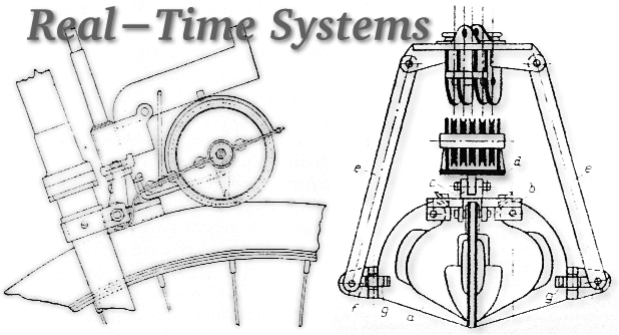
OMER Workshop Proceedings

PETER HOFMANN
ANDY SCHÜRR (EDS.)

May 28 - 29, 1999, Herrsching (Ammersee), Germany



Bericht Nr. 1999-01
Mai 1999

Universität der Bundeswehr München

Fakultät für

INFORMATIK

Werner-Heisenberg-Weg 39 • D-85577 Neubiberg

Contents

Foreword

Welcome to **OMER**, the 1st workshop on **O**bject-**O**riented **M**odeling of **E**mbedded **R**ealtime (RT) systems of the German Computer Science Society GI. The aim of this workshop is to bring together researchers and industrial professionals from a wide variety of backgrounds to discuss the present and future of object-oriented development of RT systems.

OMER is the 5th workshop of the working group GROOM (Grundlagen objektorientierter Modellierung), which belongs to the "GI-Fachgruppe" OOSE on Object-Oriented Software Engineering. Our previous workshops in München (1996, 1998), Mannheim (1997), and Frankfurt (1998) had the following main topics: "Syntax and Semantics of Object-Oriented Notations", "The Unified Modeling Language", "Pragmatic and Formal OO-Modeling Techniques", and "OO-Development Strategies and Process Models". This year we decided to choose "Object-Oriented Modeling of Embedded Realtime Systems" as our hot topic for the following reasons:

The importance of embedded realtime systems for our daily life is rapidly increasing. More or less unnoticed they fulfill the task to control the "behavior" of technical systems ranging from small coffee machines to big industrial complexes or autonomous vehicles on foreign planets. Important application areas may be found – among others – in the aerospace, automotive, chemical process engineering, and telecommunications industry. In all these areas the complexity of realized systems and their functions increases rapidly, and the software of these systems plays a more and more important role.

Until now the software of RT systems has mainly been developed using the structured analysis approach, i.e. software engineering technology of the eighties. As a consequence the developed software does not fulfill our expectations concerning maintenance and reusability of implemented subcomponents. Furthermore, the developed software often suffers from a strict separation of functionality and data. These are the reasons why object-oriented and/or component-oriented approaches are needed that facilitate the development of reusable components, thereby hiding functionality and data behind well-defined interfaces. The workshop OMER addresses all aspects of the development and application of object-oriented methods (languages, tools, processes) for the analysis, design, and implementation of embedded RT software. Its main purpose is to serve as a platform for academia and industry as well as for tool or method developers and users to exchange their experiences with OO development of embedded RT systems and to discuss new trends concerning e.g. the development of an UML standard extension for realtime modeling purposes.

First, OMER was planned to be a small German GI workshop with about 20 to 30 participants. But after a while we had to recognize that it was not feasible to keep the number of participants as small as intended. Furthermore, distinguished researchers from different countries were very interested to present their experiences and future activities in this area. Therefore, we had to switch

the workshop language (at least partially) from German to English and to rearrange many other things. We apologize for any inconveniences caused by these rearrangements.

The workshop proceedings contain a total of 28 short position papers written by authors from six different countries. About 12 of them are written by industrial professionals and about 16 by researchers from various universities and research institutes. Based on these papers of high quality we were able to organize six different sessions addressing topics such as “RT System Architectures”, “Unified Modeling Language Extensions”, and “RT Case Tools and Programming Languages”. All sessions have a length of 90 minutes with at most 60 minutes for short paper presentations and the remaining 30 minutes for an interesting discussion between the authors of the presented short papers and the other participants of the workshop.

Furthermore, we are especially honored that two distinguished researchers have accepted invitations to transmit their experiences in this area. David Harel is giving the first day’s invited talk “On the Behavioral Modeling of Complex Object-Oriented Systems” as an introduction to the whole workshop. Bran Selic is giving the second day’s invited talk about the hot topic “Using UML to Model Complex Real-Time Architectures”.

Finally, we were able to organize a panel discussion with very prominent developers of OO/RT formalisms, languages, and CASE tools as well as with managers responsible for the introduction of these techniques in industry. They are willing to share their insights about the future of object-oriented RT system modeling (standard) technologies in academia and industry.

The workshop and the position papers gathered here represent the work of many people. The online version of the workshop proceedings with links to the home pages of all authors can be found at

`http://inf2-www.informatik.unibw-muenchen.de/GROOM/OMER/`

We would like to express our sincere gratitude to the authors, the OMER Program Committee members (who had to review all submissions in less than one week), and all other participants of the workshop. Furthermore many thanks to the local staff of the “Tagungsstätte des Bayerischen Bauernverbandes” (the conference building of the Bavarian Farmers Association) and the Institute for Software Technology at the University of the German Federal Armed Forces for organizing all the details of the workshop program.

We hope you enjoy the workshop, have a pleasant stay in Herrsching am Ammersee, and find time to visit the famous monastery “Kloster Andechs”, which is (one of) the eldest Bavarian place of pilgrimage.

Peter P. Hofmann and Andy Schürr

The Program Co-Chairs

Invited Speakers

David Harel

David Harel is Dean of the Faculty of Mathematics and Computer Science at the Weizmann Institute of Science in Israel. He also co-founded I-Logix, Inc., and is an adjunct professor at the Open University in Israel. He is the inventor of the statecharts language and a co-designer of the Statemate and Rhapsody tools. His research interests include computability and complexity theory, logics of programs, theory of databases, computer science education, systems engineering and visual languages.

He has received several awards, among them ACM's 1992 Karlstrom Outstanding Educator award for his book "Algorithmics: The Spirit of Computing" (Addison-Wesley, 1987), the 1996 Stevens Award in software development methods, and the 1997 Israeli Prime Minister's Award. He is a Fellow of the ACM and of the IEEE.

Bran Selic

Bran Selic is the Vice President of Advanced Technology at ObjecTime Limited. He has over 25 years of commercial real-time software development and management experience (in telecommunications, aerospace, and robotics), including many years of experience with object-oriented design and programming. He is the principal author of the book, "Real-Time Object-Oriented Modeling" and was a member of the core team that defined the semantics of the Unified Modeling Language (UML).

