Developing Distributed Systems Step by Step with UML-RT

Robert Sandner[*]
Institut für Informatik
Technische Universität München
80333 München, Germany
email: Robert.Sandner@in.tum.de

Abstract

In this paper, we argue that Case tools provide a valuable support for the presentation and analysis of models, but more support for a stepwise development process is still needed. In particular, we focus on the development steps for behaviour models, and which support can be provided by Case tools for them. Notions of refinement can help to provide this support. We discuss the use of existing refinement calculi for this task and present additional high level refinements necessary to support important practical development steps. In particular, we address two important issues: Interface refinement and the introduction of time.

1. Introduction

Software systems development is a complex task, in particular for technical applications, where real-time requirements and various physical influences have to be considered. A number of visual modeling languages have been developed to cope with this complexity. Whereas for business software, the *Unified Modeling Language* UML [30, 29, 28] has become a dominating standard, in the field of technical applications a number of more specialized, tool supported languages have been quite successive, e.g. ROOM [35], Statemate [14], Matlab [37, 38] and Ascet-SD [2]. Currently, a *UML profile for Scheduling, Performance and Time* [26, 25] is being standardized by the Object Management Group (OMG), influenced by concepts of ROOM [22, 36]. This profile is commonly known under the name UML-RT. In this paper, we address the stepwise development of technical, time critical software systems using the notations of UML-RT.

Visual modeling languages - especially the UML - are supported by numerous commercial Case tools today. Due to the participation of Case tool vendors in the standardization process, the same can be expected for UML-RT. However, current Case tools have their strengths in the presentation and analysis of models. A structured development process is mainly provided by the definition of views for the separation of concerns. This is not sufficient to structure a stepwise development of complex systems: We also need guidance how to change a model in a development step without destroying earlier established desired properties, to which someone else may rely on. Most Case tools, however, only provide mechanisms to check the result of a change for syntactical consistency, but do not take care of the semantics of models.

The methodological support needed for development steps can be achieved by notions of refinement, based on a precise semantics for the modeling techniques. Refinement approaches are an active research field in academia. However, they are poorly used in industrial practice so far. A reason is that only a few approaches are designed to be used with popular modeling languages. Further, most approaches which do so focus on too fine grained, isolated refinement steps like removing or adding a single transition in a Statechart. These refinement principles provide a firm basis for systematic system development, but they need to be accompanied by higher level refinement principles which reflect important practical development tasks. We concentrate on the modeling and refinement of structure and behaviour using UML-RT Capsule diagrams and Statecharts. We discuss two particular refinement principles:

Behavioural interface refinement: Although it is almost a truism that a system has to be broken up into manageable components connected by well defined interfaces, usually these interfaces are not perfect at the first attempt. Reasons include both flaws in a design but also organizational matters: For developing a component of a larger systems, the deadlines of the project may make it impossible to wait until all interfaces of neighboring components are absolutely definite. To allow interfaces to be developed themselves step by step, we present an easy to use refinement principle which allows to change the interface of a component together with the Statechart specifying its behaviour. A set of easily checkable rules is given to ensure that the change is a sensible refinement.

[*] Research supported by the DFG project *InTime*.

