Modeling, analysis and implementation of infrastructure for model-based integration and testing

N.C.W.M. Braspenninog, J.M. van de Mortel-Fronczak, J.E. Rooda
Abstract

To reduce the integration and test effort for high-tech multi-disciplinary systems, we use formal and executable models of components for early system analysis and for early integration and testing with available component realizations. In this paper, we investigate the role of the infrastructure that establishes the interaction between the components. In the presented approach, a model of the infrastructure design is included during model-based system analysis. Subsequently, a corresponding model-based integration infrastructure is used to integrate and test combined models and realizations. Applications of this approach to examples of typical interaction types and to a realistic industrial case study prove to be rather straightforward, allowing proper analysis of system and infrastructure properties, which remain valid during model-based integration and system testing.
1 Introduction

High-tech multi-disciplinary systems like wafer scanners, electronic microscopes, medical MRI scanners, and high-speed printers are becoming more complex every day. Growing system complexity also increases the effort (in terms of lead time, costs, resources) needed for the, so-called, integration and testing 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 system integration and testing phases [1]. Furthermore, finding and fixing problems during integration and testing can be up to 100 times more expensive than finding and fixing the problems during the requirements and design phases [2].

Literature reports wealth of research proposing a model-based way of working to counter the increase of development effort, like requirements modeling [3], model-based design [4], model-based code generation [5], hardware-software co-simulation [6], and model-based testing [7]. 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 [8] and OMG’s model-driven architecture [9] (for software systems) are such overall model-based methods, they are mainly focusing on the requirements, design, and implementation phases, rather than on the integration and test phases. Furthermore, literature on industrial applications of such methods is scarce, at least for high-tech multi-disciplinary systems.

Our research within the TANGRAM project [10] focuses on a method of model-based integration and testing (MBI&T), introduced in [11]. In this method, formal 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, which has three main advantages. First, the fact that it is earlier means that the integration and test effort is distributed over a wider time frame. This reduces the effort to be invested during the real integration and testing phases. Second, it allows earlier and thus cheaper detection and prevention of problems that would otherwise occur during real integration. Early problem detection and prevention also reduces the corresponding diagnostic and fix effort and increases the quality of the system at an earlier stage. Third, the use of formal models enables the application of powerful model-based analysis techniques, like simulation and verification. These analysis techniques help to improve the insight in the system’s behavior for the engineers, resulting in better system quality as well.

In our project, we model the components in a process algebraic language [12]. The behavior of a process algebraic model is fully specified by formal semantics. This enables proving properties of the model, e.g. model checking of deadlock, livelock, safety, and other behavioral properties. Communication in the considered process algebraic language is synchronous, i.e. corresponding send and receive actions take place simultaneously. The reason for using a language with synchronous communication is that it allows easier reasoning about the interaction behavior, e.g. at which times interactions take place, because all processes involved need to be ready in order to execute the synchronous communication action. Furthermore, the fact that each process must be ready for interaction at the right times requires the engineer to think more carefully about the system, resulting in a better understanding of the system behavior. Finally, synchronous communication reduces the number of states in the model which improves the capabilities of model checking.

In contrast to models that use synchronous communication, real systems often use asyn-
chronous communication, i.e., send and receive actions do not take place simultaneously. This means that analysis results based on models using synchronous communication, e.g., proven correctness of behavioral properties derived from the system requirements, may become invalid when the models are used for integration with realizations in an asynchronous environment.

The goal of this paper is twofold. The first goal is to investigate an approach for the modeling, analysis, and implementation of component interaction in the MBI&T method, using a modeling language with synchronous communication, such that the infrastructural properties analyzed using the system model remain valid during the integration of models and realizations. The second goal is to show the applicability and the advantages of this approach and of the MBI&T method in general, which is done by giving industrial applications concerning examples of two typical interaction types and a realistic case study. These applications are inspired by the common industrial example for the TANGRAM research project, namely the ASML wafer scanners as used in the lithography industry [13].

Literature provides several approaches that deal with the implementation of synchronous models in an asynchronous environment. However, these approaches cannot be applied in the MBI&T method since the perspective on the goal of modeling is different. In the approaches found in literature, the goal is to automatically transform models into realizations (software components only), and the models need some adaptations before they can be implemented in the asynchronous realization environment. For example, some approaches are based on a restricted subset of the modeling language or put certain constraints on the model such that it can be implemented in an asynchronous environment [14, 15, 16]. This means that also the synchronization between components should be modeled and implemented within the corresponding subset and constraints, for example by adding explicit acknowledgement messages or by using semaphores. Other approaches augment the models with additional communication messages via some protocol in order to negotiate which components will communicate [17, 18, 19]. A common challenge in all these approaches is the correct implementation of the non-deterministic choice operator [20, 21], since this may offer many communication alternatives of which only one may be selected.

In contrast to these approaches the MBI&T method focuses on finding problems in the system as it is designed by the engineers. This means that the models are based on the ‘as is’ designs of the components and the infrastructure (software as well as hardware). When the approaches found in literature would be applied, the models would need to be adapted for asynchronous implementation, e.g., language constructs that are not in the implementable subset would have to be removed or behavior for communication negotiation would have to be added. This means that the models would deviate from the ‘as is’ designs, which does not suit the MBI&T method. Using the ‘as is’ designs as basis for modeling also means that when a non-deterministic choice appears in a component design, it must also be included ‘as is’ in the model. The model can then be used for analysis of potential problems caused by the non-deterministic choice in an asynchronous environment. In our view, solving the problems related to non-deterministic choice is not part of the modeling and asynchronous implementation activities (as in the approaches found in literature), but of the design activities performed by the engineers. Of course, the approaches found in literature can still be applied to the design (and subsequently to the model) in order to solve the problems.

In the MBI&T method, the asynchronous component interaction as specified in the system design and realized in the infrastructure is explicitly expressed in the synchronous modeling language as a separate process. This is a long known approach [22] that is often applied in process algebra [23, 24] and in synchronous programming, for example in the area of so-called globally asynchronous locally synchronous (GALS) systems [25, 26]. By modeling the asynchronous interaction behavior in a synchronous modeling language, we can use the powerful techniques available for synchronous models to analyze the system behavior in an
asynchronous environment. Most of the literature on this approach describes only how the asynchronously interacting systems can be modeled and analyzed using synchronous modeling languages and corresponding simulation and verification techniques. In most cases, the implementation of such models in order to execute them in a real asynchronous environment is not discussed. In this paper, we describe the common modeling and analysis steps, as well as the implementation step, in the context of the MBI&T method. The asynchronous interaction behavior of the components, as it is designed, is included in the synchronous system model for early system analysis. Subsequently, the modeled interaction behavior is implemented in an infrastructure that enables gradual replacement of models by realizations and early system testing of integrated models and realizations. This is done in such a way that the analysis results based on the synchronous system model remain valid during the integration and testing of models and realizations in an asynchronous environment.