Recently, Bran has been collaborating with Jim Rumbaugh and Grady Booch on the definition of a real-time extension to UML. He has given numerous invited talks and has lectured extensively on real-time and object-oriented technology at technical conferences and seminars.

On the Behavior of Complex Object-Oriented Systems

David Harel*

Over the years, the main approaches to high-level system modeling have been *structured-analysis* (SA), and *object-orientation* (OO). The two are about a decade apart in initial conception and evolution. SA started out in the late 1970's by De Marco, Yourdon and others, and is based on 'lifting' classical procedural programming concepts up to the modeling level and using diagrams [CY, D]. The result calls for modeling system structure by functional decomposition and the flow of information, depicted by hierarchical data-flow diagrams. As to system behavior, the mid 1980's saw several methodology teams (such as Ward/Mellor [WM], Hatley/Pirbhai [HP] and our own Statemate team [H$^+$]) making detailed recommendations enriching the basic SA model with means for capturing behavior based on state diagrams, or the richer language of statecharts [H]. A state diagram or statechart is associated with each function to describe its behavior. Carefully defined behavioral modeling, we should add, is especially crucial for embedded, reactive, and real-time systems. A detailed description of the way this is done in the SA framework appears in [HP]. The first tool to enable model executability and code synthesis of high-level models was Statemate, made commercially available in 1987 (see [H$^+$, IL]).

OO modeling started in the late 1980's. Here too, the basic idea for system structure was to 'lift' concepts from object-oriented programming up to the modeling level, and to do things with diagrams. Thus, the basic structural model for objects in Booch's method [B], in OMT [R$^+$], in the ROOM method [SGW], and in many others (e.g., [CD]), has notation for classes and instances, relationships and roles, and aggregation and inheritance. Visuality is achieved by basing this model on an enriched form of entity-relationship diagrams. As to system behavior, most OO modeling approaches, including those just listed, adopted the statecharts language for this. A statechart is associated with each class, and its role is to describe the behavior of the instance objects.

However, here there are subtle and complicated connections between structure and behavior, that do not show up in the simpler SA paradigm. Here classes represent dynamically changing collections of concrete objects, and behavioral modeling must address issues related to their creation and destruction, the delegation of messages, the modification and maintenance of relationships, aggregation, true inheritance, etc. These issues were treated by OO methodologists in a broad spectrum of degrees of detail — from vastly insufficient to adequate. The test, of course, is whether the languages for structure and behavior and their inter-links are defined sufficiently well to allow full model execution and code synthesis. This has been achieved only in a couple of cases, namely in the ObjecTime tool (based on

*The Weizmann Institute of Science, Rehovot, Israel, and I-Logix, Inc., Andover, MA

the ROOM method of [SGW]), and in the Rhapsody tool. Rhapsody (see [IL]) is based on the executable object modeling work presented in [HG], which was originally intended as a carefully worked out language set based on Booch and OMT object model diagrams driven by statecharts, and addressing the issues above in a way sufficient to lead to excutability and full code synthesis.

In a remarkable departure from the similarity in evolution between the SA and OO paradigms for system modeling, the last three years have seen OO methodologists working together. They have compared notes, have debated the issues, and have finally cooperated in formulating the UML, which was adopted in 1997 as a standard by the OMG (see [UML]). This sweeping effort, which in its teamwork is reminiscent of the Algol'60 and Ada efforts, has taken place under the auspices of Rational Corp., spearheaded by Booch, Rumbaugh and Jacobson. Version 0.8 of the UML was released in 1996 and was rather open-ended and vague, lacking in detail and well thought-out semantics. For about a year, the UML team went into overdrive, with a lot of help from methodologists and language designers from outside Rational Corp. Our team contributed quite a bit too, and the languages underlying Rhapsody [HG, IL] are indeed the executable kernel of the UML. The version of the UML adopted by the OMG is thus much tighter and more solid than version 0.8. With some more work there is a good chance that the UML will become not just an officially approved standard, but the main modeling mechanism for the software that is constructed according to the object-oriented doctrine. And this is no small matter, as more and more software engineers are now claiming that more and more kinds of software are best developed in an OO fashion.

The recent wave of popularity that the UML is enjoying will bring with it not only the UML books written by Rational Corp. authors (see, e.g., [RJB]), but a true flood of books, papers, reports, seminars, and tools, describing, utilizing, and elaborating upon the UML, or purporting to do so. Readers will have to be extra-careful in finding the really worthy trees in this forest. Despite this, one must remember that right now UML is a little *too* massive. We understand well only parts of it; the definition of other parts has yet to be carried out in sufficient depth as to make clear their relationships with the constructive core of UML (the class diagrams and the statecharts). Moreover, Thee are still major problems in the general area of behavioral specification and design of complex object-oriented systems that await treatment. These still require extensive research.

Here are brief discussions of two examples of research directions that seem to me to be extremely important. One has to do with message sequence charts (MSCs) and their relationship with state-based specification, and the other had to do with inheriting behavior.

As to the first one, there is a dire need for a highly expressive MSC language, with a clearly defined graphical syntax and a fully worked out formal semantics. Such a language is needed in order to construct semantically meaningful computerized tools for describing and analyzing use-cases and scenarios. It is also a prerequisite to a thorough investigation of what might be *the* problem in object-oriented specification: relating *inter*-object specification to *intra*-object specification. The former is what engineers will typically do in the early stages of behavioral modeling; namely, they come up with use-cases and the scenarios that capture them, specifying the inter-relationships between the processes and object instances in a

linear or quasi-linear fashion in terms of temporal progress. That is, they come up with the description of the scenarios, or 'stories' that the system will support, each one involving all the relevant instances. A language for scenarios is best used for this. The latter, on the other hand, is what we would like the final stages of behavioral modeling to end up with; namely, a full behavioral specification of each of the processes or object instances. That is, we want a complete description of the behavior of each of the instances under all possible conditions and in all possible 'stories'. For this, a state-machine language such as statecharts appears to be most useful. The reason the state-machine intra-object model is what we want as an output from the design stage is for implementation purposes: ultimately, the final software will consist of code for each process or object. These pieces of code, one for each process or object instance, must — together — support the scenarios as specified in the MSCs. Thus the "all relevant parts of stories for one object" descriptions must implement the "one story for all relevant objects" descriptions.

