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.

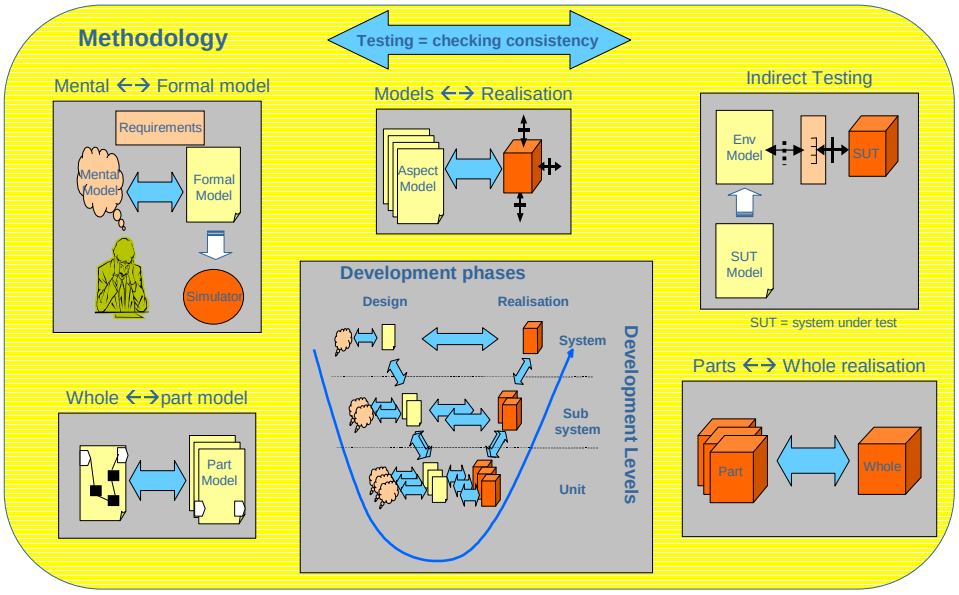

Figure 1: Different kinds of testing at different development levels and phases

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-

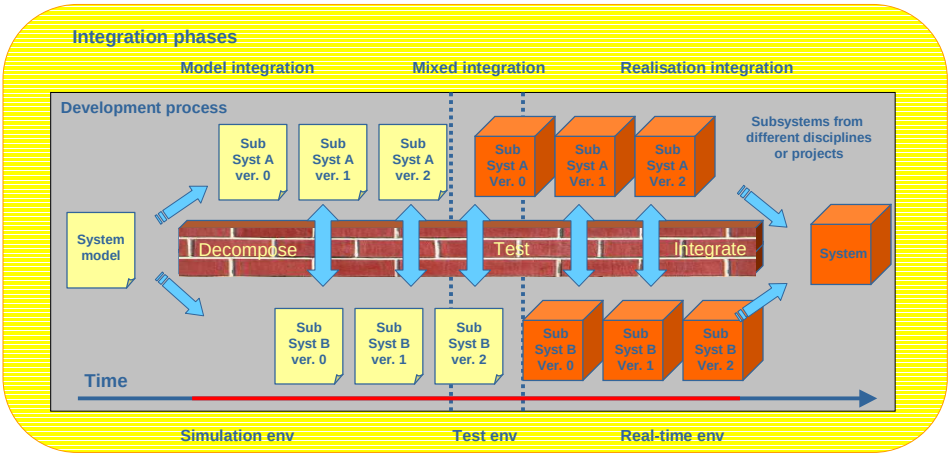

Figure 2: Early integration phases

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.

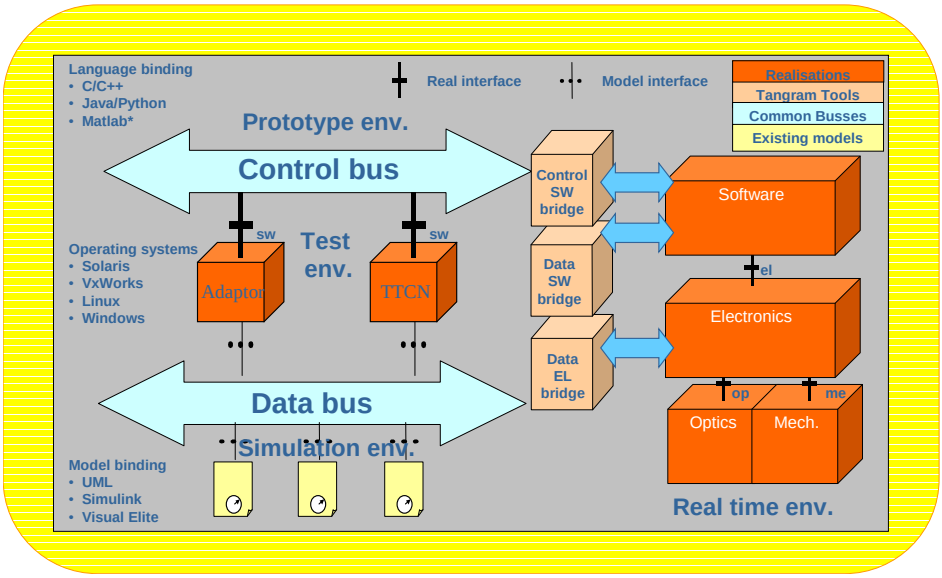

Figure 3: test and integration infrastructure

- 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

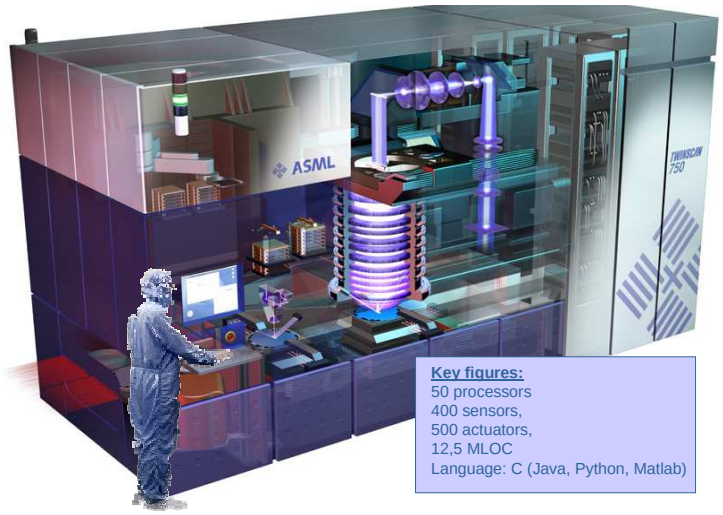

Figure 4: The system under test: The ASML Twinscan machine

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