A Model-based Integration and Testing Method to Reduce System Development Effort

N.C.W.M. Braspenning¹ J.M. van de Mortel-Fronczak² J.E. Rooda³

Department of Mechanical Engineering, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract

New methods and techniques are needed to reduce the very costly integration and test effort (in terms of lead time, costs, resources) in the development of high-tech multi-disciplinary systems. To facilitate this effort reduction, we propose a method called model-based integration. This method allows to integrate formal executable models of system components that are not yet physically realized with available realizations of other components. The combination of models and realizations is then used for early analysis of the integrated system by means of validation, verification, and testing. This analysis enables early detection and prevention of problems that would otherwise occur during real integration, resulting in a significant reduction of effort invested in the real integration and test phases. This paper illustrates how models of components, developed for model-based integration, can be used for automated model-based testing, which allows time-efficient determination of the conformance of component realizations with respect to their requirements. The combination of model-based integration and model-based testing is practically illustrated in a realistic industrial case study. Results obtained from this study encourage further research on model-based integration as a prominent method to reduce the integration and test effort.

Keywords: Model-based integration, model-based testing, industrial case study

1 Introduction

High-tech multi-disciplinary systems like wafer scanners, electronic microscopes and high-speed printers are becoming more complex every day. These systems, consisting of numerous hardware and software components connected through many inter-
faces, have to meet the strict quality requirements set by the customer in market conditions where lead time (in the context of time to market) is critical.

This increasing system complexity also increases the effort needed for the, so-called, integration and test phases. During these phases, the system is realized by combining component realizations (implementations) and, subsequently, tested against the system requirements. In most of the current development processes, the integration and test phases start when the component realizations become available, and these phases should be finished before the system’s shipment date agreed with the customer. As a result, the main lead time burden is shifting from the design and implementation phases to the integration and test phases [6]. Furthermore, finding and fixing integration and test problems late in the system development process (which is the case in the current approach) can be up to 100 times more expensive than finding and fixing the problems during the requirements and design phases [3].

Many research activities that aim at countering this increase of development effort (in terms of lead time, costs, resources) involve model-based techniques like requirements modeling [8], model-based design [13,17], model-based code generation [9], and hardware-software co-simulation [19]. In most cases, however, these model-based techniques are investigated in isolation, and little work is reported on combining these techniques into an overall method. Although model-based systems engineering [18] and OMG’s model-driven architecture [16] (for software only systems) are such overall model-based methods, these methods are mainly focusing on the requirements, design, and implementation phases, rather than on the integration and test phases. Furthermore, literature barely mentions realistic industrial applications of such methods, at least not for high-tech multi-disciplinary systems.

Our research within the TANGRAM project [20] focusses on a method of model-based integration, in which model-based techniques are developed and applied in industry in order to reduce the integration and test effort. In this method, models of system components that are not yet physically realized are integrated with available realizations of other components, establishing a model-based integrated system. This model-based integrated system is used for analysis on the system level before all components are realized. This early analysis takes integration and test effort out of its critical position and enables the developers to detect and prevent problems that would otherwise occur during real integration (i.e. earlier and thus cheaper), eventually resulting in a reduction of effort. Furthermore, the model-based analysis techniques help in clarifying and improving the often difficult decomposition of requirements and design of the system (usually clear and certain) into the requirements and designs of all components (usually unclear and based on assumptions). These improved insights in the system decomposition eventually improve the quality of the system realization.

After sufficient and successful validation and verification, the models used for model-based integration are good representations of the requirements and the designs of the corresponding components. When the realization of such a component becomes available, it would be interesting to determine whether this realization conforms to the model (and thus to the requirements and design), before integrating
it into the system. When discrepancies between the realization and the model of a component are found during this analysis, this means that either a problem in the realization is found that needs to be fixed, or it pinpoints incomplete or unclear parts of the requirements and the design that need improvement. Testing the conformance of a component realization with respect to a specification model is the topic of model-based testing research [7], for which several model-based test tools [14] are available.

In this paper, we describe how the model-based integration method is extended with model-based testing in order to determine whether a component realization conforms to the model developed for model-based integration. Both the method and the extension have been applied to an industrial case study concerning the ASML [1] wafer scanner.

