conference on systems engineering research
Paper #109

Analysis and implementation of infrastructure for model-based integration and testing*

N.C.W.M. Braspenning, J.M. van de Mortel-Fronczak, J.E. Rooda**
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, the Netherlands
* This work has been carried out as part of the TANGRAM project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under grant TSIT2026.
** Corresponding author: n.c.w.m.braspenning@tue.nl

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. We include a model of the infrastructure for model-based system analysis and we implement a corresponding integration infrastructure to integrate and test models and realizations. Application of this approach to examples of typical interaction types proves to be rather straightforward, allowing proper analysis of system and infrastructure properties, which remain valid during model-based integration and system testing.

Introduction

To reduce the ever increasing integration and test effort in high-tech multi-disciplinary system development, we propose a *model-based integration and testing* (MBI&T) method, introduced in (Braspenning et al. 2006a). The method is illustrated in Figure 1, showing the system development process that starts with requirements R and design D of the system. Subsequently, the requirements R_i, the designs D_i, the models M_i, and the realizations Z_i of all n components of the system are developed. The components, either represented by formal and executable models M_i or by realizations Z_i, 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).

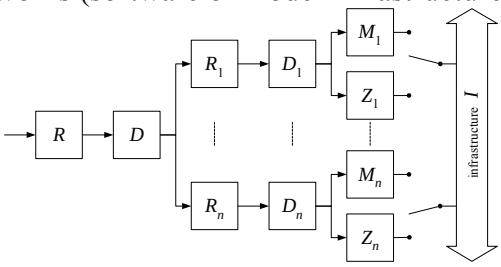

Figure 1. System development process in the MBI&T method

To detect and prevent integration problems at an early and therefore less expensive (Boehm and Basili 2001) stage of the development process, several model-based analysis and testing techniques can be applied in the MBI&T method. For example, simulation and model checking can be used to

CSER 2007, Stevens Institute of Technology, ISBN - 0-9787122-1-8
PROCEEDINGS CSER 2007, March 14-16, Hoboken, NJ , USA

validate and verify the behavior of the system model (i.e. with models only), and testing can be used to find errors in a partly realized system (i.e. with models and realizations).

In our project, we model the components in a process algebraic language (Baeten and Weijland 1990). The behavior of a process algebraic model is fully specified by formal semantics. This enables proving the correctness of a model, e.g. model checking of deadlock, livelock, safety, and other behavioral properties. Communication in the process algebraic language is synchronous, i.e. corresponding send and receive actions take place simultaneously. Synchronous communication reduces the complexity of the model, resulting in a better understanding of the system behavior. Furthermore, it reduces the number of states in the model which improves the capabilities of model checking.

In this paper, we investigate the modeling, analysis, and implementation of component interaction via infrastructure *I*. Although the models use synchronous communication, real systems use asynchronous 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.

Literature provides several approaches that deal with correct 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 models serve as basis for the realizations (software only), and they need some adaptations before they can be implemented in an asynchronous environment. For example, some approaches require that the model specifications are restricted to a certain modeling language subset that can asynchronously be implemented (Boudol 1992). Other approaches use a protocol to negotiate which components will communicate (Demaine 1998). A common challenge in these approaches is the implementation of the non-deterministic choice operator (Palamidessi 1997), since this may offer many communication alternatives of which only one alternative may be chosen.

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 (both hardware and software) and of the infrastructure. When the approaches found in literature would be applied, the models would need to be adapted for asynchronous implementation, e.g. untranslatable language constructs 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. This also means that when a non-deterministic choice appears in a component design, it must also be modeled ‘as is’ such that potential problems caused by it in an asynchronous environment can be analyzed. Solving these problems is not part of the modeling (as in the approaches found in literature), but of the design activities. 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 designed in system design D and realized in infrastructure *I* is expressed in the synchronous modeling language, similar to (Halbwachs and Baghdadi 2002). In this way, we can use the powerful techniques available for synchronous models to analyze the system behavior in an asynchronous environment. For the integration and testing of models and realizations, an asynchronous infrastructure is used that implements the communication behavior as it was designed and modeled. In this way, the

analysis results based on the synchronous system model remain valid during the integration and testing of models and realizations in an asynchronous environment.

The structure of the paper is as follows. The next section contains an overview of the forms of infrastructure *I* used in the MBI&T method. Subsequently, practical examples of modeling, analysis, and implementation of typical interaction types are given. The last section contains some concluding remarks.

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. These three forms are described in this section.