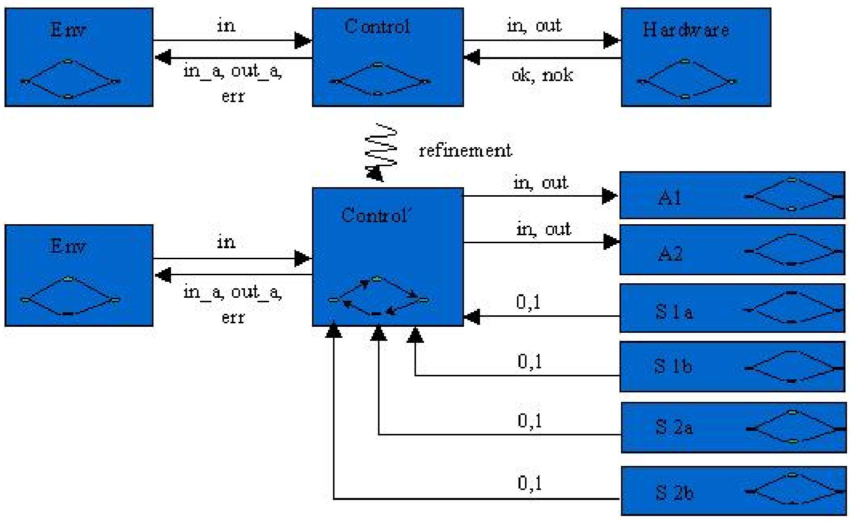

Figure 1. Example: Application scenario for behavioral interface refinement

Incorporating time: Although real-time is a critical issue for technical applications, dealing with time in a too detailed way at the very beginning of the development hardly allows the refinement of models. We suggest to ignore execution times of components as long as possible. In order to do so, we present a simple approach which helps to introduce time aspects into models only if they become necessary.

Outline: In Section 2 we sketch basic ideas of a development process which motivate the refinement principles presented in the paper. Section 3 provides an overview on existing refinement calculi. High level refinement rules which close the gap between theory and practice are discussed in Section 4. Finally, Section 5 contains concluding remarks.

2. Refinement: Existing approaches

Notions of refinement have a long tradition in the theory of programming. Notions of refinement for programs have e.g. been presented in [16]. A number of theories have been developed to generalize the notion of refinement. An underlying theory used by many approaches are process algebras [15, 10]. Notions of refinement which have been developed include trace refinement [23] and failure preorder [16]. For a discussion of their interrelations, see [24]. Another powerful framework for the definition of refinement calculi are stream processing functions, as used in the FOCUS framework [8, 9]. They allow to understand refinement in a quite simple manner, namely in terms of logical implication (cf. [5, 6]) and yield the logical basis for our approach. Interpreting refinement as logical implication has also been applied on timed specifications: An approach refining functional timed specifications to programs can be found in [33]. The use of different timing models and their interrelations are closely discussed in [7]. Not only behaviour is a matter of refinement: Structural refinement of specifications is considered in [27]. A well investigated field is the refinement of communication principles, see e.g. [11, 32, 3]. Refinements of specifications written in different languages are discussed in [12].

Of particular interest for us are approaches with apply notions of refinement on visual modeling techniques, especially on behaviour modeling. Refinement rules for Statecharts and related formalisms are presented in [31, 34, 19]. A refinement calculus for MSCs has been developed in [20]. These approaches have in common that they mostly deal with very fine grained refinements, e.g. adding a transition to a Statechart. Refinement of interfaces is only allowed in a restricted manner. Refinement of timed specifications is not considered. In conjunction with visual modeling languages, this is still an open field in research. In our setting, the refinement rules provided by the approaches discussed above are augmented by the principles discussed in Section 4.

3. Development Scenario

In this paper, we concentrate on structure and behaviour modeling using the UML-RT notations Capsule diagrams and Statecharts which have strongly been influenced by the ROOM method. It should be noticed that UML-RT Statecharts differ from standard UML Statecharts: They introduce some syntactical extensions and - more important - their semantics differs from the one given in [13] since they employ an asynchronous execution model. This will be addressed in more detail in Section 4. To motivate the refinement principles presented there, we give a sketch of a simplified development process which uses Message Sequence Charts (MSCs) [18], Capsule diagrams and Statecharts. Of course, these ideas fit into a full scale development process.