Now, there are several versions of MSC's, including the ITU standard (see [ITU]), and the UML also has a version of sequence diagrams as part of its languages. However, both versions are extremely weak in expressive power, b eing based essentially on simple constraints on the partial order of events. Nothing much can be specified about what the system will actually do when run. A particular troublesome issue is the need to be able to specify 'no-go' scenarios, ones that are not allowed to occur. In short, there is a serious need for a more powerful language for sequences. In a recent paper [DH], we have adressed this need, proposing an extension of MSCs, which we call *live sequence charts* (or *LSCs*). One of the main extensions deals with specifying "*liveness*", i.e., things that must occur. LSCs allow the distinction between possible and necessary behavior both globally, on the level of an entire chart, and locally, when specifying events, conditions and progress over time within a chart. (In so doing it makes possible the natural specification of forbidden behavior.) LSCs also support subcharts, synchronization, branching and iteration. It is far from clear whether this language is exactly what is needed: more work on it is required, experience in working with it, and of course an implementation. Nevertheless, it does make it possible to start looking seriously at the two-way relationship between the arformentioned dual views of behavioral description. How to address this *grand dichotomy of reactive behavior*, as we like to call it, is a major problem. For example, how can we synthesize a good first approximation of the statecharts from the LSCs? Finding efficient ways to do this would constitute a significant advance in the automation and reliability of system development. In very recent work, as of yet unpublished, we propose a first-cut at devising such synthesis algorithms and at analyzing their complexity [HKg].

The second direction of research involves inheriting behavior. Inheritance is one of the key topics in the object-oriented paradigm, but when working on the analysis and design levels (rather than in the programming stage) it is not at all clear what exactly it means for an object of type B to be also an object of the more general type A. In virtually all approaches to inheritance in the literature, the **is-a** relationship between classes A and B entails a basic minimal requirement of *protocol conformity*, or subtyping, which roughly means that it should be possible to 'plug in' a B wherever an A could have been used, by requiring that what can be requested of B is consistent with what can be requested of A. In addition, *structural conformity*, or subclassing, is often requested, to the effect that B's

internal structure, such as its set of composites and aggregates, is consistent with that of A.

Nevertheless, these form only weak kinds of subtyping, and they say little about the *behavioral conformity* of A and B. They require only that the plugging in be possible without causing incompatibility, but nothing is guaranteed about the way B will actually operate when it replaces A. Thus we don't have full behavioral substitutability, but merely a form of consistency. In fact, B's response to an event or an operation invocation might be totally different from A's. Here we are concerned with investigating the plausibility (and indeed also the very wisdom) of guaranteeing full behavioral conformity. In practice. behavioral conformity is often too stringent; many times one does not expect the inheritance relationship between A and B to mean that anything A can do B can do too and in the very same way. They are often satisfied with guaranteeing that anything A can do, B can be *asked* to do, and will look like it is doing, but it might do so differently and produce different results.

In recent work, also not yet published [HKp], we have obtained preliminary results that show that on a suitable schematic, propositional-like level of discourse there are strong connections between questions of inheritance and well-known semantic notions of refinement between specifications (such as trace containment and simulation). We also have several results about the computational complexity of detecting and enforcing behavioral conformity. However, here too there is still much research to be done, including the discovery of restrictions on behavioral specification that would guarantee behavioral conformity, and algorithms for finding out if given models satisfy such restrictions.

Many other significant challenges remain, for which only the surface has been scratched. Examples include true formal verification of software modeled using high-level visual formalisms, automatic eye-pleasing and structure-enhancing layout of the diagrams in such formalisms, satisfactory ways of dealing with hybrid object-oriented systems that involve discrete as well as continuous parts, and much more.

It is probably no great exaggeration to say that there is a lot more that we *don't* know and *can't* achieve yet in this business than what we do know and can achieve. Still, the efforts of scores of researchers, methodologists and language designers have resulted in a lot more than we could have hoped for ten years ago, and for this we should be thankful and humble.

References

[B] Booch, G., *Object-Oriented Analysis and Design, with Applications* (2nd edn.), Benjamin/Cummings, 1994.

[CY] Constantine, L. L., and E. Yourdon, *Structured Design*, Prentice-Hall, Englewood Cliffs, 1979.

[CD] Cook, S. and J. Daniels, *Designing Object Systems: Object-Oriented Modelling with Syntropy*, Prentice Hall, New York, 1994.

[DH] Damm, W., and D. Harel, "LSCs: Breathing Life into Message Sequence Charts", *Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-based Distributed Systems (FMOODS'99)*, (P. Ciancarini, A. Fantechi and R. Gorrieri, eds.), Kluwer Academic Publishers, 1999, pp. 293–312.

[D] DeMarco, T., *Structured Analysis and System Specification*, Yourdon Press, New York, 1978.

[H] Harel, D., "Statecharts: A Visual Formalism for Complex Systems", *Sci. Comput. Prog.* **8** (1987), 231–274. (Preliminary version appeared as Tech. Report CS84-05, The Weizmann Institute of Science, Rehovot, Israel, Feb. 1984.)

[HG] Harel, D., and E. Gery, "Executable Object Modeling with Statecharts", *Computer* (July 1997), 31–42. (Also, *Proc. 18th Int. Conf. Soft. Eng.*, Berlin, IEEE Press, March, 1996, pp. 246–257.)

[HKg] Harel, D., and H. Kugler, manuscript in preparation, 1999.

[HKp] Harel, D., and O. Kupferman, manuscript in preparation, 1999.

[H$^+$] Harel, D., H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems", *IEEE Trans. Soft. Eng.* **16** (1990), 403–414. (Preliminary version appeared in *Proc. 10th Int. Conf. Soft. Eng.*, IEEE Press, New York, 1988, pp. 396–406.)

[HP] Harel, D., and M. Politi, *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*, McGraw-Hill (258 pp.), 1998. (Early version titled *The Languages of STATEMATE*, Technical Report, I-Logix, Inc., Andover, MA, 1991.)

[HP] Hatley, D., and I. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House, New York, 1987.

[IL] I-Logix, Inc., products web page, http://www.ilogix.com/fs_prod.htm.

[ITU] *ITU-TS Recommendation Z*.120: *Message Sequence Chart* (*MSC*). ITU-TS, Geneva, 1996.

[RJB] Rumbaugh, J., I. Jacobson and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.

[R$^+$] Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[SGW] Selic, B., G. Gullekson and P. T. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons, New York, 1994.

[UML] Rational Corp., documents on the UML, http://www.rational.com/uml/index.jtmpl.

[WM] P. Ward and S. Mellor, *Structured Development for Real-Time Systems* (Vols. 1, 2, 3), Yourdon Press, New York, 1985.