Although we do not specifically focus on time behavior in this paper, we can use the same approach for time behavior as for interaction behavior described above. Also in this area, literature provides several approaches to transform models into realizations while preserving the proven time behavior properties [27, 28, 29]. In the MBI&T method, however, we model the time behavior according to the ‘as is’ component designs, as well as the solutions that are included in the designs to prevent potential problems in the time behavior, e.g. a time synchronization protocol. Regardless of whether such solutions are included in the designs or not, we use the models, based on the designs, for early system analysis and system testing to find problems in the (time) behavior of the designed system as early as possible.

The structure of the paper is as follows. The next section describes the MBI&T method, including the techniques and tools used in our research. Section 3 contains an overview of the different forms of infrastructure I and their usage within the MBI&T method. Subsequently, some practical examples of modeling, analysis, and implementation of typical interaction types are given in Section 4. Section 5 describes the application of this approach to an industrial case study. Finally, the conclusions are drawn and discussed in Section 6.

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.

The development process of a system S that consists of n components C_{1..n} (we denote a set \{A_1, \ldots, A_i, \ldots, A_n\} 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 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.

Fig. 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 infras-
structure $I$ that connects the components. For simplicity, the figure shows a ‘sequential’ development process, however in practice the development process will have an incremental and iterative nature. This involves multiple versions of the requirements, designs, and realizations, and feedback loops from certain phases to earlier phases, e.g. from the realization phase back to the design phase.

In the current way of working, only two types of system level analysis 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. $R_i$ versus $R$ and $D_i$ 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 $\{Z_i\}$ can be tested against the system requirements $R$, which requires that the involved 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 become less critical. As a result, the system could be shipped earlier (which is essential in time-driven markets like the lithography industry in which ASML is competing), or more test time could be used to further increase the system quality.

In [11], we introduced the MBI&T method to reduce the integration and test effort. This method takes the design documentation $D_i$ of the components $C_i$ as a starting point and represents them by formal executable models $M_i$, depicted by the circles in Fig. 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 models $M_i$ and realizations $Z_i$, such that all possible combinations of models and realizations can be integrated. As an example, assume that $n = 2$, i.e. the depicted components $C_1$ and $C_2 = C_n$ are the only components of the system. Then Fig. 2 shows, corresponding to the depicted integration ‘switches’, the model-based integrated system $\{M_i, Z_i\}$. In the MBI&T method, the infrastructure can be, for example, parallel composition (to connect component models) or specific software/hardware infrastructure (to connect models and realizations). In this paper, we particularly focus on the modeling, analysis, and implementation of this infrastructure $I$.

When all components are modeled as $M_i$, the integrated system model $\{M_i\}$ is obtained by integrating the component models, for example by parallel composition of process algebraic component models, i.e. $(M_1 \parallel \ldots \parallel M_n)$, where the parallel composition (denoted by $\parallel$) models the infrastructure $I$. 

5 Model-based integration and testing method
For the analysis of the integrated system model \( \{M_1, \ldots, M_n\} \), several model-based techniques can be applied. For instance, simulation can be used to analyze specific behavior scenarios of the system. The simulation results can be compared with the intended system design \( D \) and requirements \( R \).

Due to the complexity of the industrial systems, which often involve both high-level parallelism and non-deterministic behavior, simulation can only show that a system model might have correct behavior. To prove the correctness of system model properties in general, verification can be used. One of the most popular verification techniques is model checking [30], which allows to prove automatically the validity of a given property (derived from the system requirements \( R \) and system design \( D \)), for a given model of a system.

With simulation and verification, it can be determined whether the system model is a good representation of the intended system design and whether it satisfies certain properties, as reported in previous work [31]. If this is the case, and certain component realizations become available, the models of the components can be used for both model-based component testing and for model-based system testing.

Model-based component testing involves automatic testing of the realization of a component, \( Z_i \), against a model of that component, \( M_i \). Using techniques and tools from model-based testing research [7], test cases are automatically generated from the model and automatically executed on the realization. This has also been reported in previous work [11].

While model-based component testing focuses on single components, model-based system testing focuses on integrated models and realizations in order to test their interaction and cooperation, and the resulting system behavior. Here, the available realizations and the models are integrated via an appropriate infrastructure \( I \), which is the main topic of this paper. The resulting model-based integrated system, e.g. \( \{M_1, Z_2\} \) for \( n = 2 \), is tested on system level using test cases derived from the system requirements \( R \) or the system design \( D \). Since this does not require that all component realizations are available, and since models are usually available earlier than realizations, model-based system testing enables earlier detection and prevention of problems when compared to real system testing. Furthermore, models allow easier testing of exceptional behavior, because the model behavior can easily be adapted to create exceptional conditions, for example, a broken component. These advantages of model-based system testing both contribute to a reduction of the effort invested during real integration and testing.

As mentioned previously, this paper focusses on the modeling, analysis, and implementation.
of the infrastructure $I$ in the MBI&T method. The next section gives an overview of the different forms of infrastructure $I$ that are used in the method.

3 Infrastructure in the MBI&T method

The MBI&T method consists of three main activities: modeling the components and their interaction, analysis of the resulting system model, and testing of integrated models and realizations of components. In these activities, the infrastructure is used in three different forms: infrastructure realization, infrastructure model, and model-based integration infrastructure. This section describes these three forms which imply different contents of the rounded rectangle that represents the infrastructure $I$ in Fig. 2. In particular, we focus on the synchronous and asynchronous communication aspects of each form of infrastructure, as described in the introduction of this paper.

3.1 Infrastructure realization $Z_I$

This is the ‘real’ infrastructure that implements the component interaction according to system design $D$, e.g. via cables and communication networks. The example in Fig. 3 shows two component realizations $Z_1$ and $Z_2$ (boxes) and the infrastructure realization $Z_I$ (bold arrows) that enables the communication between the components. For regular electronic and software systems, on which we focus in this paper, the communication in the real infrastructure is asynchronous, i.e. send and receive actions do not take place simultaneously.

![Figure 3: Infrastructure realization $Z_I$](image)

3.2 Infrastructure model $M_I$

In the MBI&T method, the system components are modeled as $M_I$ (see Fig. 2). Besides the components, also the infrastructure that enables component interaction is explicitly modeled and analyzed.

The infrastructure can be modeled on different abstraction levels. During the initial modeling and analysis phases, there may be reasons to use synchronous communication in the model, i.e. completely ignoring the asynchronous behavior. One reason may be that a detailed infrastructure design is unavailable, but system model analysis with a synchronous abstraction of the infrastructure is still helpful. Another reason may be that the infrastructure details are not important for certain model-based analysis activities and would only increase
the complexity and state space when they are included in the model. For example, analyzing the functionality of the system may be possible when only the result of an interaction is known (e.g. a message being transferred), without knowing exactly how that interaction is established.