As a starting point of the development process, the components of the system are identified using Capsule

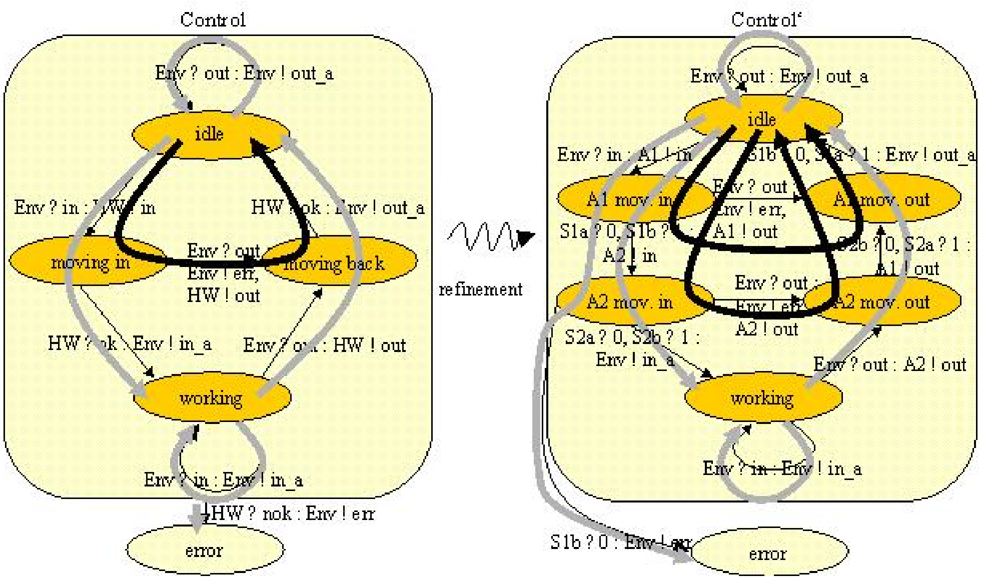

Figure 2. A mapping of scenarios

diagrams. Next, their interactions are explored in example application scenarios using both Capsule diagrams and MSCs. Scenarios serve both as requirements specifications and as a first design: Using the algorithm presented in [20, 21], Statecharts can be generated from them. These generation saves a lot of work since the information gained from the MSCs need not to be re-engineered in the design of the system components. The generated models are usually not complete and may be too application specific. Thus they need to be refined and generalized. Besides reasons given in the introduction, these is one of the motivations for our concept of interface refinement. In the following, we focus on the refinement of these generated models towards a detailed design.

We recommend that in the first steps, the refinement of the *logical* behaviour and the structure of the system should take precedence. Both refinements may be carried out interleavingly. Although some performance analysis may be carried out earlier, we suggest to incorporate time as an integral part of a model only at the late stages of development. Incorporating time dependency and execution speed into models completes the design process. This way, the design process ends up with models that can be used immediately as a starting point of the implementation.

4. High level refinements for important development tasks

In this section, we discuss two important refinement principles which play a crucial rule in practical system development. As discussed in Section 2, structural and behavioural refinement often need to be combined. In Section 4.1 we present a principle to refine both the syntactic interface and the behaviour of a component. It requires a minimum of theoretical background for the developer. We also suggested in Section 2 to postpone the treatment of real-time aspects as long as possible. This eases the "logical design" of the system, but requires a refinement principle introducing time into models, which is discussed in Section 4.2. Finally, we give an outlook on further useful refinements and future work in Section 4.3.

4.1. Behavioural interface refinement

Behavioural interface refinement allows to refine the interface of a component together with the Statechart which models its behaviour. Applications of such a refinement have already been discussed above: Interface refinements may be necessary because of design flaws, organizational reasons or generalizations of interfaces gained from application scenarios.

Application scenario

In the following, we will use a (simplified) example taken from mechatronics. The upper diagram in Figure 1 shows a component `Control`, which provides services to `Env`, using the hardware it controls. The behaviours of the components are specified by Statecharts. Since the details of the hardware are not known at the start of the development, we need to refine the hardware interface in subsequent development steps, as shown in the lower diagram. Of course, the refined component `Control′` should preserve the behaviour of `Control` shown to `Env` in some sense. However, this depends on the meaning of the messages transmitted along both hardware interfaces, e.g. which messages correspond to an `ok` message by the abstract hardware.

