a software control bus, a software data bus, and an electronical control/data bus.
- The test environment must be connected to the simulation environment to allow a partly simulated environment for the SUT while testing.
- The test environment must handle both synchronous and a-synchronous interactions with the SUT.

We selected the TTCN3[6] test language and tooling 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
- TTCN3 is designed for and by test developers
- TTCN3 is an open standard
- TTCN3 abstracts away all SUT specific details
- TTCN3 allows uniformly testing over different real interfaces.
- Robust and mature IDE's exist that help the test developer in writing, debugging and managing his test specifications.
- Several Tool vendors provide TTCN3 tools.
- A vast user community exists around TTCN3: Automotive, Telecom companies

The test developer now has the opportunity to write an executable test to test the SUT on functionality, performance, interoperability, or conformance.

The progamming model of the TTCN test language is a fully programmable closed language and is based on communicating sequential processes CSP. Test cases can run in parallel. The SUT is accessable through ports. The test cases can be connected to these ports with buffered channels.

Real time environment

The real time environment is the environment in which the system operates. The SUT within Tangram will be the ASML Twinscan machine (see Figure 4) or parts of it. Most of the software interactions are not time critical. Some interactions close the electronics have strict real time requirements. The real electronical interface of the SUT is mostly generic in the sense that generic data aquisition devices can be bought that connect to this interface. The real software interface of the SUT is ASML specific w.r.t. the client/server architecture, the interface descriptions, the message format, the protocol used, and the server address model, and the application programmers interface.

Standard busses

For scalability reasons, the test and integration infrastructure is based on a bus topology. Using a bus topology with n participants, only $O(n)$ connections need to be developed compared to $O(n^2)$ peer to peer connections. An open standard bus avoids vendor lock (i.e. no single vendor can control the future development of such a bus) and assures interoperability between the participants.

Control bus: CORBA

The prototype, test, and real time environments are all attached to a control bus. OMG's CORBA[8] is used as standard that describes its functionality. OmniOrb a freeware Orb is used as commonware that implements such a control bus. Within Tangram we will concentrate on connecting these three environments to this control bus. The rationale for selecting CORBA is:

- CORBA is based on decades of experience in driving reactive systems
- CORBA is designed for and by software developers
- CORBA is an open OMG standard
- CORBA abstracts away all transport specific details.
- CORBA is based on the proven proxy pattern (i.e allows uniform calling of services over different progamming languages, operating systems, and communication hardware)
- Several Tool vendors provide CORBA and CORBA service implementations.
- A vast demanding user community exists around CORBA: Defense, Aerospace, and Manufacturing companies

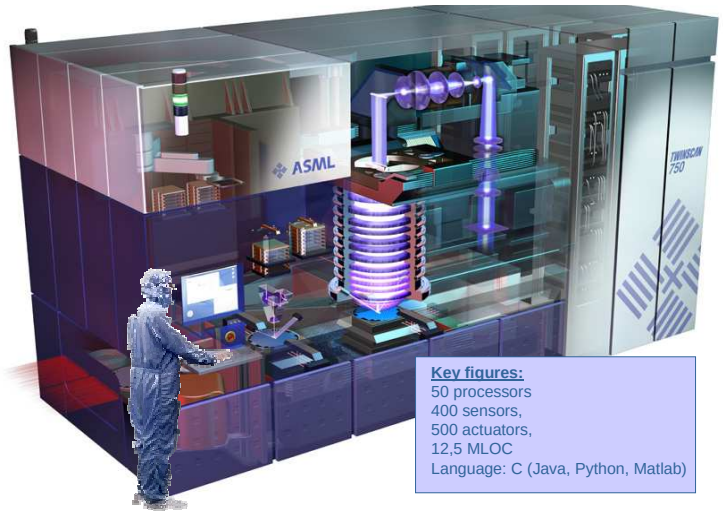


Figure 4: The system under test: The ASML Twinscan machine

Data bus: DDS

The simulation, test, and real time environments are all attached to a data bus. OMG's data distribution service[9], a CORBA service, is used as standard that describes its functionality. RTI's NDDS[1] a commercial product is used as commonware that implements such a data bus. Within Tangram we will concentrate on connecting these three environments to this data bus. The rationale for selecting DDS is:

- DDS is based on decades of experience in driving real-time reactive systems
- DDS is designed for and by software developers
- DDS is an OMG standard
- DDS is based on the proven publish/subscribe pattern.
- DDS describes a simple application programmers interface (API) with an array quality of service (QoS) configurations.
- DDS abstracts away all transport specific details.
- Several Tool vendors provide DDS tools.
- A vast demanding user community exists around DDS: Defense and Aerospace companies

Bridges

Because we try to use proven and existing simulators and commonware we can concentrate on connecting environments together. The technique for that is based on bridging. A bridge allows bidirectional flow of data and control between two worlds. A bridge does not add extra functionality to a system it just reformats information from one world into the other and vice versa. The bridges that can be identified within the test and integration infrastructure will be discussed separately in the following subsections.