Although the asynchronous infrastructure behavior may be ignored in the model initially as described above, it must be considered eventually. After all, the models developed in the MBI&T method will eventually be integrated and tested with realizations that do require an asynchronous infrastructure as in $Z_I$. It is important to ensure that the behavioral properties of the analyzed system model are still valid when the component models are integrated and tested with component realizations, as shown in the following counterexample in which the properties do not remain valid. Suppose that the properties of a system model with a synchronous abstraction of the infrastructure are found to be correct during analysis e.g. by using model checking. Subsequently, some component models are replaced by the corresponding realizations, which require an asynchronous infrastructure. The resulting model-based integrated system is then used for testing. Due to the different infrastructural behavior, the models might also interact differently with the other components, possibly resulting in wrong conclusions about the test results. Even worse, when certain safety requirements (regarding machine damage and human safety) are influenced by the infrastructure behavior, the safety, which was analyzed and found to be correct using the models, can not be guaranteed in the realization environment, possibly resulting in hazardous situations.

When infrastructure details are considered during system modeling and analysis, the asynchronous behavior of the infrastructure as designed in $D$ and realized in $Z_I$ must be expressed in the modeling language that is used. For certain interaction types, the modeling language may have constructs to directly express that type of infrastructure. For interaction types that cannot directly be expressed in a modeling language, it may be possible to model their equivalent behavior. For example, the asynchronous communication can be modeled by additional processes in a synchronous language such as the process algebraic language used in the MBI&T method, which is a common and widely used approach [22, 23, 24, 25, 26]. These additional processes are placed between two component processes to model the behavior of that particular component interaction, e.g. a message buffer. Different types of component interaction may require different additional processes in the model, as shown for some examples in the next section of this paper. We denote the modeling constructs used to express the component interaction behavior as the infrastructure model $M_I$. The example in Fig. 4 shows four processes (circles) which, conforming to the process algebraic language, use synchronous communication (arrows). The processes $M_1$ and $M_2$ represent the component models and the processes in between represent the infrastructure model $M_I$, resulting in asynchronous communication behavior between component models $M_1$ and $M_2$.

![Figure 4: Infrastructure model $M_I$](image)
3.3 Model-based integration infrastructure MZ

Besides the infrastructure realization Z_I and the infrastructure model M_I, another form of infrastructure is needed in the MBI&T method. To integrate combinations of models and realizations, a so-called model-based integration infrastructure MZ_I is that implements the component interaction as designed in D and modeled in M_I. Several requirements should be satisfied by MZ_I.

First of all, the communication paradigm of MZ_I should be asynchronous, since the realizations which are integrated by it will also communicate asynchronously. Furthermore, different types of component interaction may require different behavior from the infrastructure. Therefore, these different interaction types should be supported by MZ_I, similar to the different behavior that can be modeled in the infrastructure model M_I. Finally, the model-based integration infrastructure should allow easy integration of models and realizations. This requires that both models and realizations can be connected to the infrastructure with minimal effort. To achieve this, the connection of components to the infrastructure should be independent of the form (model or realization) of the other components and of their exact name, location and interfaces. This makes the integration of components independent of whether models or realizations are used.

The last requirement, independency of connected components, is one of the main features of so-called middleware, which consists of intermediate software that connects software components with each other. Examples of middleware are remote procedure calls (synchronous and asynchronous), object request brokers, and message-oriented middleware (e.g. publish-subscribe) [32]. The components only need to connect and communicate with the middleware and do not depend on the form, name, location, and interfaces of the other components. In the MBI&T method, the model-based integration infrastructure MZ_I is also based on middleware, as described in the next section of this paper.

To connect the components to the middleware, the communication paradigms used by the component models or realizations must be adapted to the communication paradigm of the middleware. This is done by creating ‘connectors’ for the models and realizations such that they communicate via the communication paradigm of the middleware. Different types of components, e.g. software components developed in different languages and tools or hardware components, may require different connectors to be created. We denote the middleware together with the connectors for the models and realizations as model-based integration infrastructure MZ_I. The example in Fig. 5 shows the integration of a model M_I and a realization Z_I using middleware (vertical double headed arrow). Both components are connected to the middleware via connectors (small rectangles) that adapt the communication paradigm of M_I (normal arrows, as in Fig. 4) and the communication paradigm of Z_I (bold arrows, as in Fig. 3) to the middleware and vice versa. The middleware is configured such that the component interaction corresponds to that of Fig. 3 and Fig. 4, i.e. the middleware connects the outgoing communication of M_I to the incoming communication of Z_I and vice versa.

With these three forms of infrastructure, the MBI&T method can be summarized in the following procedure. This procedure takes the component designs D_i and the infrastructure design D_I (part of system design D) as a starting point and consists of three phases.

1. Modeling
   (a) Components M_i based on D_i,
   (b) Infrastructure M_I based on D_I, if available and important for the analysis.

2. Model-based system analysis
Figure 5: Model-based integration infrastructure $MZ_I$

(a) With synchronous abstraction of the infrastructure.
(b) With infrastructure model $M_I$.

3. Model-based system testing
This comprises the replacement of the infrastructure model $M_I$ by the model-based integration infrastructure $MZ_I$, and then for each realized component $Z_i$:

(a) Automatic model-based component testing of $Z_i$ against $M_i$.
(b) Removal of model $M_i$ from the system integrated via $MZ_I$, and integration of realization $Z_i$ as its replacement.
(c) Model-based system testing of the integrated system obtained in step 3b.

In the following sections, we describe different applications of infrastructure modeling, analysis and implementation, inspired by the ASML wafer scanners. Section 4 gives two intuitive examples of the approach applied to interaction types that are typically used in wafer scanners. Subsequently, Section 5 describes how the MBI&T method was applied to a realistic industrial application, particularly focusing on how the infrastructure was modeled (step 1b) and how it was used for model-based system analysis (step 2b) and model-based system testing (steps 3b and 3c). Industrial application of the other steps has been reported in previous work [11, 31].