We need a simple way for the developer to specify this interrelation. Let us consider the Statecharts of both components: The left Statechart in Figure 2 models the behaviour of `Control`, the right one the behaviour of `Control'`. To establish a refinement relation between them, we use a mapping between their interaction scenarios. In our setting, we map *concrete* scenarios to *abstract* ones. Thus, the mapping goes into the opposite direction to the development process. As we will see below, this simplifies the definition of constraints which are desirable for methodological reasons. The mapping can be easily defined by identifying execution paths through the Statecharts which start and end in a state, as illustrated in Figure 2. For example, the scenarios identified by the black arrows in `Control'` are both mapped on the black marked scenario in `Control`. The gray arrows show scenarios which also have to be mapped to define a total mapping[1]. To be more precise, we define a mapping of scenarios by cases. The cases for the black paths are shown below. For matters of space, we use the abbreviations *C, C'* and *HW*:

$$\begin{gathered}
Env \overset{in}{\to} C' \;;\; C' \overset{in}{\to} A1 \;; \\
Env \overset{out}{\to} C' \;;\; C' \overset{err}{\to} Env \;;\; C' \overset{out}{\to} A1 \;; \\
S1b \overset{0}{\to} C' \;;\; S1a \overset{1}{\to} C' \;;\; C' \overset{out\_a}{\to} Env \\
\Downarrow \\
Env \overset{in}{\to} C \;;\; C \overset{in}{\to} HW \;; \\
Env \overset{out}{\to} C \;;\; C \overset{err}{\to} Env \;;\; C \overset{out}{\to} HW \;; \\
HW \overset{ok}{\to} C \;;\; C \overset{out\_a}{\to} Env
\end{gathered}$$

$$\begin{gathered}
Env \overset{in}{\to} C' \;;\; C' \overset{in}{\to} A1 \;; \\
S1a \overset{0}{\to} C' \;;\; S1b \overset{1}{\to} C' \;;\; C' \overset{in}{\to} A2 \;; \\
Env \overset{out}{\to} C' \;;\; C' \overset{err}{\to} Env \;;\; C' \overset{out}{\to} A2 \;; \\
S2b \overset{0}{\to} C' \;;\; S2a \overset{1}{\to} C' \;;\; C' \overset{out}{\to} A1 \;; \\
S1b \overset{0}{\to} C' \;;\; S1a \overset{1}{\to} C' \;;\; C' \overset{out\_a}{\to} Env \\
\Downarrow \\
Env \overset{in}{\to} C \;;\; C \overset{in}{\to} HW \;; \\
Env \overset{out}{\to} C \;;\; C \overset{err}{\to} Env \;;\; C \overset{out}{\to} HW \;; \\
HW \overset{ok}{\to} C \;;\; C \overset{out\_a}{\to} Env
\end{gathered}$$

Conditions for a refinement

Of course, just to define a function between scenarios does not necessarily lead to a sensible refinement. However, the mapping of the scenarios shown above can be used to define a refinement relation between `Control` and `Control'`. `Control'` is a refinement of `Control` if each *complete execution trace* of `Control'` relates properly to an execution trace of `Control`. This coincides with common notions of refinement: Each system run of the refined system must also be possible in the more abstract system in terms of its interface definition. We have to define a refinement relation on complete traces: It is given as a total mapping from traces of `Control'` to traces of `Control`, defined inductively by the cases shown above which map finite scenarios. To get a proper inductive definition of a total mapping, we need to impose two conditions on the scenarios identified in the refined component:

- *No scenario defining the mapping may be a prefix of another.* Otherwise, the mapping on execution traces would be ambiguous. This condition is well known as the Fano condition from coding theory.
- *The scenarios must provide a complete path coverage. The mapping of the finite scenarios must be total*[2]. This is necessary for that the inductive definition defines a total mapping of execution traces. A partial mapping would leave parts of the behaviour unconstrained. This clearly does not coincide with refinement.