MVC-Based Modeling Support for Embedded Real-Time Systems

– Position Statement –

Stefan Sauer Gregor Engels

University of Paderborn
Dept. of Computer Science
D 33095 Paderborn, Germany
{sauer | engels}@uni-paderborn.de

1. Introduction

Several software architecture paradigms have been proposed for the development of interactive software systems and applications. Their common idea is the separation of user interface components from application logic. Normally, they are deployed as design patterns or basic frameworks of interactive systems.

Deploying these architecture paradigms for multimedia systems or embedded real-time systems, first requires an appropriate adaption to the specific aspects of these application areas. We will compare within this position paper two well-known architecture paradigms and motivate an advanced architecture paradigm applicable to real-time, embedded multimedia systems. In addition, we will discuss that the architecture paradigm should be reflected within the structure of a modeling language.

2. Architectural Patterns for Interactive Systems

The Model-View-Controller paradigm (MVC) is an architectural model, widely known in object-oriented software development, for interactive applications that distinguishes a model component holding the core functionality and data, a view component for displaying information to the user, and a controller component for handling user inputs. A change-propagation mechanism ensures consistency between the model on the one hand and the two user interface components on the other hand (cf. [6, 1]). Another example of such an architectural model for interactive software systems is the Presentation-Abstraction-Control (PAC) approach (cf. [2, 1]). It consists of a hierarchy of cooperating agents each with specific functional responsibilities.

A comparison of these two architectural paradigms reveals that the abstraction aspect of PAC agents resembles the model component of MVC while its presentation aspect contains both perspectives of the view and the controller components of MVC (as in the Document-View approach known from many text editors). In addition, PAC agent's control aspect is responsible for the communication between its corresponding abstraction and presentation aspects as well as with other agents.

Summarizing, these two architectural patterns agree on the three aspects model, view, and controller, while the PAC paradigm identifies an additional explicit communication aspect.

3. Extending MVC

Architectural patterns for interactive software systems like the MVC model are normally deployed for modeling the software architecture during system design. We think that especially nowadays where multimedia user interfaces gain a more prominent role in software systems, aspects of user interface and application logic should already be considered and distinguished on the analysis level (see also [5]). We suggest to deploy the MVC paradigm to specify and structure the architecture of an analysis model of an integrated interactive application. Model, view, and controller then become more abstract concepts and form the basis for analysis model classes that can then be further refined in the design phase.

By modeling all these aspects, we enable consistency checks between them on the model level as well as we foster an integrated, comprising model of all application aspects and perspectives.

The main distinction between real-time systems and conventional applications is that for the former the timeliness of responses to events and of results of computations is essen-

tial for their correctness. Embedded systems are systems that contain a computer which reacts to the inputs from and controls other external (hardware) devices, e.g. sensors and actuators.

Therefore, we stress that communication between the model, view, and controller components deserves more attention. An explicit extension of the MVC paradigm by a fourth component for communication is necessary, as it is already reflected in the PAC paradigm. The communication aspect regards both management and control of communication (protocols), as within PAC agents, and the properties of the communication channel. This is especially important as many applications evolve towards distributed architectures, e.g. client/server or network-based systems. Furthermore, the aspect of communication is essential for embedded real-time applications with hard or soft time constraints on the signal delay between internal and to or from external components of the system.

In the MVC approach, communication is only specified by interrelationships and message sending between the other three components. This approach is sufficient for conventional (object-oriented) applications, but lacks an appropriate representation to support different forms of and media for communication. As it is realized in middleware approaches like CORBA or DCOM, modelers or programmers do not need a specific notion for modeling or implementing the middleware, they just use it.

So why bother? In distributed and especially mobile system architectures as well as multimedia and real-time applications, the characteristics and state of the communication medium play a significant role for the functioning of the system. For example, communication channels exist between sensors, actuators, and different processing units, but also between a computer system and its user for both input and output of data. Questions that need to be taken into account are, beyond others, what amount of data can be transmitted via a specific communication channel (data rate), or what types of data have what time constraints.

Therefore, we claim that the communication components should be modeled as first-order objects and thus extend the MVC paradigm to a *Model-View-Controller-Communication (MVCC)* paradigm.

4. Incorporating MVCC into Modeling Languages

Two alternatives for supporting the modeling task at the analysis level according to the MVCC paradigm can be distinguished:

framework-based modeling, i.e. refinement on the modeling level and specialization of model classes (of the framework),

metamodeling, i.e. refinement of the metamodel and specialization (stereotyping) of its classes for the domain to be modeled.

In the first approach, the model of the system is based on an MVCC framework containing abstract classes for the architectural components. In the second approach, a specialized language is developed and, thus, explicit language support can be offered to the modeler.

In detail, in the process of designing such a (visual) specialized modeling language the following tasks need to be accomplished:

1.) Identify the aspects of the application that need to be modeled, e.g. model, view, controller, and communication aspects as well as domain specific aspects like, e.g. for real-time systems, time representation, timing constraints, concurrency, and synchronization.

2.) Define the language concepts and elements as well as diagram types to model these aspects, and decide which aspects are modeled by which language features.

3.) Specify syntax and context-sensitive constraints of the language by metamodeling and eventually define the dynamic semantics of the language and its full pragmatics.

Following this approach, one defines packages or collaborating classes in the metamodel that represent the four architectural aspects of MVCC. In addition to identifying these packages, their contained classes, and the associations among the latter, basic metamodel classes like Class need to be specialized to get metamodel classes for ViewClass, ModelClass, ControllerClass, etc. Then, we can define a mapping between the architectural aspects and existing basic as well as added or refined metamodel classes.

As we state in our main thesis, we long for direct support of the system's main architectural components model, view, control, and communication by the modeling language. This means language incorporation of these aspects. This approach has several advantages:

it offers adequate, intuitive, and specialized language features for the problem domain,

a mapping of model elements to aspects specified in different diagram types can be given,

force is put on application modelers to follow the paradigm ensuring its usage.

By migrating the architectural paradigm towards the metamodel level, one can also build a reference model for analyzing and comparing different modeling languages regarding which aspects of the domain under consideration

are actually represented. Thus, different languages populate different entities of the metamodel that can comparatively be evaluated then.

5. Integrate General Modeling Language with Domain-Specific Language

We expect multimedia, interactive, and embedded real-time systems to become both an important and a challenging class of future applications. To be able to model such a complex type of system with all its aspects, requirements, and constraints, a modeling language is needed that supports both the manifold aspects and the details of any of them. Our idea of solving this problem is to integrate appropriate aspect or domain-specific language features and base them on a common concept and model. We consider the object-oriented modeling standard UML (see [7, 8]) being capable of playing the role of the general basis for integration. It contains already many established and promising diagrammatic languages and modeling elements, and its extension mechanisms offer means to add special language features. In addition to stereotyping of metamodel classes, actually changing the metamodel by adding or changing classes would further support the approach. Refinements of syntax, semantics, and pragmatics of the above introduced general modeling language are needed for the incorporation of domain-specific aspects using metamodeling, context-sensitive dependencies and constraints.