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 Figure 2 shows two component realizations Z_1 and Z_2 (boxes) and the infrastructure realization Z_I (double lined arrows) that enables the communication between the components. Since Z_I is part of the real system in the real world, communication is asynchronous.

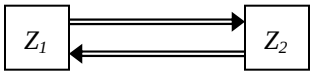

Figure 2. Infrastructure realization Z_I

Infrastructure model M_I. In the MBI&T method, the system components are modeled as M_i (see Figure 1). Besides the components, also the infrastructure that enables component interaction is 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. 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. Suppose that a system model with a synchronous abstraction of the infrastructure is found to be correct during analysis. 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 checked during model-based analysis are influenced by the infrastructure behavior, safety is not guaranteed in the realization environment, possibly resulting in hazardous situations.

When the infrastructure details are taken into account during the modeling and analysis of the system, the asynchronous behavior of the real infrastructure 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, asynchronous communication can be modeled in a synchronous language such as the process algebraic language used in the MBI&T method, similar to (Halbwachs and Baghdadi 2002). Additional processes are placed between two component processes to model the behavior of that particular component interaction. 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 Figure 3 shows four processes (circles) which, conforming to the process algebraic language, use synchronous communication (single lined arrows). The processes M_1 and M_2 represent the component models. The processes between M_1 and M_2 represent the infrastructure model M_I, resulting in asynchronous communication behavior between component models M_1 and M_2.

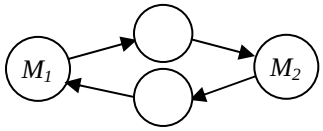

Figure 3. Infrastructure model M_I

Model-based integration infrastructure MZ_I. Besides the infrastructure realization Z_I and the infrastructure model M_I, another form of infrastructure is needed in the MBI&T method. A so-called model-based integration infrastructure MZ_I is used to integrate combinations of models and realizations, i.e. implementing the component interaction as designed in D and modeled in M_I. To enable this, MZ_I should satisfy several requirements.

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. The components only need to connect and communicate with the middleware and do not depend on the form, name, and location of the other components. In the MBI&T method, the model-based integration infrastructure MZ_I is also based on middleware. An example is given in the next section of this paper.

Connecting components to middleware requires that the communication paradigms used by the component models or realizations are 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 Figure 4 shows the integration of a model M_1 and a realization Z_2 using middleware (vertical double headed arrow). Both components are connected to the middleware via connectors (small boxes) that adapt the communication paradigm of M_1 (single lined arrows) and the communication paradigm of Z_2 (double lined arrows) to the middleware. The middleware is configured such that the component interaction corresponds to that of Figures 2 and 3.

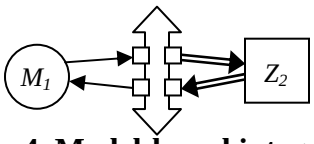

Figure 4. Model-based integration infrastructure MZ_I

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
   a. With synchronous abstraction of the infrastructure.
   b. With infrastructure model M_I.
3. For each realized component Z_i:
   a. Replacement of model M_i by realization Z_i, via model-based integration infrastructure MZ_I.
   b. System testing of the integrated system obtained in 3a.

In the following section, we practically illustrate the modeling, analysis and implementation of interaction types typically used in the high-tech multi-disciplinary systems we are considering in our project, taking the wafer scanner from ASML (ASML 2007) as an industrial example.

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 modeling and analysis of concurrent behavior.

In the following paragraphs, we give 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 the model-based integration infrastructure MZ_I using middleware.

In the examples, we use the process algebraic language χ (Chi) (van Beek et al. 2006) and its toolset (Systems Engineering Group 2007) to model and analyze components and systems. As demonstrated in an industrial case study (Braspenning et al. 2006b, 2006c), the χ toolset allows simulation and model checking of system models, as well as real-time execution of component models integrated with other (non-χ) components via middleware (Millard et al. 2006).

The middleware used as basis for the model-based integration infrastructure MZ_I in the MBI&T method and in the χ toolset is based on communication via the publish-subscribe paradigm (Eugster et al. 2003),

which satisfies all requirements for MZ_I 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. Communication via the publish-subscribe paradigm is simple. Components can publish messages of a certain type (also called topic) to the middleware, and they can subscribe to message types published by other components. 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. Different types of component interaction, also modeled in different models M_I, can be configured by quality of service (QoS) properties like the number of messages to keep as history, or 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 (for published message types) and read actions (for subscribed message types) of the publish-subscribe middleware. The χ toolset includes an automatic generator of connectors for a χ 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.

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 errors 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. Since the blocking function call is a special case of non-blocking (with no statements between request and reply and no time-out), we will only discuss the more generic non-blocking function call here.