For methodological reasons, standard notions of refinement also imply the following conditions:

- *The cases need to define a function.* This is already given by definition since each scenario has to be mapped on a scenario in the Statechart of the abstract component. A function ensures that behaviour which can be distinguished at the abstract level can also be distinguished at the concrete level.
- *The range of the mapping provides a complete path coverage in the abstract component.* This way, we disallow to throw away behaviours shown to other components not subject to the interface refinement. This would only be a refinement if we omit a nondeterministic behaviour in the abstract component but *not* in other cases. Reduction of non determinism can be performed using standard refinement rules, e.g. given in [31, 34]. Therefore, we can exclude it completely here to avoid to make things unnecessarily complicated.
- *The projection of each mapping to unchanged interfaces is the identity.* This ensures that causality dependencies of all actions not subject to the interface refinement are preserved[3]. Note that causality dependencies on the refined interface are only subject to the scenarios. Fortunately, their order cannot be changed by the inductive definition given above.
- *Each image of a scenario starting in an initial state must also start in an initial state. Each image of a scenario ending in an terminal state must also end in a terminal state.*
- *Concrete scenarios need to be finite.* Together with the two previous items, this condition ensures that termination is preserved: If the abstract component terminates, also the refined component does.

[1]For clarity, we only showed one error path in the system. Of course, the real system, and also the mapping, would be more complex if all errors are considered.

[2]For each scenario, there must be a non-empty scenario to which this scenario is mapped.

[3]The term causal means that each two events in the concrete model must be in the same order as their counterparts in the abstract one.

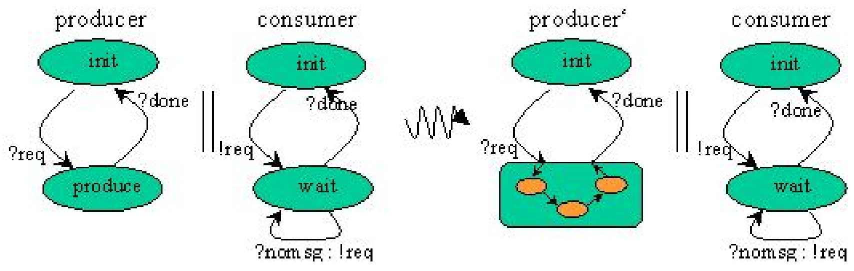

Figure 3. Implicit time dependency due to synchronous execution

Applicability in practice

At a first glimpse, the conditions introduced above seem to be rather extensive. However, if we skip the methodological justifications, they are relatively short: We need a total function from concrete to abstract scenarios, which has to ensure complete path coverage for its domain and image, and which has to be the identity on unchanged parts of the model. These conditions are understandable without deep mathematical knowledge, and they can be checked fully automatically by tool support. Although we have not yet gained much experience with large industrial case studies using this approach, we are confident that it scales up well to real life applications.

It should be noted that in the literature, much simpler refinement rules for interface refinement can be found (e.g. in [6]). They introduce mappings between input and output *messages* of an abstract and a refined component. However, they are not convenient for complex cases as the example in Figure 2: A message by a sensor can both mean `ok` and `nok`, depending on the state of the interaction with the component `control`.

The approach can be used in different situations: A given Statechart can be verified whether it refines an abstract one, as shown above. For components of moderate size, it is also possible to specify the refined scenarios independently and generate the Statechart model for the refined component using the algorithm presented in [20].

4.2. From untimed to timed models

Introducing time aspects late eases the development in the earlier phases. Since both developing an appropriate “logical” design and treating real-time requirements are complex issues, each of these tasks should be supported by best suited modeling environments. Statecharts have proved their adequacy for both levels of abstraction. The question addressed here is which underlying semantics is appropriate for a logical design and which for time dependent models, and how they can be integrated. In the following, we will consider time dependent models first. Given these models, we discuss which abstractions are desirable in the course of a logical design.