When integrating the architectural MVCC paradigm with the metamodel, the components of model, view, and controller should be identifiable and separately accesible within the metamodel. In this context, a profound solution requires an analysis of shortcomings of the current UML metamodel. Additional work needs to be done on the interrelations between pragmatics of the modeling language and the metamodel:

1.) pragmatics refine the metamodel,

2.) domain-specific modeling further requires refinement of pragmatics and the metamodel of a multi-purpose modeling language,

3.) some aspects of pragmatics can be captured within the metamodel or in context-sensitive constraints regarding it, others need to be specified elsewhere, e.g. the time when a context-sensitive constraint is checked.

6. Experience: OMMMA-L and SOCCA

We have developed a UML-based modeling language for interactive multimedia applications called OMMMA-L (Object-oriented Modeling of MultiMedia Applications - the Language, see [9])). To model the diverse multimedia characteristics and application aspects, we extended the MVC paradigm towards a specialized model for multimedia (communication still needs to be added). Within the model, a distinction between the static application model, consisting of the application logic and a hierarchy of discrete and continuous media types, and the dynamic behavior of the model has been made. Refinements of the other components consider multimodality in the presentation of information and in the control component.

OMMMA-L adds specific concepts for modeling e.g. timing behavior and synchronization as well as presentation layout to standard UML and constrains the usage of notational elements. OMMMA-L comprises class diagrams, state diagrams, extended sequence diagrams and a newly introduced presentation layout diagram. Each of them is devoted to model a specific aspect of the application.

The extended language features have been added to UML by refining its metamodel by stereotyping and extension resulting in devoted meta model classes for each aspect.

In another project called SOCCA [3, 4], the model and the control components of an application have been supplemented by a communication component to facilitate the specification of different communication types and architectures.

7. Perspective for Embedded Real-Time Applications

We consider the presented approach to be successfully applicable to complex embedded real-time applications.

Many real-time aspects can be found in multimedia applications. Multimedia applications have real-time requirements and most of them are also embedded systems containing hardware devices to play or record media data. Data of continuous media types need to be received, processed, transmitted, and presented under real-time constraints. Video or audio streams consist of a sequence of logical data units that need to be presented within specific timing bounds and perceivable fluctuations of presentation speed should be avoided.

Furthermore, as in other real-time applications, external events to the system need to be handled within a predefined time interval. Most multimedia tasks dealing with continuous data streams do not only require an upper bound for the duration of this interval (as in synchronous communication), they also need a lower bound to limit the storage requirements for buffering (as in isochronous communication), especially in cases of distributed systems with varying network latencies.

To refine MVC to an analysis needs to be undertaken to find the aspects that need to be considered. Def-

initely, timeliness as well as hard and soft time constraints need an appropriate representation on the integrated model level. Effects of real-time requirements on model, view, controller and communication components have to be identified, e.g. in the scenario for signalling a warning message to and handling it by the user or in the case of exception handling. It has to be answered, which time constraints apply to each component.

References

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: *A System of Patterns: Pattern-Oriented Software Architecture*. Chichester: Wiley 1996

[2] J. Coutaz: PAC: An object-oriented model for dialog design. In: H.-J. Bullinger, B. Shackel (eds.): *Proc. Human Computer Interaction - INTERACT'97*, Stuttgart. Elsevier 1997, pp. 431-436

[3] G. Engels, L. P. J. Groenewegen, G. Kappel: Object-oriented specification of coordinated collaboration. In N. Terashima, Ed. Altman (eds.): *Proc. IFIP World Conference on IT Tools*, September 2-6, 1996, Canberra, Australia. London: Chapman & Hall 1996, pp. 437-449

[4] G. Engels, L. P. J. Groenewegen, G. Kappel: Coordinated collaboration of objects. In M. Papazoglou, St. Spaccapietra, Z. Tari (eds.): *Object-Oriented Data Modelling Themes*. Cambridge MA: MIT Press 1999. To appear

[5] M. Fowler: *Analysis Patterns: Reusable Object Models*. Menlo Park CA: Addison-Wesley 1997

[6] G. E. Krasner, S. T. Pope: A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, Vol. **1**(3) (Aug./Sept. 1988):26-49

[7] Rational Software et al. UML Notation Guide, UML Semantics, version 1.1. Rational, 1997. URL:http://www.rational.com/uml

[8] J. Rumbaugh, I. Jacobson, G. Booch: *The Unified Modeling Language Reference Manual*. Reading MA: Addison-Wesley 1999

[9] S. Sauer, G. Engels: UML-based modeling of multimedia applications. In: J. Desel, K. Pohl, A. Schürr (eds.): *Modellierung'99*, Karlsruhe, March 10-12, 1999, Stuttgart: Teubner 1999, pp. 155-170 (in German)

A different notion of component

position paper for OMER'99

Harald Störrle
Ludwig-Maximilians-Universität München
Oettingenstr. 67, 80538 München, Germany
++49-89-2178-2134
stoerrle@informatik.uni-muenchen.de

May 7, 1999

1 Introduction

Component based software development (CBSD) can greatly improve the quality and effectiveness of the development of software. This is particularly important for control software for embedded and real-time Systems (ERTSes), that are particularly sensitive to malfunctions and, once deployed, particularly expensive to modify/correct. So, unsurprisingly, methodological approaches with an emphasis on these application areas have long since been occupied with aspects of component technology. For instance, HOOD and ROOM (aka. UML/RT) provide abstractions similar to the notion of component as understood in classical software development today (i.e. subsystems, actors and capsules).

So far, however, little attention has been paid to the interactions of the development *process* and the notations available. Current component technologies define components basically as executable code only. Many (software) engineering activities, however, also require other types of information on a piece of software, that is to be reused: code reuse is only one possibility for reuse. In fact, empirical studies strongly suggest, that it is far superior to reuse documents from earlier phases of software development (see the empirical data quoted in [12, p. 83]).

Therefore, a different notion of component is proposed. Some examples are given where a CBSD-process differs from an ordinary software development process, and which informations would have to be included in the notion of component in order to support them.

2 A critique of today's component technology

Embedded systems pervade manufacturing and infrastructure. Thus, the failure of an ERTS is potentially disastrous, and so the quality requirements for the development of ERTSes are much higher than those for desktop software. Quality, however, is expensive and takes time. Components (i.e. architecture-level entities) have been proposed as a solution to this problem (see [12] on all aspects of reuse).