Important properties of function calls 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 an error detection mechanism could be captured in a property 'time-outs may never be triggered', however this property does not relate to infrastructure but to required system behavior.

Function calls can easily be modeled in χ. Asynchronous communication can be modeled in a synchronous modeling language such as χ by including a 'buffer' process between two communicating processes. The χ code of a buffer process B is shown in Figure 5. This process has two communication channels, input *a* and output *b*, for messages of type *msg*, and shows repetitive behavior (denoted by *). Each repetition starts with guarded expressions (denoted by ->) 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*). The buffer overflow check is followed (denoted by sequential composition ;) by two alternatives (denoted by |) 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*, 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.

```
proc B(chan a?, b!: msg, val n: nat) =
|[ var xs: [msg] = [], x: msg
:: *( ( len(xs)  > n -> !!"buffer overflow"
      | len(xs) <= n -> skip
      )
    ; ( a?x; xs:= xs ++ [x]
      | len(xs) > 0 -> b!hd(xs); xs:= tl(xs)
      )
    )
]|
```

Figure 5. Buffer process B

Using multiple instantiations of buffer process B, we can model a function call as shown in Figure 6. The χ code shows four processes in parallel composition (denoted by ||). The first process is a partial specification (denoted by …) 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_req!arg*) and receiving an asynchronous reply of the function with the results (*f_rep?res*). Between these two statements, other internal actions (denoted by …) 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, 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 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 *arg*, the result of the function executed on *arg* is sent back as a reply. Finally, two buffer processes B 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 Figure 3. 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. A function required by multiple clients can be modeled in a similar way.

```
(  ( ...
   ; f_req!arg
   ; ...
   ; (f_rep?res | delay t -> !!"time-out")
   ; ...
   )
|| *( f_req'?arg; f_rep'!f(arg) )
|| B(f_req,f_req',1)
|| B(f_rep',f_rep,1)
)
```

Figure 6. M_I for function call

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 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 (Braspenning et al. 2006b). In the model of the server, each incoming request is immediately followed by sending the reply, so the number of requests is always equal 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: $0 \leq$ nr_requests – nr_replies ≤ 1. The wait/time-out property is covered in the infrastructure model M_I of Figure 6, because the communication in the χ model is urgent, i.e. a process may not delay if a communication

action is enabled. This implies that the time-out (*delay t*) can only be triggered when after *t* time units the receive action (*f_rep?res*) 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 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. Figure 7 shows the implementation of MZ_I for the example of Figure 6 with a model of client M_C, a realization of server Z_S, and the required message types for requests and replies. The client is configured as publisher of requests for function *f*, and it is subscribed to replies of *f*. 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 message types on the publish-subscribe middleware. The time-out is implemented in the connector, which 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 it notifies the model that a time-out has occurred. For the integration of a client or server realization, the connector should translate between publish-subscribe messages and real function call requests and replies. For example, when the connector of server realization Z_S receives a request *f_req* via the middleware, it should call the real function *f* of Z_S, after which the result is published on the middleware as *f_rep*. For a realization, the time-out functionality is included in the component realization itself.

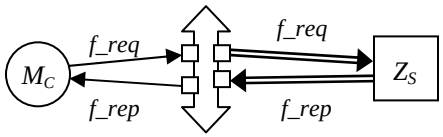

Figure 7. MZ_I for function calls

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.

Below are some typical sequential logic properties, taking the SR-latch as an 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 χ (van Beek et al. 2006), we restrict ourselves to the discrete-event version of χ, in which we

abstract from the continuous behavior of the SR-latch. A discrete-event model of the SR-latch is shown in Figure 8. The highest level parallel composition (first || in code) contains the processes of the sending and the receiving component 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* and later, when it is ready, to *true*. The receiving process checks whether the other component is ready via the latch value *ready*. 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 process to the model of the receiving component, denoted by the parallel composition on the second level of the model (second || in code). This additional process is always able to receive new values of the *ready_latch* from the sending component. The variable *ready*, which is used to store the latest latch value, is shared with the other processes of the parallel composition. In this way, only the latest latch value is considered in the behavior of the receiving component.

```
(  ( ready_latch!false
   ; ...
   ; ready_latch!true
   )
|| (  (     ready -> skip
       | not ready -> !!"error"
       )
   || *(ready_latch?ready)
   )
)
```

Figure 8. M_I for SR-latch

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 or reset to false 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 publish-subscribe middleware acts as a FIFO buffer that does not store its value after delivering it to the receiving component. However, for the SR-latch, it should store the value that is last received from the sending component. This is achieved by configuring the publish-subscribe middleware with the QoS property 'keep one message as history'.