A semantics for timed models

For time dependent models, we are looking for a semantics which is easy to use and especially allows to easily analyze time aspects of a model. The crucial decision which has to be made thereby is the choice of an appropriate execution model. For the analysis of time dependent models, we need an execution model which allows to predict the execution time of all actions of a component. A quite useful and widely used approach is to synchronize the execution of all components in a system by global clock ticks: Each time the clock ticks each component performs one step - i.e. one transition - of execution. Because of this synchronization mechanism, we refer to this approach as a *synchronous execution model*. Often, a more refined variant of this model is used: Components are driven by the ticks of local clocks, but all these clocks are synchronized with respect to a global clock. This allows to model different execution speeds in a system. Synchronous execution models are widely used in industrial practice: For example, they are used in the Case tools Matlab/Stateflow [37, 38] and Ascet-SD [2], but also in popular implementation platforms based on Programmable Logical Controls (PLCs): The visual programming language Higraph [1] uses the synchronization model described above.

It is sometimes criticized that synchronous execution models are inappropriate for the development of distributed systems. The major reason is that they do not express delays in communication. Because of matters discussed below, we agree to this opinion in the sense that they are inadequate for *development steps* in the early design of a system. However, they are quite well suited for the *analysis* of implementation oriented models since they ease the prediction of execution times. Synchronous execution models are typically combined with unbuffered communication or at least with bounded buffering. Together with the restriction of interleaving through clock ticks, this enables the analysis of models by fully automatic techniques like model checking, successfully used e.g. in the AUTOFOCUS Case tool [17, 39, 4]. Further, the absence of expressing delay in communication is not really a disadvantage: If we are interested in the characteristics of a communication medium, we can model it as a component. Since the characteristics of communication mediums differ widely, they can not be commonly expressed by a single semantics. Thus we will probably be forced to do quite the same using any other execution model.

However, synchronous execution models do not allow time independent specifications: Since each transition is taken at a tick of the global clock, the relative execution time of computations is always implicitly specified by the number of transitions. As an example, the component `producer` in Figure 3 is twice as fast than the refined component `producer'`. As a consequence, the component `customer` will work well with `producer` but not with `producer'`. Although the situation above could be easily avoided, it is hard to avoid such effects in general. Especially, these effects prevent the definition of simple refinement rules for components.

A semantics for logical design

The effects shown above arising from a too close coupling of components by synchronous execution can be avoided by *asynchronous execution models*. In this approach, the time instant at which a component reacts to a received message is left unspecified. It is only defined that the reaction happens after the receipt. Although this approach abstracts from execution times of actions, it is successfully used in the development of real-time systems: The ROOM [35] method provides timer facilities which allow to introduce timed behaviour like delay or timeouts. These concepts seem also to be adopted in the standardization process of the UML profile for Scheduling, Performance and Time. Although system development makes use of simulation facilities, the scheduling algorithm of the underlying ROOM virtual machine is not explicitly defined in order to keep the modeling language independent from platform dependent implementations. Asynchronous execution models facilitate the definition of refinement calculi. However, since the analysis of real-time properties requires assumptions on the scheduling of actions, and asynchronous execution also complicates automatic reasoning using techniques like model checking, we suggest to use both synchronous and asynchronous execution in the development process.

Introducing time as a refinement

To provide a most appropriate semantics for both levels of abstraction, we propose to use the asynchronous model at the beginning of the development and switch to the synchronous model later on. This step can be quite easily understood as a sensible refinement step: It is a restriction of *timed* nondeterminism in the sense that the time instant at which a transition is taken is left unspecified before the refinement and is fixed afterwards. This can be easily formulated if asynchronous execution are formulated in a logical model using time ticks $\surd$: A trace in an asynchronous model

$$HW \overset{ok}{\to} C \; ; \; C \overset{fin}{\to} Env$$

corresponds to the set of traces

$$\surd^{\omega} \; HW'' \overset{ok}{\to} C'' \; \surd^{\omega} \; C'' \overset{fin}{\to} Env''$$

Therefore, switching from asynchronous to synchronous execution complies with the classical notion of refinement and can be formalized on a simple mathematical basis. It also complies with the practical development process in ROOM which requires to develop a scheduler in the course of implementing models. For that reason, it is surprising that this step is not treated in the literature[4].