The structure of the paper is as follows. First, the model-based integration method and the accompanying techniques and tools are introduced in Section 2. Section 3 describes how this method is extended with model-based testing. The case study application and results are presented in Section 4. Finally, the conclusions are drawn and discussed in Section 5.

2 Model-based integration

In current industrial practice, the system development process is subdivided into multiple concurrent component development processes. Subsequently, the resulting components are integrated into the system. The development process of a component $C_i$ consists of a requirements definition phase, a design phase, and a realization phase. Each of these three phases results in a different representation form of the component, namely the requirements, the design, and the realization of the component, denoted here as $R_i$, $D_i$, and $Z_i$, respectively. In the development process of a system $S$ that consists of multiple components, e.g. components $C_1$ and $C_2$, the system requirements and system design, denoted here as $R$ and $D$, respectively, precede the development processes of the components. The realization of system $S$ is the result of the integration of realizations $Z_1$ and $Z_2$ of components $C_1$ and $C_2$. This integration is denoted as $\{Z_1, I_{12}, Z_2\}$, where $I_{12}$ denotes the infrastructure connecting $Z_1$ and $Z_2$. Figure 1 shows the development process of system $S$.

![Fig. 1. Current system development process](image)

In this way of working, only two types of system level analysis can be applied.
On one hand, the consistency between requirements and designs on component and on system level can be checked, e.g. \( R \) vs. \( R_1, R_2 \) and \( D \) vs. \( D_1, D_2 \) (which usually means reviewing lots of documents). On the other hand, the integrated system realization, e.g. \( \{ Z_1, I_{12}, Z_2 \} \), can be tested against the system requirements and design, \( R \) and \( D \), provided that all components are realized and integrated. This means that when problems occur and need to be fixed during the integration and test phases, the effort invested in these phases immediately increases, directly threatening on-time system shipment.

We propose a method of model-based integration to reduce the integration and test effort. In this method, the designs of the components (e.g. software, mechanics, electronics) are represented by formal executable models of communicating concurrent processes, expressed in a process algebra \([2]\). The resulting models, denoted here as \( M_i \) for a component \( C_i \), enable formal analysis of component and system behavior. With model validation (e.g. simulation), the behavior of certain traces of the system model, e.g. \( \{ M_1, I_{12}, M_2 \} \), can be inspected and compared with the intended system design \( D \). With model verification (e.g. model checking), it can be checked whether certain properties derived from the system requirements \( R \) and system design \( D \), are satisfied by the system model. Such model-based system analysis helps in evaluating and improving the correctness of the decomposition of the requirements and design of the system into the requirements and designs of the components.

Besides that models enable these additional analysis techniques, they can also replace realizations for integration testing. This means that integrations of models and realizations can be tested against the system requirements and design without the necessity that all component realizations are available. As models are usually available earlier than realizations, testing on the system level can start earlier. Earlier testing allows earlier detection and prevention of system integration problems, which should lead to a reduction of the effort invested during real integration and testing. Figure 2 shows the development process of system \( S \) in the model-based integration method, where \( M_i \) denotes a model of component \( C_i \) (based on its design \( D_i \)), and where \( I_{12} \) denotes an infrastructure that allows the integration of components \( C_1 \) and \( C_2 \), both represented by either a model or a realization. Note that with code generation, the realization of a software component \( Z_i \), could be based on its model \( M_i \).

In our research, components are modeled as processes in the timed process algebra \( \chi \) \([23]\) (a simplified version of hybrid \( \chi \) \([24]\)), developed at the Systems Engineering Group, Eindhoven University of Technology. For each component, the internal behavior of the process (assignments, guarded alternatives, guarded repetitions, delays) and the external communication with other processes (sending, receiving) is modeled. The system of components is modeled as the parallel composition of all component processes, connected by communication channels. The \( \chi \) toolset contains a simulator to simulate such a system model. Furthermore, several back ends are added to the \( \chi \) toolset to enable other analysis techniques like model verification, distributed/real-time simulation, and software/hardware-in-the-loop testing,
Fig. 2. System development process in the model-based integration method

which are all used in the model-based integration method.

As mentioned in the introduction, it would be interesting to determine whether a component realization, when it becomes available, conforms to the model used for model-based integration, before the component realization is integrated with other components. Therefore, the set of analysis techniques mentioned previously is extended with model-based testing.

