Model-based integration and testing in practice

Chapter 7

Model-based integration and testing in practice

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

7.1 Introduction

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

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

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

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

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

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

7.2 Model-based integration and testing method

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

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

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

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

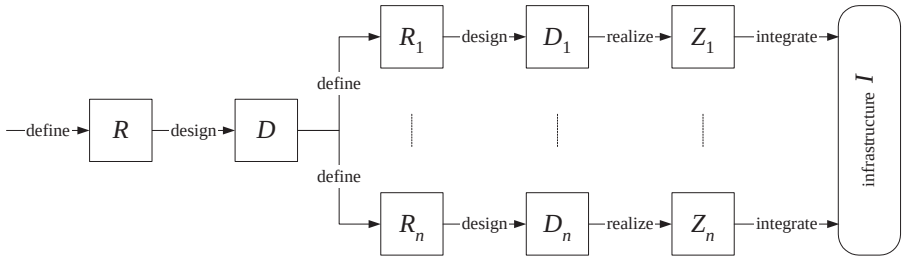

Figure 7.1: Current system development process.

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

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

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

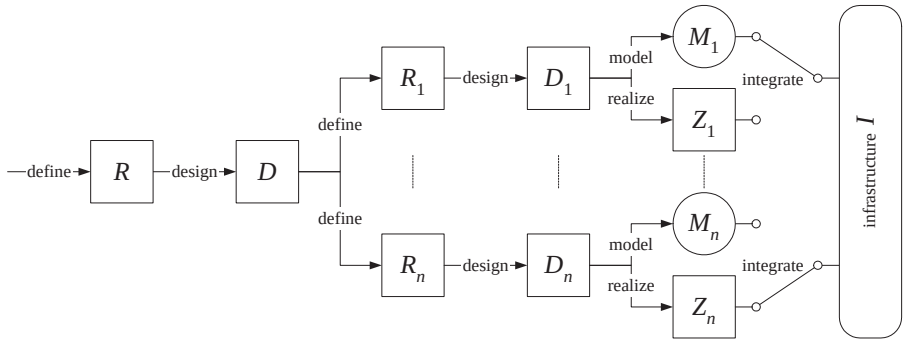

Figure 7.2: System development process in the MBI&T method.

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

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

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

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

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

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

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

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

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

7.3 Case study: ASML EUV machine

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

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

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

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

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

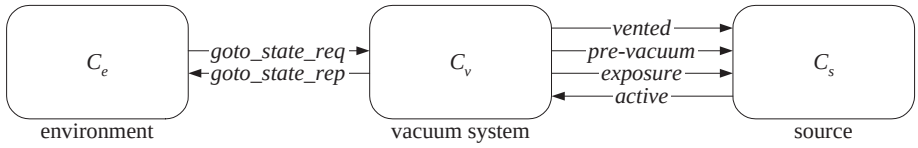

Figure 7.3: Components and interfaces involved in the case study.

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

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

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

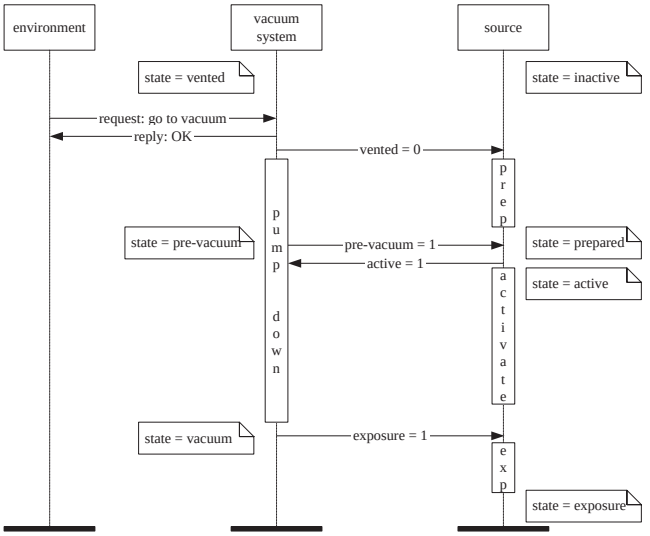

Figure 7.4: Nominal system behavior for the vacuum sequence.

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

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

7.4 Step 1: Modeling the components in χ

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

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

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

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

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

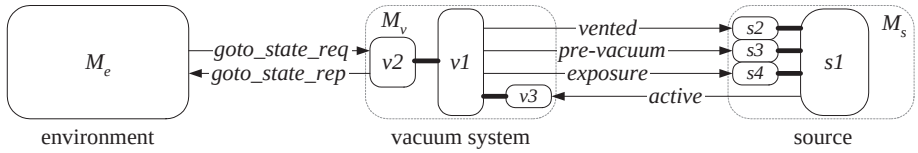

Figure 7.5: Process layout of the χ system model.

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

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

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

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

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

7.5 Step 2a: Simulation of the integrated system model

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

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

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

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

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

7.6 Step 2b: Verification with UPPAAL

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

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

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

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

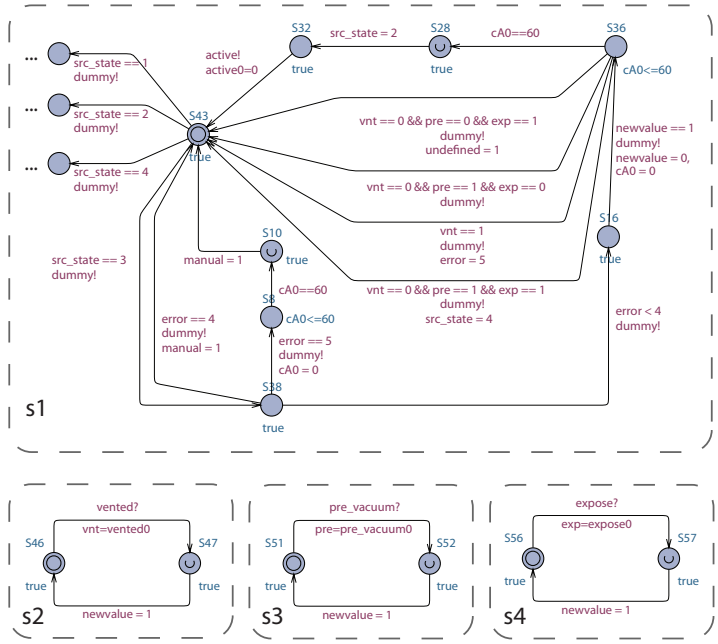

Figure 7.7: Generated UPPAAL automata for the source.

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

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

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

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

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

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

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

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

7.7 Step 3: Model-based integration and system testing

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

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

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

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

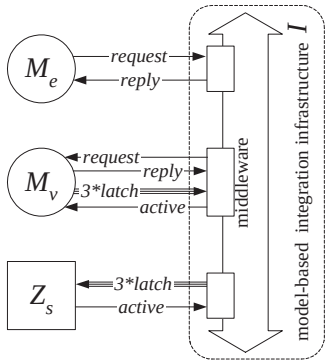

Figure 7.8: Model-based integration infrastructure for the case study.

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

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

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

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

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

7.8 Conclusions

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

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

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

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