CORBA to SUT Software bridge

The CORBA to SUT SW bridge opens up the SUT for control over the software control bus. Fortunately the software control interface implemented by the ASML execution environment greatly resembles the interface of the CORBA control bus. The ASML specific interface descriptions, expressed in so called ddf files, can be translated into the standard CORBA **interface description language** (IDL). Using these IDL files a bridge can be generated automatically. Therefore, the bridge can follow each interface modification for each build of each release. This bridge can intercept function calls at each selected software interface. Participants on the CORBA bus can act as clients of the SUT, or as a server for the SUT, or both at the same time.

TTCN to CORBA bridge

By building a TTCN3/CORBA bridge we succeeded in attaching Telelogic's Tau Tester[3] to the

CORBA control bus. The bridge can be generated from the same IDL descriptions that were used in the CORBA to SUT bridge. From a testers point of view the complete software interface to the SUT is described in TTCN interfaces: types, functions, interaction ports etc.

TTCN to DDS bridge

The TTCN to DDS bridge allows an information flow from the TTCN3 test environment to the test data bus and vice versa.

DDS to SUT Electronics bridge

The DDS to Electronics bridge connects the DDS data bus to the electronics interface of the SUT. National Intruments' Labview[2] will be used as 'commonware' to implements this bridge.

Model to real interface adaptor

When connecting models to realizations the sparse information that flows over a model interface must be converted into an information rich data and control flow that a realisation interface needs. When timing is an issue the adaptor needs to convert logical time into real-time and vice versa (e.g. triggering calls at some point in real time, and timestamping replies). An **interface adaptor** is just doing that. An interface adaptor is connected both to the DDS data bus and the CORBA control bus and is progammable.

When converting model interfaces into real interfaces extra information is added to the real interface. This extra data are called **test vectors**. When converting real interfaces into model interfaces only portions of the data flow needs to be filtered out of the information rich data coming from the real interface. Every programmable application that is connected to both the control and data bus can act as an adaptor, like TTCN3 for instance.

Case studies

With the help of concrete case studies the applicability, usability, and robustness of our test and integration infrastructure will be assessed. The cases must preferrably cover the 4 different kinds of testing, for each development phase and level. The motivating examples will be sorted according to the importance as perceived by the ASML developers. As a first case we are thinking of testing the hardware software interface, where the interface is described as a memory map.

Future work

Future work might include: management tools (e.g. a time manager for the simulation environment, integration of requirement management tools, and versioning systems), diagnostic tools (like UML model animators and code instrumentation), and test tools (test case generators and extensions for timed testing).

Conclusions

We have presented a generic test and integration infrastucture based on COTS products. Some parts are already glued together with relatively low effort. In our own first experiments we could already appreciate the flexibility of the infrastructure. Real ASML case studies must show the added value of the infrastructure. This will be the main remaining challenge for the rest of project.

Acknowledgements

We gratefully acknowledge the feedback from the discussions with our TANGRAM project partners from ASML, Eindhoven University of Technology, Embedded Systems Institute, Delft University of Technology, Twente University and the University of Nijmegen.

References

[1] Ndds, http://www.rti.com, Real-Time Innovations, 2005.

[2] labview, http://ni.com/labview, National Instruments.

[3] Tau tester, http://www.telelogic.com/products/tau/tautester/index.cfm, Telelogic, 2000.

[4] ASML, http://www.asml.com.

[5] TANGRAM, http://www.esi.nl/tangram/, 2003.

[6] TTCN-3 standard. http://www.etsi.org/ptcc/ptccttcn3.htm, 1998-2003.

[7] matlab/simulink, http://www.mathworks.com/products/, Mathworks.

[8] CORBA, http://www.omg.org/technology/documents/formal/corba_iiop.htm,2003.

[9] Data distribution service for real-time systems. http://www.omg.org/technology/ documents/formal/data_distribution.htm, 2005.

[10] UML 2.0, Unified modeling language 2.0, http://www.uml.org/, 2005.

[11] Summits visual elite, http://www.summit-design.com.

[12] System c 2.1, http://www.systemc.org/, 2005.

[13] Unigraphics, http://www.ugs.com/products/nx/.

Contact Information

Will Denissen

TNO Science and Industry
P.O. Box 155, NL-2600 AD Delft
The Netherlands
Will.Denissen@tno.nl