3 Model-based testing

Model-based testing provides theories and tools for automated testing, which is receiving more and more attention as an alternative to manual and scripted testing, which are becoming incapable of finding all errors within time. In model-based testing, a formal specification model of a component is used to generate tests from, and these tests are executed on-the-fly on the component realization, resulting in a ‘pass’ or ‘fail’ test verdict. In the TANGRAM project, the test tool TORX [21], based on the theory of input-output conformance (ioco), is used for model-based testing. While extensions towards timed testing [4] and testing with more complex data [12] are being developed, the version of TORX used in our experiments only supports model-based testing of untimed, discrete-event systems without complex data.

As previously mentioned, the models used for model-based integration are developed in $\chi$, currently not supported by TORX. However, TORX supports TROJKA [10], a slightly modified version of PROMELA, the specification formalism for the model checker SPIN [15]. SPIN is also used in the model verification back end of the $\chi$ toolset [5], for which a translation scheme from $\chi$ to PROMELA is developed [22]. By combining the translation of $\chi$ to PROMELA and the model-based testing capabilities of TORX, the conformance of a component realization with respect to the $\chi$ model used for model-based integration can be determined. This approach is visualized in Figure 3 for system $S$, in the case that the realization of component 2 becomes available first. In this figure, the $\chi$ model of component 2, $M_{2,\chi}$, used for model-based integration with $M_{1,\chi}$, is translated into a PROMELA equivalent, $M_{2,P}$. Subsequently, TORX tests whether the realization $Z_2$, used for model-based integration with $M_{1,\chi}$ and later for real integration with $Z_1$, is ioco conforming to
The procedure for the model-based integration method extended with model-based testing is as follows:

1. Modeling of components, e.g. $M_1$ and $M_2$, based on their designs.
2. Validation and verification of the integrated system with models only, e.g. \{M_1, I_{12}, M_2\}, using $\chi$ simulation and SPIN model checking.
3. For each component:
   (a) Replacement of model by realization, e.g. $M_2$ by $Z_2$, using an infrastructure that enables the integration with the other components.
   (b) Model-based testing of component realization with respect to model, e.g. $Z_2$ with respect to $M_2$, using TorX.
   (c) Model-based integration testing of combined models and realizations, e.g. \{M_1, I_{12}, Z_2\}, by executing test cases derived from $R$ and $D$.
4. Integration testing of the complete system realization, e.g. \{Z_1, I_{12}, Z_2\}, by executing test cases derived from $R$ and $D$.

Note that in this paper, different test techniques are used for component testing (step 3b, using model-based testing) and for system integration testing (step 3c and 4, using execution of manually derived test cases on integrated models and realizations). The reason for this is that testing only the aspects that are covered by the model-based testing techniques used in the method is not sufficient for the systems considered in our research, while the techniques used in model-based and real integration testing are capable of testing other important aspects (e.g. time and data) as well. Nevertheless, model-based testing could in principle be used for system integration testing, by using the integrated system model, e.g. \{M_1, I_{12}, M_2\}, as basis for test generation and by using the integrated component realizations, e.g. \{Z_1, I_{12}, Z_2\}, as system under test. This is possible as long as the size and complexity of the integrated system model is not beyond the limitations of the test tool (model abstraction can be used to solve this issue), and as long as the test tool can
access the required test interfaces of the integrated system realization.

Furthermore, this paper does not consider a compositional testing technique, as described in [25], to imply conformance of the integrated system based on the conformance of the individual components. This compositional testing technique requires that the models are complete, i.e. explicit specification of all allowed responses for any possible input, which is not the case for the models developed for model-based integration.

4 Case study: ASML laser subsystem

The model-based integration and testing method described in the previous sections has been applied to a case study concerning the laser subsystem of the ASML wafer scanner, which is used in lithography industry for the production of integrated circuits or chips. In a wafer scanner, the lithographic process of exposing a silicon wafer with a certain pattern (corresponding to one layer of a chip) takes place. The laser subsystem of a wafer scanner generates the laser light that is used for this lithographic process. A controller, that is part of the wafer scanner, communicates with the laser subsystem in order to get the required amount of laser light for each exposure. This communication is realized by two bi-directional interfaces: a serial (RS232) interface for commands and responses and a parallel interface for multiple status signals. Experience has shown that the interface between the wafer scanner controller and the laser subsystem is difficult to understand, integrate and diagnose. This is mainly caused by the fact that the laser subsystem is produced by a third party manufacturer, meaning that the ASML engineers do not have full insight in and control over the behavior as implemented in the laser subsystem. Therefore, correct integration of the wafer scanner controller and the laser subsystem is an important aspect for the performance and reliability of the wafer scanner.