4 Examples

In our project, we mainly focus on the behavior of concurrent processes that communicate data. This behavior is an important aspect of software and electronic components and strongly relates to the interaction between these components. In general, concurrent behavior is less relevant for mechanical components, and these components themselves are often controlled via electronics and software. Therefore, we concentrate on software and electronic components and their interaction. In the following subsections, we give intuitive examples of the main software and electronic interaction types used in the ASML wafer scanner, namely function calls in software and sequential logic in electronics. For each interaction type, we explain the behavior and properties of the infrastructure realization $Z_i$ and how this behavior can be captured in a synchronous process algebra model $M_i$. The system model with all component models $M_i$ and the infrastructure model $M_I$ is used to analyze behavioral properties of the system and the infrastructure. Subsequently, we show how each interaction type can be implemented in a model-based integration infrastructure $MZ_I$ using middleware.
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 applications presented in this paper, we use the process algebraic language $\chi$ (Chi) [33] to model and analyze components and systems. The $\chi$ language is intended for modeling, simulation, verification, and real-time control of discrete-event, continuous or combined, so-called hybrid, systems, such as manufacturing systems. The $\chi$ toolset [34] allows modeling and simulation of $\chi$ models, as well as their translation to different formalisms, allowing verification of the model using different model checking tools like SPIN, UPPAAL, and $\mu$CRL [35]. A large number of industrial case studies including integrated circuit manufacturing plants, process industry plants, and machine control systems [31, 36, 37] have shown successful applications of simulation and verification using the $\chi$ language and toolset.

To enable model-based integration and system testing in the MBI&T method, the $\chi$ toolset also supports real-time execution of $\chi$ models and integration of $\chi$ component models with other (non-$\chi$) components via middleware [38]. The middleware used as basis for the model-based integration infrastructure $MZ_I$ in the MBI&T method and in the $\chi$ toolset is based on communication via the publish-subscribe paradigm [39], which satisfies all requirements for $MZ_I$ as defined in the previous section. The publish-subscribe paradigm is suitable to decouple the components, since the components do not need to know the exact name, form, location, and interfaces of the other components. The communication paradigm of publish-subscribe is simple. All publish-subscribe messages are related to a so-called ‘topic’, which identifies the contents and type of a message. Components can publish messages of a certain topic to the middleware, and components can subscribe to a certain topic, which means that they will receive all published messages of that topic. Communication via a publish-subscribe middleware is asynchronous since a message is first published to the middleware by a sending component, and then delivered by the middleware to the subscribed components, i.e. the middleware acts as a message buffer. Different types of component interaction, also modeled in different models $MI_i$, can be configured by quality of service (QoS) properties e.g. the number of messages to keep as history, and the reliability of message delivery. Finally, both models and realizations can easily be connected to the publish-subscribe middleware. The connectors for a model of a component must relate all send and receive actions of the model to the corresponding write actions (to publish messages of a topic) and read actions (to receive messages of a topic) of the publish-subscribe middleware. The $\chi$ toolset includes an automatic generator of connectors for a $\chi$ model of a component. The connectors for a component realization depend on the components themselves and may for example involve adapters that translate subscribed messages to function calls and function replies back to published messages, or software-hardware adapters that translate between software messages and electronic signals.

4.1 Function calls (software)

A wafer scanner is controlled by a large amount of software, consisting of more than 12 million lines of code. The main interaction type used in this software system is the function call. A function call consists of an asynchronous request from a client to a server that provides the requested function, followed by waiting for an asynchronous reply from the server with the results of the function. The ‘wait for reply’ action can possibly contain a time-out that is triggered when the reply is not received within a specified amount of time. In practice, these time-outs are used to detect problems in the communication between client and server or in the function execution by the server.

There are two different types of function calls, blocking and non-blocking. In a blocking function call, no other statements may be executed between the request and the reply, while this is allowed in a non-blocking function call. Furthermore, blocking function calls do not
use time-outs. Since a blocking function call is a special case of the non-blocking function call (with no statements between request and reply and no time-out), we only discuss the more generic non-blocking function call here.

Important properties of function calls as used at ASML are:

- **Lossless communication**
- **FIFO order of requests and replies between client and server and vice versa**
- **Limited number of messages in asynchronous communication buffer**
- **Consistency:** the number of requests is equal to the number of replies or at most one larger (during function execution)
- **Wait/time-out:** a time-out may only be triggered when the reply buffer is empty for the specified amount of time since the start of the ‘wait for reply’ action.

Note that using the time-out as a detection mechanism for communication problems could be captured in a property ‘time-outs may never occur’, however this property is not related to infrastructure but to required system behavior.

Function calls with asynchronous communication can easily be modeled in a synchronous modeling language such as $\chi$ by including a ‘buffer’ process between two communicating processes. The $\chi$ code of a buffer process $B$ is shown in Fig. 6, with two communication channels, input $a$ and output $b$, for messages of the $msg$ topic. The process shows repetitive behavior (denoted by $\ast$, line 3), starting with guarded expressions (denoted by $\rightarrow$, lines 3-4) to check for buffer overflow, i.e. whether the length of message list $xs$ exceeds the configured buffer size $n$. If this is not the case, the process continues its behavior (denoted by skip, line 4). The buffer overflow check is followed (denoted by the sequential composition operator $;$, line 6) by two alternatives (denoted by $\parallel$, line 7) of which the one that is enabled first will be selected. Either a new message $x$ is received via channel $a$, which is then appended to $xs$ (line 6), or, if $xs$ is not empty, the head (first item) of $xs$ is sent via channel $b$, after which the tail (all but first item) of $xs$ remains (line 7).

```
proc B(chan a : b ! : msg, val n : nat) =
| var xs : [msg] = []
| x : msg
| a ? x : (len(xs) > n → !"buffer overflow"
| ∥ len(xs) ≤ n → skip
| 5 )
| 6 )
| 7 )
| 8 )
| 9 )
| 10 )
```

**Figure 6: Buffer process B**

Using multiple instantiations of buffer process $B$, we can model a function call as used at ASML as shown in Fig. 7. For simplicity, the declaration and initialization part for the channels and variables is omitted and only the body of the $\chi$ model is shown, namely four processes in parallel composition (denoted by $\parallel$). The first process (lines 1-6) is a partial specification (denoted by $\ldots$) of a client that calls some function $f$. This function call is modeled as a sequential composition (denoted by $;$) of sending an asynchronous request with the function arguments ($f_{\text{req!arg}}$, line 2) and receiving an asynchronous reply of the function with

```
1 | { \ldots
2 | \ f_{\text{req!arg}}
3 | \ f_{\text{req?arg}} ∥ \ ∆t → !"time-out" \}
4 | \ f_{\text{rep?rep}} \ f\ (\ f_{\text{req!arg}} ∥ \ f_{\text{req?arg}} ∥ \ f_{\text{rep?rep}} f_{\text{rep?arg}} ) \}
5 | \ f_{\text{req?arg}} ∥ \ f_{\text{req!arg}} ∥ \ f_{\text{rep?rep}} f_{\text{rep?arg}} \}
6 | \ f_{\text{req?arg}} ∥ \ f_{\text{req!arg}} ∥ \ f_{\text{rep?rep}} f_{\text{rep?arg}} \}
7 | \ f_{\text{req?arg}} ∥ \ f_{\text{req!arg}} ∥ \ f_{\text{rep?rep}} f_{\text{rep?arg}} \}
8 | \ f_{\text{req?arg}} ∥ \ f_{\text{req!arg}} ∥ \ f_{\text{rep?rep}} f_{\text{rep?arg}} \}
9 | \ f_{\text{req?arg}} ∥ \ f_{\text{req!arg}} ∥ \ f_{\text{rep?rep}} f_{\text{rep?arg}} \}
10 | \ f_{\text{req?arg}} ∥ \ f_{\text{req!arg}} ∥ \ f_{\text{rep?rep}} f_{\text{rep?arg}} \}
```