- Being used many times, the cost of thorough testing or even formal verification is shared among all users, and so becomes more affordable.
- Using prefabricated parts should reduce the amount of code to be written, and thus hopefully speed up development.

Current component-technologies such as Java Beans or COM, however, fail to satisfy the expectations raised. Shortcomings can be identified in three areas:

- **too small:** the abstractions they provide are too fine grained, they are on the class- or package level rather than on the subsystem level. Also, no abstractions are provided for larger levels of abstraction (see [13]).
- **too late:** they are concerned with the implementation phase of SW development only. A Bean or COM-component contains the information necessary to program with

it, but not the information needed to use it in the analysis and design. However, reuse is applicable already to the documents of the analysis and design phases. And it is evident that components can be used effectively only their usage is planned early on in the development. This is already partially true for today's components, but it is inevitable for larger components (subsystem level or above). Since many authors rightfully stress that components are supposed to be self-contained entities, they should be self-contained in all phases of development, including the early ones.

- **too soft:** They do not only provide not enough information, but the little information they do provide is also too soft: anything beyond the signature of a component is an unstructured natural language comment, at best. Thus, it can not be used for semantic based tool support (such as focused retrieval and checking for compatible communication behavior).

Beyond these three issues are aspects of a larger and even more severe shortcoming, the lack of a process for CBSD. Obviously, a CBSD-process is in many respects different from a conventional development process:[1]

- Apart from the desired product, the development process also produces new components, change requirements (bug fixes, improvements, etc.) for existing components and requests for new components.
- Existing components must be considered very early on in the development, i.e. there must be a search-and-evaluate-activity in the design phase, possibly already in the analysis or even earlier.
- To be practical, the search and evaluation of components has to be done with extensive machine support. Thus, the functionality, dependencies and interfaces of a component have to be described in a way that can be exploited by tools.

Today's component technologies, however, are not intended to support these activities: they are programming components rather than software engineering components. Therefore, in this paper a new kind of components is proposed to improve in this area. For want of a better name they are called Software Engineering Components (SEC). In [2] a somewhat similar yet less powerful notion for the small to medium levels of abstraction has been proposed under the name of component package.

3 Software Engineering Components

A SEC consists of several *views*, featuring information for different development activities. Which views are present will depend on the precise purpose they are put to, and the defining characteristics one accepts for components, but in general one would expect at least the following:

- the code view represents the executable code for the component;
- a description of the functionality of the component, relevant system boundaries, and intended usage;
- the architectural representation of a component, i.e. its behavior (if it is elementary) or its internal structure, its interfaces (signature and behavior), and dependencies on other components.

For embedded systems, one might also have a deployment view, that determines what part of the functionality (i.e. which components) is to be implemented as hardware, and what part is to be implemented as software. Other possible views might be: a samples view (e.g. pointers to sample applications with the component, experience reports), a performance view (e.g. load figures relative to number of users and so on), a test view (e.g. test cases, partitions of input data, error density and so on). The notations for expressing the views are not necessarily fixed, but standard techniques such as UML probably are a good choice. Thus, a primitive realization of SECs themselves might simply be

[1] Some first steps towards development process and methodologies for component based systems have been made in [3] (focus on organizational issues), [4] (proposing a fractal process model) and [2] (focus on practical questions and "process patterns"). Similar work has been done in software evolution (see [6]), a term coined by L.J. Arthur. It is currently not clear, however, how much of this prior work can be adapted to the case of component based software development.

a stereotyped UML-package diagram (see Figure 1), with a fixed set of views with (a set of) appropriate diagram types or tagged values. The individual views in turn might be implemented as follows:

- The code view would contain a set of executables for different types of environments (choices of hardware, operating system, windowing systems, class libraries and so on). Also, different revisions could be provided in order to guarantee backward compatibility. For some types of license agreements, the executables might be supplements with (excerpts of) the source code, too. The code view would also document dependencies to external software, such as operating system calls.
- The functionality view would be expressed as a use case diagram, some role and use case descriptions, and comments about intended usage (possibly supplemented by pointers to previous usages of the component). While informal, this description is not unstructured (see below).
- The architecture view is probably the most valuable view of all. The notation of the ROOM method (see [7, 5, 8] called architecture diagrams in the remainder), has been used successfully for similar purpose in the past. A ROOM-capsule is thought to represent a component, ROOM-layers are not used yet. In [9, 11, 10][2] a Petri-net semantics of UML/RT and some analysis algorithms have been proposed.

Depending on the intended application, other or modified views might be useful, too. In the context of real-time applications, information on temporal behavior (e.g. response times etc.) might be added, in the context of embedded systems, the distribution of functionality on hardware or software might be worth including.

4 Some SE-activities supported by SECs

In this section, it is sketched how the additional information of SECs might be used for some different non-programming Software Engineering activities (see [1] for some hints of other activities requiring similar support).

[2] All of these papers are currently submitted, but preprint versions may be obtained from the author. They will also appear as technical reports later on.

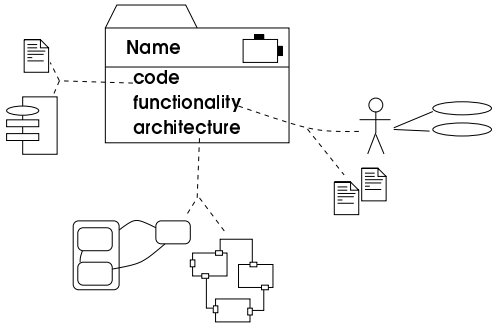


Figure 1: Template of a SE component.

4.1 Design debugging

One particularly valuable potential of SECs is the possibility of *design debugging*, i.e. the simulation and analysis of designs (rather than code). Obviously, this requires a formal semantics, appropriate analysis techniques and suitable tools to go along with them. The degree of elaboration and completeness of the semantics directly depends on the precise type of analysis to be done. Among other features, they allow to check the following properties:

- Do the interfaces of two capsules' ports fit together, not just by the sets of signals they wish to exchange, but by the protocols they use?
- Does an interaction diagram really represent a sample run of a protocol?
- Is there a run where a given protocol will deadlock? Which one?
- Apart from the protocols, are there other circumstances (arguments, schedulings) where a system of components reaches an unwanted state (deadlock, livelock etc.)?

Of course, verifying these properties covers only logical errors, but these are the most expensive and difficult anyway. Also, it is clear, that for many systems properties as those mentioned are not (efficiently) decidable, so that user interaction (e.g. introduction of abstractions) is necessary. Yet, a set of useful properties can often be checked for automatically, and further additions

to this set can be expected from ongoing work. Two possible applications of design debugging techniques are sketched below.