Due to safety and cost reasons, a hardware laser simulator has been used in the case study instead of the real laser. This hardware laser simulator has the same software and electronic components and is specified to behave the same as the real laser, however it does not have the physical components to generate laser light. The laser simulator has been developed by ASML and is used for testing the software and electronics of the wafer scanner controller, without the need for a real laser (including the required space and facilities). Because the laser simulator is used for testing of the wafer scanner controller, it is important that the behavior of the laser simulator satisfies the behavior specification of the real laser, in order to avoid faulty test outcomes and, even worse, faulty fixes in the wafer scanner controller.

In the case study, we illustrate the application of steps 1, 2, 3a, and 3b of the procedure described in the previous section. Models of the wafer scanner controller and of the laser subsystem have been developed and integrated in $\chi$ and analyzed by $\chi$ simulation and Spin model checking. Subsequently, the conformance of the hardware laser simulator with respect to the PROMELA equivalent of the laser subsystem $\chi$ model is determined using model-based testing with TorX. The application and the results of each of these steps are presented in the sequel.
Step 1: Modeling of components

The specification documents of the laser subsystem and of the communication with the wafer scanner controller have been taken as a starting point for modeling the components. Our experience is that the modeling activities help in finding and clarifying errors, inconsistencies, and incompleteness in the requirements and design documents.

Figure 4 shows the processes (circles) and communication channels (arrows) that have been modeled as described in the sequel. Note that processes IO, LC, and LS are all part of the laser subsystem.

**Wafer scanner controller C** This process can be configured (using an external configuration file) to execute specific command sequences for behavior validation, e.g. operational sequences as specified in the documentation.

**I/O interface IO** This process receives the commands from C and, after the handling of the commands by LC or by LS, it sends the responses back to C.

**Laser communication LC** This process receives the commands from C (passed through by IO) and, according to its configuration (stored in an external configuration file), it performs the necessary actions (e.g. a state change) and creates the corresponding responses.

**Laser state LS** This process keeps track of the laser state, which is used by IO for the response to a laser state query command by C.

Each process definition contains the state and temporal behavior of the component and the communication behavior including the data that is communicated. Here, the communication involves both the serial and the parallel interface of the laser subsystem. Furthermore, the laser subsystem processes contain the *error handling* of ‘unknown’ commands (unspecified commands) and ‘bad context’ commands.
As previously mentioned, the temporal behavior and the complex data, frequently used in the models developed for model-based integration, are not supported by the version of TORX used in the case study. Also the $\chi$ to PROMELA translation scheme used in step 2 of the case study and the PROMELA language itself have their limitations concerning time and data. Therefore, the original $\chi$ processes have been made suitable for translation to PROMELA and for model-based testing with TORX by applying abstractions from time and complex data. These abstractions do not influence the state and communication behavior of the system that is analyzed in steps 2 and 3 of the case study.

The resulting $\chi$ processes are configurable in the sense that the command sequences of the wafer scanner controller and the behavior of the laser subsystem can be modified in external configuration files without modification and recompilation of the $\chi$ system model. This flexibility in the modeling of system behavior has shown its advantage during the case study when it became clear that the hardware laser simulator was not available for a certain laser type and another laser type had to be modeled.

The integrated system model, consisting of both the wafer scanner controller and the laser subsystem, is obtained by the parallel composition of all $\chi$ processes. Here, the parallel composition operator, as defined in the process algebra $\chi$, is used as infrastructure between the two components (corresponding to $I_{12}$ in Figure 3). The resulting $\chi$ system model contains 350 lines of code in total, including the necessary data definitions and functions.