**Figure 7: $M_f$ for function call**
the results \((f_{\text{rep}} ? \text{rep}, \text{line 4})\). Between these two statements, other actions (denoted by \ldots\), \text{line 3}) may be performed (not for blocking function calls). The possible time-out on the ‘wait for reply’ action is modeled as an alternative composition of the receive action and a delay of \(t\) time units (denoted by \(\Delta t\), \text{line 4}), which means that either the reply is received or the delay is finished, resulting in a time-out. Note that \(t\) is infinity (no time-out) for blocking function calls. The second process (\text{line 7}) models the server, which repetitively waits for requests for the only function it provides, function \(f\) (more provided functions can be added in a similar way). Upon receiving a function call request from a client with certain arguments \(\text{arg}\), the result of the function executed on \(\text{arg}\) is sent back as a reply. Finally, two buffer processes \(B\) (lines 8 and 9) are used to model the asynchronous communication. The buffer processes are connected to the request and reply channels of the client and server, similar to Fig. 4. The buffer sizes are set to one since a client process may only call one function at a time. To simplify the example, we assume that a function is required by only one client. More complex clients and servers with different properties can be modeled in a similar way, e.g. clients that make multiple subsequent calls, servers that allow multiple calls to be ‘pending’, or servers that provide functions required by multiple clients.

Using this infrastructure model \(M_I\) for function calls, we can include the infrastructural properties mentioned earlier in this section during system model analysis. In this paper, we only make a reasonable case for the correctness of these properties by informal explanation, formal proofs are subject of further research. In this paper, we assume that all communication is lossless. Due to the use of lists and their head and tail functionality in the buffer processes, it is not possible for two messages to overtake each other in the buffer, so FIFO behavior is guaranteed. The validity of the limited buffer property depends on the behavior of all components, and can be checked by performing a reachability analysis of the buffer overflow state of all buffer processes, e.g. by using a model checker as in [31]. In the model of the server, each incoming request is immediately followed by sending the reply, so the number of requests is always equal to or at most one larger than the number of replies. For more complex server models, e.g. with functions required by multiple clients, request and reply counters can be added to enable model checking of the property: \(0 \leq \text{nr_requests} - \text{nr_replies} \leq 1\). The wait/time-out property is covered in the infrastructure model \(M_I\) of Fig. 7, because the communication in the \(\chi\) model is urgent, meaning that a process may not delay if a communication action is enabled. This implies that the time-out (\(\Delta t\)) can only be triggered when after \(t\) time units the receive action \((f_{\text{rep}} ? \text{rep})\) has not been enabled. Besides these already listed properties, two properties of blocking function calls, namely subsequent requests and replies (no intermediate statements) and infinite time-outs, can be checked by static analysis of the model structure (e.g. by a compiler).

When integrating models and realizations of components that use function calls to interact, the model-based integration infrastructure \(MZ_I\) can easily be implemented in the publish-subscribe middleware. Since the middleware uses asynchronous communication and acts as a message buffer itself, it is well suited as implementation of the buffer processes \(B\) from \(M_I\) and of the real buffers used in the real function calls in \(Z_I\). Fig. 8 shows the implementation of \(MZ_I\) for the example of Fig. 7 with a model of client \(M_C\), a realization of server \(Z_S\), and the topics for requests and replies. The client is configured as publisher of requests for function \(f\) (topic \(f_{\text{req}}\)), and it is subscribed to replies of \(f\) (topic \(f_{\text{rep}}\)). The server is subscribed to function call requests for its provided function \(f\), and publishes the corresponding replies.

With this component configuration, the translation from \(M_I\) to \(MZ_I\) is simple, namely all send and receive actions in the client and server models are replaced by write and read actions to the corresponding topics on the publish-subscribe middleware. This is accomplished in the generated connectors for the \(\chi\) models, which are able to determine possible next steps (transitions) of a component model, select one of them, and execute them using the \(\chi\) simulator. Possible steps that can be executed include internal behavior, temporal behavior, and communication behavior of the model. Steps related to the internal behavior, e.g. assign-
ments, choices between alternatives, calculations, are directly executed according to the \( \chi \) semantics implemented in the \( \chi \) simulator. The execution of steps related to the temporal behavior, e.g. real-time delays and time-outs, is implemented in the connector, using features from the event-driven networking framework in which the publish-subscribe middleware is implemented as well [40]. For a time-out, the connector checks whether the read action corresponding to a ‘wait for reply’ action in the model can be executed within the specified amount of time. Otherwise a time-out is triggered and the corresponding steps are chosen and executed. Finally, the execution of steps related to communication behavior, i.e. the send and receive actions, is implemented using the publish-subscribe middleware in the following way. When a transition in the model is selected that involves a send action, e.g. \( a! \text{true} \), then the connector executes this send action by publishing the value (\text{true}) of the corresponding topic (\( a \), the name of the channel) to the middleware, and by updating the state of the model to the state that follows the transition. When a possible transition in the model involves a receive action, e.g. \( b?x \), then the connector determines whether a new value of the corresponding topic (\( b \), the name of the channel) is available to be received by the model. Since the connector is subscribed to all topics that correspond to incoming channels of a component model, it will receive all published values of those topics. The received values are stored in a queue within the connector. For the function call interaction type, the queue behavior is FIFO. As we will see in the next example, the queue behavior can be influenced via the QoS properties of the publish-subscribe middleware. With the FIFO queues in the connectors, a possible receive action \( b?x \) can only be selected if there is at least one message in the queue for the corresponding topic \( b \). If the receive action is selected, the value of the oldest message in the queue is assigned to the variable (\( x \)) and the message is removed from the queue. In this way, the model-based integration infrastructure \( MZ_I \) behaves in a similar way as the infrastructure model \( M_I \) and the earlier defined properties, which were analyzed and found to be correct for \( M_I \), remain valid in the realization environment. As mentioned previously in this paper, we only make a reasonable case for the correctness of these properties by informal explanation, formal proofs are subject of further research.