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.

Concluding remarks

In this paper, we presented an approach on how to deal with infrastructure in the MBI&T method in order to guarantee that the analysis results on the system model also hold when the models are combined with realizations and the real infrastructure. In the presented approach, the behavior of the infrastructure realization Z_I is modeled as infrastructure model M_I. This model M_I is included during system model analysis, and implemented in a model-based integration infrastructure MZ_I, using a publish-subscribe middleware, for integration and testing with component realizations. For the examples taken from industrial practice, the transition from synchronous process algebraic models to distributed asynchronous realizations is rather straightforward. The synchronous models provide a good understanding of system behavior and enable verification of properties related to both infrastructural and system behavior. The integration of models and realizations allows fast and cheap system integration and testing several months before real integration and testing, as shown in an industrial case study (Braspenning et al. 2006b, 2006c). The described approach can be applied to other interaction types in a similar way. The authors would like to thank Albert Hofkamp, Ralph Meijer, Johan Neerhof, and Jan Tretmans for the fruitful discussions and their valuable comments.

References

ASML website, 2007. http://www.asml.com

Baeten, J.C.M. and Weijland, W.P., *Process algebra*. Cambridge University Press, 1990.

van Beek, D.A., Man, K.L., Reniers, M.A., Rooda, J.E., Schiffelers, R.R.H., Syntax and consistent equation semantics of hybrid Chi, *JLAP*, 68(1-2):129–210, 2006.

Boehm, B.W. and Basili, V.R., Software defect reduction top 10 list. *IEEE Computer*, 34(1):135–137, 2001.

Boudol, G., Asynchrony and the π-calculus, Technical report 1702, INRIA, 1992.

Braspenning, N.C.W.M., van de Mortel-Fronczak, J.M., and Rooda, J.E., A model-based integration and testing method to reduce system development effort, in Proc. of MBT'06, *ENTCS* 164(4):13–28, 2006a.

Braspenning, N.C.W.M., Bortnik, E.M., van de Mortel-Fronczak, J.M., and Rooda, J.E., Model-based system analysis using χ and Uppaal: an industrial case study, SE report 2006-07, Eindhoven University of Technology, ISSN 1872-1567, 2006b.

Braspenning, N.C.W.M., van de Mortel-Fronczak, J.M., Rooda, J.E., Model-based integration of high-tech multi-disciplinary systems, 12e Nederlandse Testdag, Veldhoven, The Netherlands, 2006c. http://www.asml.com/testdag2006

Demaine, E.D., Protocols for non-deterministic communication over synchronous channels, in Proc. of IPPS-SPDP'98, pp. 24–30, 1998.

Eugster, P.Th., Felber, P.A., Guerraoui, R., Kermarrec, A., The many faces of publish-subscribe, *ACM Computing Surveys*, 35(2):114–131, 2003.

Halbwachs N. and Baghdadi S., Synchronous modeling of asynchronous systems, in Proc. of EMSOFT'02, *LNCS 2491*, 2002.

Millard, P., Saint-Andre, P., Meijer, R., XEP-0060: Publish-Subscribe, Jabber Software Foundation, 2006.

Palamidessi, C., Comparing the expressive power of the synchronous and the asynchronous π-calculus, In Proc. of POPL'97, pp. 256–265, 1997.

Systems Engineering Group, Mechanical Engineering Dept., Eindhoven University of Technology, Chi website, 2007. http://se.wtb.tue.nl/sewiki/chi

Biography

N.C.W.M. Braspenning graduated at the Systems Engineering group of the Mechanical Engineering Department of Eindhoven University of Technology, the Netherlands. In 2003, he started his Ph.D. project 'Model-based integration and testing of high-tech multi-disciplinary systems'.

J.M. van de Mortel-Fronczak graduated in computer science at the AGH University of Science and Technology of Cracow, Poland, in 1982. In 1993, she received the Ph.D. degree in computer science from the Eindhoven University of Technology, the Netherlands. Since 1997 she has worked as an assistant professor at Eindhoven University of Technology, focusing on supervisory machine control.

J.E. Rooda received the M.S. degree from Wageningen University of Agriculture Engineering and the Ph.D. degree from Twente University of Technology, Enschede, the Netherlands. Since 1985 he has been Professor of the Systems Engineering group at the Mechanical Engineering Department of Eindhoven University of Technology, the Netherlands. His research interests are analysis, modeling, and control of manufacturing systems and supervisory machine control.