Step 2: Validation and verification of integrated models

The integrated system model developed in step 1 has been validated using the model simulator of the $\chi$ toolset. Several simulation runs have been executed, in which the command sequences from the specification documents, e.g. for switching the laser subsystem on and off, are specified in the configuration file of the wafer scanner controller process $C$. Based on the simulation results, the laser subsystem behavior conforms to the specification documents for all command sequences.

Besides validation, also certain properties of the system model have been verified by SPIN model checking. To perform this type of analysis, the $\chi$ system model has been translated into PROMELA, using the translation scheme from [22]. As previously mentioned, this translation scheme and the PROMELA language itself have their limitations regarding time and data, however the abstractions applied in step 1 result in a model that is suitable for translation to PROMELA. The resulting PROMELA system model contains 1850 lines of code, including 900 lines for representing the equivalent of all data definitions used in the $\chi$ model and 300 lines for representing the equivalent of all functions used in the $\chi$ model as additional processes.

Besides a model expressed in PROMELA, the properties to be verified have to be specified for model checking with SPIN. Eight properties of the system have been verified: absence of deadlock, a system invariant concerning the translation of
a specific \( \chi \) statement, and six model specific behavioral properties. Checking the absence of deadlock, or invalid end states, is a standard option in SPIN. The system invariant has been checked by defining a safety property on the precondition and the postcondition of the translated \( \chi \) statement, which is expressed in the linear temporal logic (LTL) formula \( \Box (\text{precondition} \rightarrow \Diamond \text{postcondition}) \). Two of the model specific behavioral properties concern the allowed order of state transitions, e.g. from the ‘off’ state, the laser state can only become ‘standby’ without being ‘on’ in between, which is expressed in the LTL formula \( \Box (\text{state}_{\text{off}} \rightarrow \neg \text{state}_{\text{on}} \cup \text{state}_{\text{standby}}) \).

The other four model specific behavioral properties concern all possible actions and responses to each command. For example, when the laser receives the ‘go off’ command, while it is in the ‘off’ state or in the ‘on’ state, it stays in the current state and responds with ‘not allowed’, or, when it receives the ‘go off’ command while it is in the ‘standby’ state, it goes to the ‘off’ state and responds with ‘state off’. This is expressed in the LTL formula:

\[
\Box (\text{cmd}_{\text{go off}} \rightarrow (\text{state}_{\text{off}} \cup \text{rsp}_{\text{not allowed}}) \lor (\text{state}_{\text{on}} \cup \text{rsp}_{\text{not allowed}}) \lor (\text{state}_{\text{standby}} \cup (\text{state}_{\text{off}} \land (\text{state}_{\text{off}} \cup \text{rsp}_{\text{state off}}))))
\]

All these properties have been verified and found to be correct. Based on these verification results, together with the correct simulation results, there is enough confidence that the model is a good representation of the requirements and the design of the laser subsystem, and therefore a good basis for automated model-based testing of the hardware laser simulator.

**Step 3a/3b: Model-based testing of the laser subsystem**

In this step of the case study, we use the model developed in step 1, and validated and verified in step 2, for automated model-based testing of the hardware laser simulator using the T0RX test tool. As visualized in Figure 3, T0RX is connected to the model on one side, and to the realization (in this case the hardware laser simulator) on the other side.

To connect the model to T0RX, the PROMELA model of step 2 has been slightly modified, resulting in a TROJKA model that is suitable for T0RX. A TROJKA model must be an open model, meaning that some channels of the processes are not connected to other processes. These unconnected channels are the so-called observable channels, on which test inputs can be given and on which test outputs can be observed. In the case study, an open TROJKA model with observable channels has been obtained by removing the C process from Figure 4 and by giving the unconnected command and response channels of process IO the special channel attribute OBSERVABLE. The resulting TROJKA model of only the laser subsystem contains 1000 lines of code, including 300 lines for representing all data definitions and 300 lines for representing all functions.

To connect the realization to T0RX, the abstract commands from the TROJKA model need to be transformed into the real commands for the realization and vice
versa for the responses of the realization. As previously mentioned, the real commands and responses for the hardware laser simulator (as well as for the real laser) are sent and received through a serial interface and a parallel interface. Unfortunately, direct access of these interfaces from outside, as required for model-based testing with TorX, is limited. While functionality for direct access from outside is provided for the serial interface, this is not the case for the parallel interface, because this interface uses an ASML specific communication protocol, embedded in the electronics of the wafer scanner controller. This limitation in interface access from outside drastically reduces the laser subsystem behavior that can be automatically tested, since the larger part of the laser subsystem state space can only be reached by using the parallel interface. However, the reduced behavior that can be tested with serial communication only is still sufficient to demonstrate automatic testing with TorX based on the models developed for model-based integration.