For the integration of a client or server realization, the connector should translate between publish-subscribe messages and real function call requests and replies, including the transformation or ‘marshalling’ of the arguments. For example, when the connector of server realization \( Z_S \) receives a request of the topic \( f_{\text{req}} \), it should call the real function \( f \) of \( Z_S \), after which the result is published on the middleware with \( f_{\text{rep}} \) as topic. For a realization, the time-out functionality is included in the component realization itself.
4.2 Sequential logic (electronics)

Many interaction types for electronic components are based on sequential logic, which depends not only on the current state, but also on the previous state. It is typically used to create memory in which values are stored as voltages in the circuits. Latches and flip-flops are well-known sequential circuits that appear in many forms for direct communication between electronic components (e.g. via cables) or for communication between software and electronics (e.g. via memory mapped I/O or distributed I/O). In all these forms of sequential logic, the sending component is able to set a certain value that is stored in the circuit, and the receiving component is able to observe or read this value. Taking the set/reset or SR-latch as a simple example, a sending component can set the SR-latch to active or reset it to inactive (i.e. high or low voltage). In most cases, the state of an SR-latch relates to some internal state of the sending component, e.g. ‘standby’, ‘ready for next action’, or ‘error’. Via the SR-latch, the receiving components can observe this internal state at all times.

Some typical sequential logic properties, with the SR-latch as example:

- The output value of an SR-latch is continuous (active/inactive) and can only be changed by a set or reset input from the sending component.
- A set or reset input results in an active or inactive latch output, respectively.

Although the SR-latch contains both discrete-event and continuous behavior, which could directly be modeled in hybrid $\chi$ [33], we restrict ourselves to the discrete-event version of $\chi$, in which we abstract from the continuous behavior of the SR-latch. A discrete-event model of the SR-latch is shown in Fig. 9, in which the declaration and initialization part for the channels and variables are omitted for simplicity. The highest level of parallel composition (|| on line 5) contains the processes of the sending component (lines 1-4) and the receiving component (lines 5-9) of the ready_latch, which indicates whether the sending component is ready for some next action. The sending process first sets the latch output to false (line 1) and later, when it is ready, to true (line 3). The receiving process checks whether the other component is ready via the latch value $\text{ready}$ (lines 5-7). If this is the case, it continues its behavior (skip); otherwise it raises an error. The discrete-event abstraction of latch communication is modeled by adding another parallel process (line 8) to the model of the receiving component, i.e. on the second level of parallel composition (|| on line 8). This additional process of the receiving component repetitively waits for new values of the ready_latch from the sending component. The variable $\text{ready}$ is used to store the latest latch value and to share it with the other processes in the parallel composition of the receiving component. In this way, only the latest latch value is considered in the behavior of the receiving component.

![Figure 9: $M_1$ for SR-latch](image-url)
The properties given for the SR-latch are satisfied by the model since the ready variable always has a value (mimicking continuous behavior) and can only be set to true (active) or reset to false (inactive) by the sending component.

For the SR-latch, the publish-subscribe middleware for the model-based integration infrastructure $MZ_I$ is configured with different QoS properties than for the function call interaction type. For function calls, the queues in the connectors act as multi-message FIFO buffers from which the messages are removed after delivery to the receiving component. However, for the SR-latch, the queues should store and keep only the last value that is received from the sending component. This is achieved by configuring the publish-subscribe middleware with the QoS property ‘keep one message as history’. This QoS property changes the behavior of the connector queue in such a way that only one message is stored in the queue, and that the message is not removed from the queue after delivery to the receiving component. Besides this difference in queue behavior, the model connectors for the SR-latch interaction type are generated in the same way as for the function call interaction type. For a component realization that uses the SR-latch or another sequential logic interaction type (implemented in electronics), the connector should adapt from publish-subscribe messages to software and electronic signals and vice versa. An example of such a realization connector is given in the industrial case study described in the next section.

In the described SR-latch example, only one value is stored (single-address memory). The infrastructure model $M_I$ and its implementation $MZ_I$ can easily be extended to represent multi-address memories as used in memory mapped I/O and distributed I/O.

5 Industrial case study

The described approach on how to deal with infrastructure in the MBI&T method was applied to an industrial case study in order to show proof of concept and to evaluate the MBI&T method. In the case study, all steps except step 3a of the MBI&T procedure described in Section 3 were performed. Component modeling (step 1a) and model-based system analysis by means of simulation and verification (step 2) have been reported in [31]. Although model-based component testing (step 3a) was not performed, a similar industrial application has been reported in [11]. In these previous papers, the focus was on the functional behavior of the components and the system. In this paper, however, we do not go into detail about the functional behavior; we now focus on the interaction behavior of the system and on the modeling (step 1b), analysis (step 2b), and implementation (steps 3b and 3c) of the infrastructure.

The case study involves a part of an ASML wafer scanner. In an ASML wafer scanner, laser light transfers the image of a lithographic pattern onto the surface of a silicon wafer with nanometer accuracy. The laser light passes through an optical system that scales down the pattern image before it is projected onto the wafer. Currently, a new type of wafer scanner is under development within ASML, which uses extreme ultra violet (EUV) light 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 absorbed by nearly all materials, including air.

The main aspect considered in the case study is 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.

Fig. 10 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 (SR-type latches, as used in Subsection 4.2 as example), 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 current vacuum system state can be observed via the vacuum system user interface. For the system design considered in the case study, it was sufficient to allow only observation but no notification of the vacuum system state to the environment.

![Diagram of components and interfaces](image)

**Figure 10:** Components and interfaces involved in the case study

The next subsections describe how the steps of the MBI&T procedure were applied in the case study. As previously mentioned, this paper concentrates on the interaction behavior of the components and on the modeling (step 1b), analysis (step 2b), and implementation (step 3b and 3c) of the infrastructure. Where necessary, we give a summary of the other steps, which have been described in more detail in [31].

### 5.1 Steps 1a and 1b: Modeling the components and their interaction

In step 1a of the MBI&T method, the components shown in Fig. 10 were modeled as $\chi$ processes. The environment is modeled as a single sequential process $M_e$ which can request the vacuum system to go to a certain state at certain points in time. In this way, different scenarios for analysis and testing of the system can be configured in $M_e$. The vacuum system $M_v$ is modeled as a parallel composition of the processes ($v_1 \parallel v_2 \parallel v_3$): the core process $v_1$
models the internal state behavior, while the processes \( v_2 \) and \( v_3 \) model the interaction with \( M_v \) and \( M_s \), respectively. The source \( M_s \) is modeled as a parallel composition of the processes \((s_1 \parallel s_2 \parallel s_3 \parallel s_4)\): the core process \( s_1 \) models the internal state behavior and the processes \( s_2, s_3, \) and \( s_4 \) model the interaction with \( M_v \) via the latches. The integrated system \( \{M_e, M_v, M_s\} \) is modeled in \( \chi \) as the parallel composition of the component processes: \( (M_e \parallel M_v \parallel M_s) \). The parallel composition operator \( \parallel \) synchronizes the components on time and on communication actions.