4.2 Restructuring

Another activity is facilitated that is typical for CBSD: the *restructuring* of systems. This requires suitable equivalences or implementation relations on components to determine whether one component can be replaced by another. Consider the following example: an architectural design of a distributed system that has to be adapted to meet some load requirements. In distributed systems, response times and throughput often critically depends on the the distribution, i.e. what is done where, and how much network traffic is involved with a particular distribution. So suppose the desired speedup is to be achieved by a new distribution. As communication across distribution boundaries is asynchronous while communication within one machine can be considered to be synchronous. Thus, a new distribution of the same components will probably introduce new distribution boundaries, and thus turn some synchronous communications into asynchronous ones, and vice versa. The logical effect of this change can be examined based on the formal semantics of architecture diagrams.

4.3 Evaluation of components

Using components requires the evaluation of components at some point. In the previous sections, some formal analysis methods have been hinted at, that can help decide whether or not a component is indeed applicable in some given context. However, there is more to a component's evaluation: it also relies on non-functional aspects such as reliability, conformance to legal requirements, and temporal properties. These could be supported by including in a component experience reports, legal assessments and empirical measurements of the timing, respectively.

4.4 Support for classification and retrieval

Before one can use a prefabricated component, it has to be retrieved first. There has been a lot of work on software reuse that can be applied here (see for instance [12]), though in the reuse-world, little or no difference is made between the different types of documents, and their semantics is not exploited in retrieval. Additional cues for classification and search can be obtained from the fact that SECs are *structured bundles* of documents rather than single documents. So, the views (and individual items in them) can act as a kind of classification facets, reducing the number of items precisely matching a query (i.e. increasing *search precision*). Thus, a greater number of rough matches may be accepted (i.e. increased *search recall*), and so the overall search quality (ratio between found matches and possible matches) can be improved. Put in a nutshell: A search guided predefined categories such as a role is better focused and yields higher retrieval precision than an unguided search.

5 Further work

We are currently investigating, what impact on the development process the activities described above will have, and what other activities call for support by semantics, algorithms and tools. As a support for case studies in this area, a prototype implementation is currently being prepared by some students at the chair of Prof. Wirsing, Univ. München.

References

[1] Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig. A componentware development methodology based on process patterns. In *Proceedings of the 5th Annual Conference on the Pattern Languages of Programs, (PLOP)*, 1998.

[2] Desmond F. D'Souza and Alan Cameron Wills. *Objects, Components and Frameworks with UML. The Catalysis Approach.* Addison-Wesley, 1999.

[3] Adele Goldberg. A reuse business model. *Software Concepts & Tools*, 19(1):11–13, 1998.

[4] Wolfgang Hesse. Baustein-orientiert statt Phasen-zentriert: Neue Entwicklungsmethoden erfordern neuartige Vorgehensmodelle. In *Workshop Anwendungen von objektorientierten Entwicklungsstrategien und*

deren Unterstützung durch Vorgehensmodelle, 1998. Published in Rundbrief 2/1998 of the Fachausschuß 5.1 of the Gesellschaft für Informatik.

[5] Andrew Lyons. UML for Real-Time (Overview). Technical report, ObjecTime INC., see www.objectime.com, 1998.

[6] Robert Moreton. *A process model for software maintenance*, pages 29–33. IEEE Computer Society Press, 1996.

[7] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real Time Object Oriented Modeling*. Wiley & Sons, Inc., 1994.

[8] Bran Selic and James Rumbaugh. Die Verwendung der UML für die Modellierung komplexer Echtzeitsysteme. *OBJEKTspektrum*, (4):24–36, 1998. also published in English as: *Using UML for Modelling Complex Real-Time Systems*, technical whitepaper of Rational Software Corp. & ObjecTime Ltd.

[9] Harald Störrle. Analyzing behavioural designs. submitted to ESEC, 1999.

[10] Harald Störrle. Towards a denotational semantics of dynamic aspect of UML/RT. submitted to UML, 1999.

[11] Harald Störrle. Towards a semantic for sequence diagrams. to appear in Proc. 9. GI/ITG-Fachgespräch formale Beschreibungstechniken für verteilte Systeme, 1999.

[12] Andreas Zendler. *Konzepte, Erfahrungen und Werkzeuge zur Software-Wiederverwendung.* Tectum Verlag, Reihe Softwaretechnik, Nr. 1, 1997. Schriftenreihe des FAST.

[13] Andreas Zendler. Foundation of an taxonomic object system. *Information and System Sciences*, (40):475–492, 1998.

Steuergeräteentwurf auf Basis der ESCAPE-Architektur

Stefan Leboch, Michael Ryba, Florian Wohlgemuth

Universität Stuttgart
Institut für Parallele und Verteilte
Höchstleistungsrechner (IPVR)
Integrierter Systementwurf
Breitwiesenstraße 20 - 22
D-70565 Stuttgart

Stefan.Leboch@informatik.uni-stuttgart.de
Michael.Ryba@informatik.uni-stuttgart.de

DaimlerChrysler AG
Forschung und Technologie
T721
D-70546 Stuttgart

Florian.Wohlgemuth@daimlerchrysler.com

1. Einführung

Vermehrt wird im Kraftfahrzeugbau Angebotsdifferenzierung und zusätzlicher Kundennutzen über Ausstattungsmerkmale verwirklicht, die nur durch den Einsatz von elektronischen Bauteilen und damit durch Software erreicht werden können. Die Softwarekomplexität der Funktionalität eines Fahrzeuges steigt daher stetig an.

Im Bereich der Software-Entwicklung für administrative und kommerzielle Anwendungen wurden ähnliche Komplexitätszuwächse beobachtet. Hier konnte durch den Einsatz objektorientierter Technologien die Beherrschbarkeitsgrenze weiter hinausgeschoben werden. Es ist daher konsequent, auch für die Erstellung von Software für eingebettete Systeme die Eignung objektorientierter Technologien zu prüfen und gegebenenfalls notwendige Anpassungen zu erarbeiten.

Bei der objektorientierten Modellierung wird ein System anhand seiner Daten partitioniert, wobei die Datenstrukturen zusammen mit Zugriffs- und Manipulationsmethoden in Klassen gekapselt sind, welche zur Laufzeit Objekte instanziieren – d.h. die objektorientierte Modellierung orientiert sich im wesentlichen an den Daten. Bei eingebetteten Systemen im Automobilbereich liegen typischerweise keine komplexen Datenstrukturen vor. Statt dessen herrschen in diesem Bereich kontrollastige Anwendungen stark vor, d.h. es gibt nur wenige Anhaltspunkte für eine auf den Datenstrukturen basierende objektorientierte Modellierung.