For correct communication over the serial interface, an adapter component has been written in Python that accepts test inputs from TorX, performs the necessary transformations of the abstract model commands into real laser commands (e.g. a left justified string of 128 bits), and uses the provided direct access functionality to send the command over the serial interface. The response to the command is received from the laser subsystem, and after it is transformed back into the abstract response used in the model, it is sent back to TorX by the adapter component.

Now both the model and the realization have been connected to TorX, the conformance of the hardware laser simulator with respect to the model can be determined by model-based testing. For all three test runs that have been performed, a random test selection strategy is used to select the test inputs from the set of commands in the Trojka model. The selected commands are sent to the laser simulator through the adapter and the provided direct access functionality, and subsequently the responses from the laser simulator are observed and compared with the behavior specified in the model.

The first test run had a limited depth (less than 20 events) and took less than ten seconds until a discrepancy between realization and model was found. To clarify the discrepancy found, Figure 5 shows the state diagram of the laser subsystem that has been automatically tested (i.e. using the serial interface only). In this figure, the nodes depict the states of the model, the solid edges depict the commands sent over the serial interface (starting with ‘LS’), and the dashed edges depict the responses to the commands (starting with ‘LS’ or ‘??’). The central states at the top and bottom denote the actual laser states ‘off’ and ‘standby’, numbered ‘00’ and ‘03’, respectively. The ‘state change’, ‘state query’, and ‘bad context’ states are intermediate states between the commands and the corresponding responses. Note that any other command not shown in the figure results in an ‘unknown command’ response (‘??=00’).

Figure 6 shows the message sequence chart of the first model-based test run, where the ‘TDRV’ thread represents the test tool, the ‘iut’ (implementation under test) thread represents the realization, and the ‘out’ thread represents the output of the test run. Note that the ‘=’ and ‘?’ characters are replaced by ‘_eq_’ and ‘_QM’,
Fig. 5. Laser subsystem behavior that has been tested

respectively, because they are not allowed to be used in PROMELA/TROJKA. When the hardware laser subsystem is in the ‘standby (03)’ state (the response to ‘LS?’ is ‘LS=03’, fourth arrow from below), it receives the command ‘LS=03’ (third arrow from below) and responds with the current laser state, ‘LS=03’ (second arrow from below). However, according to the specifications (see Figure 5), giving a command to go to the current state (here the command ‘LS=03’ in the ‘standby (03)’ state) should result in a ‘bad context’ response (‘??=02’), which is indicated by the last ‘Expected’ arrow in Figure 6.

This discrepancy, resulting in a ‘test fail’ verdict, means that the realization is not conforming to the model. Further diagnosis has shown that this non-conformance is due to an incorrect implementation of the error handling behavior of the hardware laser simulator. Directly fixing this error in the laser simulator software was impossible, because the required knowledge and tools were not available at that moment. Therefore, in order to enable further testing, a small modification has been made in the model such that for the next test run there is no discrepancy between model and realization for the handling of the ‘bad context’ error.

The second test run had a limited depth as well (less than 20 events) and again took less than ten seconds until another discrepancy was found, involving the handling of the ‘unknown command’ error. According to the laser subsystem specifications, any laser command other than ‘LS=00’, ‘LS=03’, or ‘LS?’, for instance ‘LS=01’, is an unknown command and should thus result in an ‘unknown command’ response (‘??=00’). However, when such an ‘unknown’ command (e.g. ‘LS=01’) was selected, which is an allowed test input, the laser simulator responded with the current laser state, as if the laser state query command ‘LS?’ was given. Further
diagnosis has shown that also this non-conformance is due to an incorrect implementation of the error handling behavior of the hardware laser simulator. To enable further testing, the set of allowed test inputs to be selected by TorX has been restricted to known commands only, i.e. ‘LS=00’, ‘LS=03’, or ‘LS?’, such that for the next test run the ‘unknown command’ error handling behavior of the laser simulator will not be tested.