Fig. 11 shows how the \( \chi \) processes of the system model are mapped onto the system design shown in Fig. 10. The arrows depict the channels that model the communication between processes of different components, which is a direct mapping of the interfaces in Fig. 10. 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 these processes. Here, we only describe how the interaction between the components was modeled as infrastructure model \( M_I \) (step 1b), more details on the functional behavior of the system model can be found in [31].

The interaction between environment \( M_e \) and vacuum system \( M_v \) is based on the non-blocking function call interaction type as explained in Subsection 4.1. The infrastructure model \( M_I \) for this interaction behavior is shown in Fig. 12, very similar to Fig. 7, with four processes in parallel composition. The first process is \( M_e \) (lines 1-7), which is the client that makes a non-blocking function call via goto_state_req using the requested state (\( \text{state}_{\text{req}} \)) as argument (line 2). After that, \( M_e \) waits for the reply \( \text{rep} \) via goto_state_rep with a time-out to detect communication problems between \( M_e \) and \( M_v \) (lines 3-5). The function goto_state is provided by the second process (lines 8-11), which is process \( v_2 \) of \( M_v \) (i.e. the server). After successfully receiving a function call request (line 8), \( v_2 \) uses the requested state argument to set the corresponding values of the variables shared with the core process \( v_1 \) (line 9, actual variable assignments are omitted for simplicity), followed by sending back a reply ‘OK’ to \( M_e \) (line 10). Based on the shared variable values, \( v_1 \) performs the actions necessary to get to the requested state. Finally, the asynchronous communication is modeled by placing two buffer processes \( B \) (lines 12 and 13) between the client and server.

The interaction between vacuum system \( M_v \) and source \( M_s \) is based on the SR-latch interaction type as explained in Subsection 4.2. The infrastructure model \( M_I \) for the SR-latches in the case study is similar to the model shown in Fig. 9. For each of the four SR-latches, a process like the one on line 8 of Fig. 9 is added to the model of the receiving component of that latch, namely processes \( s_2, s_3, \) and \( s_4 \) for the ‘vented’, ‘pre-vacuum’, and ‘exposure’ latches from \( M_v \) to \( M_s \), and process \( v_3 \) for the ‘active’ latch from \( M_s \) to \( M_v \). Taking the three SR-latches from \( M_v \) to \( M_s \) (Fig. 13) as an example, the source model \( M_s \) with its core behavior specified in process \( s_1 \) (lines 1-4, actual behavior is omitted for simplicity) is extended with processes \( s_2, s_3, \) and \( s_4 \) (lines 5, 6, and 7, respectively). These additional processes repetitively wait for new values of the ‘vented’, ‘pre-vacuum’, and ‘exposure’ latches, which are stored in corresponding variables \( (vnt, pre, exp) \) that are shared with the other processes of the receiv-
ing component. Based on the values of these shared variables and on the current source state, the core process \( v \) decides which behavior it should perform.

\[
\begin{align*}
\text{1) } & \{ \ldots \} \\
\text{2) } & \text{\textbf{go to state} req ! state req} \\
\text{3) } & \text{\{ goto state rep ? rep} \\
\text{4) } & \text{\{ }\text{.dll \( \Delta \) \( \rightarrow \) \"communication time-out\\} } \\
\text{5) } & \text{\}} \\
\text{6) } & \text{\ldots} \\
\text{7) } & \text{\ldots} \\
\text{8) } & \text{\ldots} \\
\text{9) } & \text{\ldots set variables shared with \( v \ldots \)} \\
\text{10) } & \text{\{ goto state rep ! \"OK\"} \\
\text{11) } & \text{\}} \\
\text{12) } & \text{\{ }\text{\textbf{goto state req} goto state req, }1\} \\
\text{13) } & \text{\{ }\text{\textbf{goto state req} goto state rep, }1\} \\
\end{align*}
\]

Figure 12: \( M_1 \) for \text{\textbf{goto state} function}

\[
\begin{align*}
\text{1) } & \{ \ldots \} \\
\text{2) } & \text{\ldots core behavior based on \( \text{\textbf{vnt, pre, exp}} \ldots \)} \\
\text{3) } & \text{\ldots} \\
\text{4) } & \text{\ldots} \\
\text{5) } & \text{\ldots} \\
\text{6) } & \text{\ldots} \\
\text{7) } & \text{\ldots} \\
\end{align*}
\]

Figure 13: \( M_1 \) for SR-latches from \( M_v \) to \( M_s \)

### 5.2 Step 2b: Model-based system analysis

As reported in [31], the functional behavior of the \( \chi \) system model was analyzed by simulation of both nominal and non-nominal scenarios and by UPPAAL model checking of formal properties derived from the system requirements \( R \) and the system design \( D \). During this model-based system analysis, several design and integration problems were detected and repaired at an early stage of the development process. In this paper, we only discuss the analysis of the infrastructural properties of the system model.

As described in Subsections 4.1 and 4.2, most of the function call and SR-latch properties are directly satisfied by the infrastructure model, independent of the system behavior. However, two function call properties depend on the system behavior and should be verified in the context of the system model. These properties, limited buffer size and consistency between the number of requests and replies, were verified in the case study in the following way. To detect buffer overflows, a boolean variable \( \text{overflow} \) was added to the model which becomes true whenever a buffer overflow is detected (line 3 of Fig. 6). The limited buffer size property was verified by model checking the UPPAAL query \( A[] \text{not overflow} (A[]) \) stands for ‘always’), which is only valid when \( \text{overflow} \) is false in all possible states of the model. To verify consistency, i.e. the number of requests must be equal to or at most one larger than the number of replies, two counting variables, \( nr\_requests \) and \( nr\_replies \), were added to the model and the UPPAAL query \( A[] 0 \leq nr\_requests \cdot nr\_replies \leq 1 \) was used for model-checking. These two infrastructural properties of the function call interaction type were both satisfied for the system model developed in step 1 of the case study. Successful analysis of the integrated system model in step 2 gives enough confidence that the model is a good representation of the system design, making it suitable for model-based integration and system testing in step 3.

### 5.3 Steps 3b and 3c: Model-based system testing

In these steps 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 by a model-based integration infrastructure \( MZ_I \). In the infrastructure realization \( Z_I \), the latch communication is established
via a multi-pin cable, of which four pins relate to the four latches ‘vented’, ‘pre-vacuum’, ‘exposure’, and ‘active’. In order to integrate \( M_e \) and \( Z_s \), they must be able to communicate with each other via this multi-pin cable. This is achieved by using the model-based integration infrastructure \( MZ_I \) as proposed in Subsection 3.3, together with appropriate connectors for \( M_v \) and \( Z_s \). Also the environment model \( M_e \) is integrated via \( MZ_I \), implementing the function call interaction type for the goto_state function between \( M_e \) and \( M_v \).