4.3. Further refinements and future work

We expect the refinement principles discussed above to be quite useful in practice. Yet, they form only a step in the development of a refinement calculus for the development of time critical distributed systems. Our work on these principles is far from being complete: To gain more experience, more real life applications need to be tried using the approaches. Further, we intend to formalize the principle using the logical framework FOCUS [8, 9] in order to give a formal proof that it coincides with common notions of refinement.

Both principles need also be accompanied by further rules. The switch from an asynchronous to a synchronous execution model mainly incorporates time in terms of *speed* into models. It is also necessary to support the development process by refinement rules which allow the introduction and adaption of timers into models. Another interesting issue is the treatment of partial transition relations in Statecharts. Whereas most Case tools provide a default completion (typically ignoring messages), approaches like [31, 34] define a closure allowing arbitrary behaviour, which nicely combines with refinement principles. Also an issue of practical interest is the abstraction from communication paths in a system at some stages of development.

5. Conclusion

We have argued that although Case tools used in industrial practice today provide a valuable support for presentation and analysis of models, there is a lot of improvement needed to support consistent development steps. Notions of refinement are well suited to cope with this challenge. We have shown that there exist a huge number of refinement principles originating from theoretical approaches, which provide a firm basis but need to be accompanied by additional refinement principles which support development steps important in practice. In particular, we have presented two principles for interface refinement and for the introduction of time into models. These principles are based on simple mathematical concepts and are compatible with industrial Case

[4]Only synchronous and asynchronous communication is treated, see Section2.

tools and implementation platforms. They work well with, but are not limited to, the notations of UML-RT. It is our hope that notions of refinement will be taken up by Case tools in the future to support a methodologically founded development process.

Acknowledgments: The author owes thanks to Ingolf Krüger, Bernhard Rumpe and Michael van der Beeck for intensive discussions on refinement rules. Heiko Lötzbeyer and Wolfgang Prenninger read previous drafts of this text and provided valuable comments.

References

[1] *SIMATIC Software - Higraph für S7-300/400 Zustandsgraphen programmieren - Handbuch*. Siemens, 1997.

[2] *Ascet- SD 4.0 Users Guide*. ETAS-Engineering Tools, Stuttgart, 2000.

[3] A. Beneviste, B. Caillaud, and P. L. Guernic. From synchrony to asynchrony. In J. M. Baeten and S. Mauw, editors, *CONCUR99 (Concurrency Theory)*, volume 1664, pages 162–177, Eindhoven, The Netherlands, aug 1999. Springer Verlag, LNCS.

[4] P. Braun, H. Lötzbeyer, B. Schätz, and O. Slotosch. Consistent integration of formal methods. In *Proc. 6th Intl. Conf on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, volume 1787. Springer Verlag, LNCS, 2000.

[5] M. Broy. Compositional refinement of interactive systems. In *DIGITAL Systems Research Center SRC 89*, 1992.

[6] M. Broy. (Inter-)Action Refinement: The Easy Way. *Program Design Calculi, Series F: Computer and System Sciences. Vol. 118.*, 1993.

[7] M. Broy. Refinement of time. In *ARTS'97*. to appear, 1997.

[8] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The Design of Distributed Systems – An Introduction to FOCUS. Technical Report SFB 342/2/92 A, Technische Universität München, 1992.

[9] M. Broy and K. Stølen. Focus on system development. Book manuscript, January 2000.

[10] R. Cleaveland and G. Smolka. Process algebra. *Encyclopedia of Electrical Engineering*, 1999.

[11] C. Fischer and W. Janssen. Synchronous development of asynchronous systems. In U. Montanari and V. Sassone, editors, *CONCUR96 (Concurrency Theory)*, volume 1119, pages 735–750, Amsterdam, The Netherlands, aug 1996. Springer Verlag, LNCS.