The third test run, in which the discrepancies found in the first two test runs are not detected any more, kept going for a long time (test depth of more than 1000 events, taking more than 15 minutes), without finding new discrepancies. Although no coverage metrics have been applied (as this is not a current feature of TorX), the test results of the third test run provided enough confidence that no other discrepancies between the realization and the model would be found.

The two implementation errors that have been found by automatic model-based testing are both related to the error handling behavior of the laser simulator. After discussion with the ASML engineers, it became clear that the laser simulator is mainly used for testing the wafer scanner controller under nominal behavior conditions. Although the errors may appear to be trivial and should normally not be
encountered during nominal testing with the laser simulator, this experiment shows that such errors are not easily detected in the current industrial way of working, and that a more systematic approach like model-based integration and testing certainly has potential. Furthermore, when these errors would remain undetected, they may still have a substantial impact in the case that the wafer scanner controller contains errors related to the laser simulator errors. In that case, the errors in the wafer scanner controller remain hidden when the development tests rely on the laser simulator, and these errors may cause problems later when the wafer scanner controller is used together with a real laser in a real production environment.

In the testing experiments that have been performed, only the relation between individual commands and responses has been tested using the regular testing features of TorX. However, it would also be interesting to test the specific behavioral properties as verified in step 2 of the case study, involving relations of subsequent state transitions and combinations of responses and current states. Focussing model-based test runs towards such specific behaviors can be achieved by defining test purposes, a feature that is supported by TorX [11].

5 Conclusions

A method of model-based system integration with $\chi$ is extended with model-based testing using TorX. Both the method and the extension are successfully applied in a realistic industrial case study to practically illustrate the advantages of model-based validation, verification, and testing, by means of $\chi$ simulation, SPIN model checking, and test generation and execution with TorX, respectively. These model-based analysis techniques facilitate detection of documentation errors and provide automatic detection of non-conformance of a component realization with respect to the corresponding requirements.

The case study presented is an instructive investigation of the advantages and challenges of combining model-based integration and model-based testing. The use of process algebra $\chi$, as a basis for model-based integration, allows easy specification of state, temporal and communication behavior of components; moreover, several complex data structures are supported in $\chi$. Models implemented in $\chi$ can easily be integrated, allowing the application of different model-based system analysis techniques. With further improvements of these analysis techniques, e.g. time and data extensions for SPIN and TorX, and by improving and automating the back ends of the $\chi$ tool set for verification with SPIN and for model-based testing with TorX (either directly or via PROMELA), we can achieve a powerful environment for formal model-based system analysis and automated test generation. Additionally, more work is needed on facilitating the integration of models and realizations. Currently, an infrastructure that allows straightforward coupling of models and realizations is under development. This model-based integration infrastructure should be capable of dealing with the issues of synchronous/asynchronous communication and real-time execution of distributed components.

This paper practically illustrates a formal approach for early detection and pre-
vention of system integration problems, on one hand, and time-efficient determination of conformance of component realizations with respect to their requirements, on another. Both early prevention of problems and efficient conformance testing contribute to the reduction of integration and test effort, in comparison with current industrial practice focussing only on realizations and usually using manual test techniques. Altogether, the current results of the model-based integration method look promising, however further research and application in industry is needed to really illustrate a reduction of integration and test effort in the development of high-tech multidisciplinary systems.

6 Acknowledgements

The authors would like to thank Nicola Trčka, René de Vries, and Will Denissen for their support regarding PROMELA, TORX and test infrastructure. We also thank Jan Tretmans, Dragan Kostić and all other TANGRAM project members for their valuable comments and fruitful discussions.

References

[1] ASML, Website, http://www.asml.com.

[2] Baeten, J. C. M. and W. P. Weijland, “Process Algebra,” Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, 1990.

[3] Boehm, B. W. and V. R. Basili, Software defect reduction top 10 list, IEEE Computer 34 (2001), pp. 135–137.

[4] Bohnenkamp, H. and A. Belinfante, Timed testing with TorX, in: FM 2005: International Symposium of Formal Methods Europe, Newcastle, UK, Lecture Notes in Computer Science 3582 (2005), pp. 173–188.