Similar to the examples in Section 4, we use the publish-subscribe communication paradigm as a basis for the model-based integration infrastructure \( MZ_I \). The interaction between \( M_e \) and \( M_v \) via the goto_state function is implemented in \( MZ_I \) by defining the topics request and reply and by configuring the published and subscribed topics in \( M_e \) and \( M_v \) accordingly, in a similar way as shown in Fig. 8. For the interaction between \( M_e \) and \( Z_s \), a topic is defined for each of the four latches. This results in the configuration for \( MZ_I \) as shown in Fig. 14, in which the arrows for the three SR-latches from \( M_v \) to \( Z_s \) (‘vented’, ‘pre-vacuum’, and ‘exposure’) are combined, denoted by 3*latch.

Using the \( \chi \) toolset, the middleware configuration described above and the connectors for the models are automatically generated from the \( \chi \) system model. The behavior of the message queues in the connectors is configured using the QoS properties corresponding to the interaction types, as described in the examples in Subsections 4.1 and 4.2.

Although the model connectors can automatically be generated from the \( \chi \) system model, the realization connectors are case specific and should be created manually (or selected from a library with generic and configurable connector templates, if available). In the case study, the connector for \( Z_s \) should adapt the real latch communication via the multi-pin cable to the publish-subscribe communication used in the middleware and vice versa. To achieve this, a SW/HW adapter is used in the form of a remote I/O unit, e.g. from Opto 22 [41] or National Instruments [42]. A remote I/O unit consists of a central processing unit and a configurable set of modules that allow different forms of analog and digital input and output. The central processing unit is controlled by giving serial commands from a common PC, which allows to get and set input or output values in the modules, for example getting the value from an analog input module or setting the value of a digital output module. 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 $MZ_I$ as shown in Fig. 14, we were able to integrate and test the environment model $M_e$, the vacuum system model $M_v$, and the source realization $Z_s$ significantly earlier (20 weeks before all realizations were available and integrated) and cheaper (no critical cleanroom time as for real system testing). Similar to the simulation analysis in step 2 of the case study, the environment 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, while 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 during the model-based system tests. 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 cleanroom. The models supported immediate diagnosis (i.e. determining the cause of a problem) and repairing of the detected implementation errors, as well as immediate retesting of the repaired system. This means that the model-based integration and testing activities potentially saved several days of expensive cleanroom time during real integration and testing 20 weeks later (if the problems would remain undetected until that time). In the period after the model-based system tests, no additional problems in the source realization were found (and no expensive fixing was necessary during real system testing), at least not for the aspects that were analyzed and tested using 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, respectively. 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 abstract from behavior that is not important for the tests and thus from the minor problems that potentially prohibit test execution.

Third, the use of models for testing reduces the time spent on diagnosis of problems when compared to real system testing. On the one hand, the number of sources that could potentially cause a problem is reduced since the models abstract from behavior that is not important for the tests, i.e. all other components and aspects which form potential problem sources in real system testing are not included in the model. On the other hand, the complete insight in and control over the models makes the distinction between the potential problem sources more clear.
6 Conclusions

The goal of this paper was twofold. The first goal was to investigate an approach for the modeling, analysis, and implementation of component interaction in the MBI&T method, using a modeling language with synchronous communication, such that the infrastructural properties analyzed using the system model remain valid during the integration of models and realizations. In the presented approach, the behavior of the (asynchronous) infrastructure realization $Z_I$, based on the system design $D$, is modeled as infrastructure model $M_I$ using synchronous communication, for example in the form of message buffers between components. This infrastructure model $M_I$ is included in the system model in order to analyze properties related to infrastructural and system behavior, for example by using simulation and model-checking techniques. Subsequently, the infrastructure as designed in $D$ and modeled in $M_I$ is implemented in a model-based integration infrastructure $MZ_I$, using a publish-subscribe middleware, allowing easy integration and testing of combined models and realizations.

The second goal of the paper was to show the applicability and the advantages of this approach and of the MBI&T method in general, by giving industrial applications concerning examples of two typical interaction types and a realistic case study. In the industrial applications, the transition from synchronous process algebraic models to distributed asynchronous realizations proved to be rather straightforward. The considered asynchronous interaction types (function calls and SR-latches) can easily be modeled in a synchronous modeling language like $\chi$, using additional processes in the system model. These synchronous system models provide a good understanding of system behavior and enable verification of properties related to both infrastructural and system behavior. The publish-subscribe middleware and the connectors provide a simple means to implement the modeled interaction behavior, allowing easy integration of models and realizations. By using different QoS properties of the middleware, the behavior of the queues in the connectors can be influenced in accordance with the required behavior, e.g. the ‘keep one message as history’ QoS property for the SR-latch interaction type. For the examples, we gave an informal and intuitive indication of the correctness of the properties for both $M_I$ and $MZ_I$, leaving formal proofs as future work. The described approach can be applied to other interaction types in a similar way.

In the industrial case study, the modeling and analysis activities in the MBI&T method, as previously reported in [31], show relevant advantages for the system development process. The modeling activities helped to clarify, correct, and complete the design documentation. By simulation and verification, a number of design and integration problems was detected and fixed earlier and cheaper when compared to current system development. Finally, the integration of models and realizations using the model-based integration infrastructure enabled earlier, faster and cheaper system integration and testing, several months before real integration and testing. During the tests, multiple implementation errors in the realization were detected and repaired, saving significant amounts of time and rework during the real integration and test phases. This clearly shows the applicability and the advantages of the MBI&T method for industrial system development. However, 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 from the engineers.

In contrast to the potential effort reduction (in terms of lead time, costs, or resources), the effort required to develop the models for the MBI&T method should also be considered when deciding where and when to apply the method. As shown in [43], integration and test sequencing techniques can be used for this necessary trade-off analysis between required effort and potential benefits of using models for integration and testing. By applying models at places where it is possible and most profitable, the integration and testing process can be improved in terms of time to market, quality, and costs.
In principle, the MBI&T method could support system development in any industry that has separate development processes for the (multi-disciplinary) components, and for which it is difficult to perform analysis on the system level without having a realized and integrated system available. However, to be able to apply the MBI&T method to the development of a particular system, the method needs to be instantiated with paradigms, mathematical techniques, and tools that are appropriate to model, analyze, and test the system views that are important for the particular system and development process.

Acknowledgements

This work has been carried out as part of the TANGRAM project under the responsibility of the Embedded Systems Institute, partially supported by the Netherlands Ministry of Economic Affairs under grant TSIT2026. The authors would like to thank Ton Geubbelis, Johan Neerhof, Ralph Meijer, Albert Hofkamp, Jan Tretmans, and Wan Fokkink for their support and comments.


26 Bibliography
[34] Systems Engineering Group, Mechanical Engineering Department, Eindhoven University of Technology, Chi language and tools website, http://se.wtb.tue.nl/sewiki/chi.