[12] M. Große-Rhode. A compositional comparison of specifications of the alternating bit protocol in ccs and unity based on algebra transformation systems. In K. Araki, A. Galloway, and K. Taguchi, editors, *1st International Workshop on Integrated Formal Methods (IFM'99)*, pages 253–272, York, UK, jun 1999. Springer Verlag.

[13] D. Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

[14] D. Harel, H. Lachover, A. Naamad, A. Pnuel, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–413, 1990.

[15] M. Hennessy. *Algebraic Theory of Processes*. The MIT Press, Cambridge, Mass., 1988.

[16] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1986.

[17] F. Huber, S. Molterer, B. Schätz, O. Slotosch, and A. Vilbig. Traffic Lights - An AutoFocus Case Study. In *1998 International Conference on Application of Concurrency to System Design*, pages 282–294. IEEE Computer Society, 1998.

[18] ITU-T. *Z.120 – Message Sequence Chart (MSC)*. ITU-T, Geneva, 1999.

[19] C. Klein. *Anforderungsspezifikation durch Transitionssysteme und Szenarien*. PhD thesis, Technische Universität München, 1998.

[20] I. Krüger. *Distributed System Design with Message Sequence Charts*. PhD thesis, Technische Univerität München, 2000.

[21] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to Statecharts. In *Proceedings of DIPES'98*, 1999.

[22] A. Lyons. UML for Real-Time Overview. *Objectime Ltd.*, April 1998. `http://www.objectime.on.ca/otl/technical/umlrt.html`.

[23] M. Main. Trace, failure and testing equivalences for communicating processes. *Int. Journal of Prallel Programming*, 16(5):383–400, 1987.

[24] M.v.d.Beeck. Behaviour specifications: Semantics, equivalence and refinement. submitted for publication, 2000.

[25] OMG. Response to the OMG RFP for Scheduling, Performance, and Time. `http://www.omg.org/techprocess/meetings/schedule/UML_Profile_for_Scheduling_RFP.html`, 1999.

[26] OMG. UML^{TM} Profile for Scheduling, Performance, and Time - Request for Proposal. `http://www.omg.org/techprocess/meetings/schedule/UML_Profile_for_Scheduling_RFP.html`, 1999.

[27] J. Philipps and B. Rumpe. Refinement of pipe-and-filter architectures. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 – Formal Methods, Proceedings of the World Congress on Formal Methods in the Development of Computing System. LNCS 1708, pages 96-3115*. Springer, 1999.

[28] Rational. UML Notation guide, version 1.3. `http://www.rational.com/uml/resources/documentation`, June 1999.

[29] Rational. UML Semantics, version 1.3. `http://www.rational.com/uml/resources/documentation`, June 1999.

[30] Rational. UML Summary, version 1.3. `http://www.rational.com/uml/resources/documentation`, June 1999.

[31] B. Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. PhD thesis, Technische Univerität München, 1997.

[32] B. Schätz. *Ein methodischer Übergang von asynchron zu synchron kommunizierenden Systemen*. PhD thesis, Technische Univerität München, 1998.

[33] D. Scholefield, H. Zedan, and H. Jifeng. A specification-oriented semantics for the refinement of real-time systems. *Theoretical Computer Science*, 131:219–241, 1994.

[34] P. Scholz. *Design of Reactive Systems and their Distributed Implementation with Statecharts*. PhD thesis, Technische Univerität München, 1998.

[35] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley and Sons, Inc., 1994.

[36] B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. Available under http://www.objectime.com/uml, April 1998.

[37] The MathWorks Inc. Stateflow. `http://www.mathworks.com/products/stateflow`, 1999.

[38] The MathWorks Inc. MATLAB. `http://www.mathworks.com/products/matlab`, 2000.

[39] G. Wimmel, H. Lötzbeyer, A. Pretschner, and O.Slotosch. Specification Based Test Sequence Generation with Propositional Logic. *J. Software Testing, Verification & Reliability (STVR): Special Issue on Specification Based Testing*, 2000. To appear.