[5] Bortnik, E. M., N. Trčka, A. J. Wijs, S. P. Luttik, J. M. van de Mortel-Fronczak, J. C. M. Baeten, W. J. Fokkink and J. E. Rooda, Analyzing a χ model of a turntable system using SPIN, CADP and Uppaal, Journal of Logic and Algebraic Programming 65 (2005), pp. 51–104.

[6] Bratthall, L. G., P. Runeson, K. Ädelsward and W. Eriksson, A survey of lead-time challenges in the development and evolution of distributed real-time systems, Information and Software Technology 42 (2000), pp. 947–958.

[7] Brinksma, E. and J. Tretmans, Testing transition systems: An annotated bibliography, in: MOVEP 2000 – Modelling and Verification of Parallel Processes, Lecture Notes in Computer Science 2067 (2001), pp. 187–195.

[8] Broy, M. and O. Slotosch, From requirements to validated embedded systems, in: Embedded Software: First International Workshop, EMSOFT 2001, Tahoe City, CA, USA (2001), pp. 51–65.

[9] Budinsky, F. J., M. A. Finnie, J. M. Vlissides and P. S. Yu, Automatic code generation from design patterns, IBM Systems Journal 35 (1996), pp. 151–171.

[10] de Vries, R. G. and J. Tretmans, On-the-fly conformance testing using Spin, in: Fourth Workshop on Automata Theoretic Verification with the Spin Model Checker, ENST 98 S 002 (1998), pp. 115–128.

[11] de Vries, R. G. and J. Tretmans, Towards formal test purposes, in: Proceedings of the Workshop on Formal Approaches to Testing of Software, FATES ’01, Aalborg, Denmark, BRICS Notes Series NS-01-4, University of Aarhus (2001), pp. 61–76.

[12] Frantzen, L., J. Tretmans and T. A. C. Willemse, Test generation based on symbolic specifications, in: Proceedings of the Workshop on Formal Approaches to Software Testing (FATES 2004), Linz, Austria, Lecture Notes in Computer Science 3395 (2004), pp. 1–15.
[13] Gomaa, H., “Designing Concurrent, Distributed, and Real-Time Applications with UML,” Addison-Wesley Professional, 2000, 1st edition.

[14] Hartman, A., Model-based test generation tools, AGEDIS report, AGEDIS project (2002).

[15] Holzmann, G. J., The model checker SPIN, Software Engineering Journal 23 (1997), pp. 279–295.

[16] Kleppe, A., W. Bast and J. Warmer, “MDA Explained: The Model Driven Architecture: Practice and Promise,” Addison-Wesley Professional, 2003, 1st edition.

[17] Liu, X., J. Liu, J. Eker and E. A. Lee, Heterogeneous modeling and design of control systems, in: Software-Enabled Control: Information Technology for Dynamical Systems, Wiley-IEEE Press, 2003 pp. 105–122.

[18] Ogren, I., On principles for model-based systems engineering, Systems Engineering 3 (2000), pp. 38–49.

[19] Rowson, J. A., Hardware/software co-simulation, in: DAC ’94: Proceedings of the 31st annual conference on Design automation (1994), pp. 439–440.

[20] TANGRAM Project, Website, http://www.esi.nl/tangram.

[21] Tretmans, J. and E. Brinksma, TorX : Automated model based testing, in: First European Conference on Model-Driven Software Engineering, AGEDIS project, 2003.

[22] Trcka, N., Verifying χ models of industrial systems with Spin, Computer Science Reports 05-12, Eindhoven University of Technology (2005).

[23] van Beek, D. A., K. L. Man, M. A. Reniers, J. E. Rooda and R. R. H. Schifflers, Syntax and semantics of timed Chi, Computer Science Reports 05-09, Eindhoven University of Technology (2005).

[24] van Beek, D. A., K. L. Man, M. A. Reniers, J. E. Rooda and R. R. H. Schifflers, Syntax and consistent equation semantics of hybrid Chi, Journal of Logic and Algebraic Programming 68 (2006), pp. 129–210.

[25] van der Bijl, M., A. Rensink and J. Tretmans, Compositional testing with ioco, in: Formal Approaches to Software Testing: 3rd Int. Workshop, FATES 2003, Montreal, Quebec, Canada, Lecture Notes in Computer Science 2931 (2003), pp. 86–100.