2. ESCAPE-Architektur

Aufgrund ähnlicher Probleme wurde im Forschungsprojekt ESCAPE (Easy Software Composition with Autonomeous Patternbased Entities) eine Methode für die Entwicklung von EDA-Werkzeugen entwickelt, bei der die Aspekte Wiederverwendbarkeit und Wartbarkeit im Vordergrund standen. Entwurfswerkzeuge implementieren verschiedene Entwurfsoperationen, welche alle auf einer zentralen, oft komplexen Datenstruktur arbeiten. Die übliche objektorientierte Zerlegung auf der Basis der Datenstrukturen führt zu unbefriedigenden Ergebnissen – es entstehen sehr große, unübersichtliche Klassen (sog. CISC-Klassen) und die Kopplung zwischen den einzelnen Entwurfsoperationen ist sehr hoch [Lebo 98].

Die Strategie von ESCAPE führt zu einer Trennung von Daten und Methoden (ähnlich der Strukturierten Programmierung), ohne jedoch die Grundideen der Objektorientierung, wie Kapselung und explizite Schnittstellen (Information Hiding) und Vererbung aufzugeben. Hierzu werden die Daten mit den notwendigen Grundoperationen zu deren Manipulation in sogenannte RISC-Klassen gepackt und jede Methode in eine Dienstklasse herausfaktorisiert (siehe Abbildung 1), wobei die Dienste ausschließlich über die Grundoperationen auf die Daten einer RISC-Klasse zugreifen.

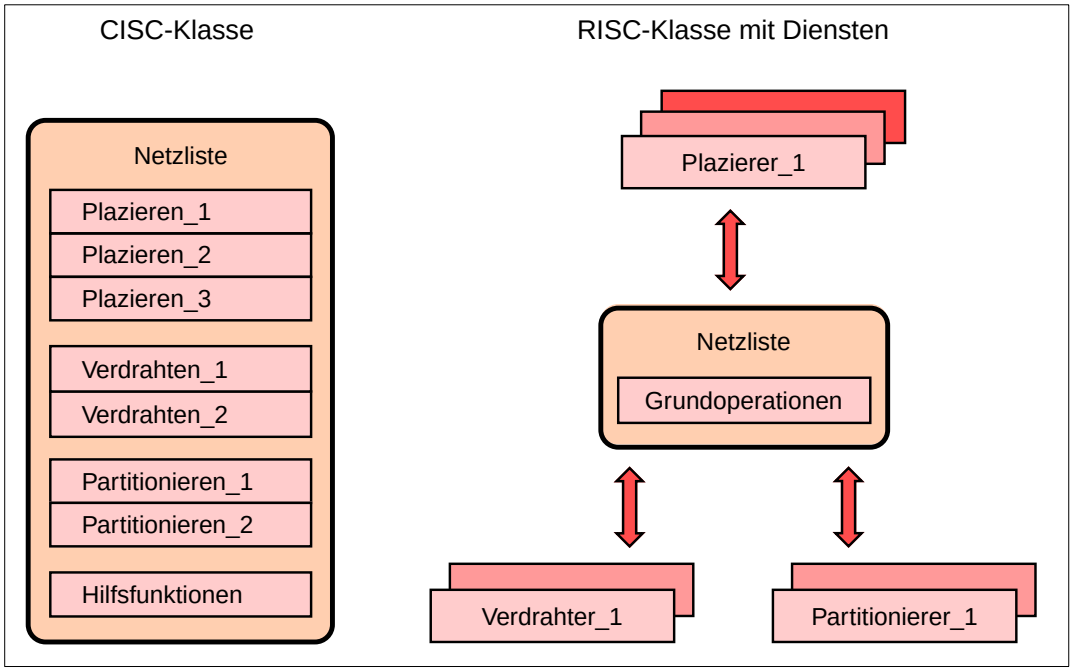


Abbildung 1: Unterschied zwischen CISC- und RISC-Klassen

Der geschilderte Ansatz wurde auf die Modellierung von eingebetteten Systemen übertragen (ESCAPE / RT), wobei hier die RISC-Klassen als Hardware-Wrapper [Awad96] dienen, d. h. sie stellen eine Schnittstelle zur Umgebung des eingebetteten Systems dar.

Abbildung 2 zeigt den Aufbau eines nach der oben erläuterten Architektur beschriebenen Steuergeräts und wie es in seine Umgebung eingebettet ist. Das Steuergerät erhält Informationen aus seiner Umgebung (z.B. durch Sensoren, Schalter, ...) und produziert daraus Ereignisse. Die Ereignisse steuern Dienste, welche die RISC-Klassen verwenden. Die RISC-Klassen wiederum geben bestimmte Informationen (Signale, Werte, ...) an die Umgebung zurück und steuern die Aktoren.

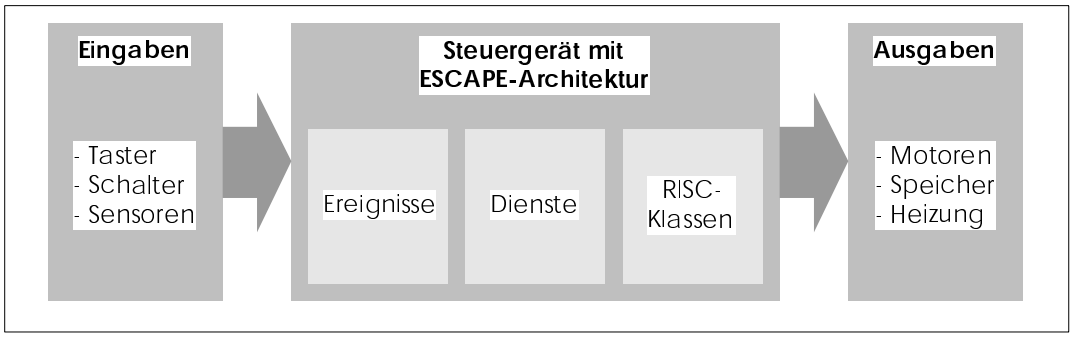


Abbildung 2: Prinzipielle ESCAPE / RT-Architektur

Die Kommunikation im Steuergerät erfolgt weitgehend asynchron und wurde deshalb ereignisbasiert modelliert. Der somit resultierende Aufbau eines Steuerdienstes wird am Beispiel der Steuerung für die Sitzheizung in Abbildung 3 dargestellt.

Auf der linken Seite der Abbildung sind die Komponenten dargestellt, mit denen der Benutzer direkt interagiert. Auf der rechten Seite sind Aktoren und weitere Dienste des Sitzsteuergerätes abgebildet. Am unteren Bildrand befinden sich Ereignisquellen, die von anderen Steuergeräten geliefert werden.

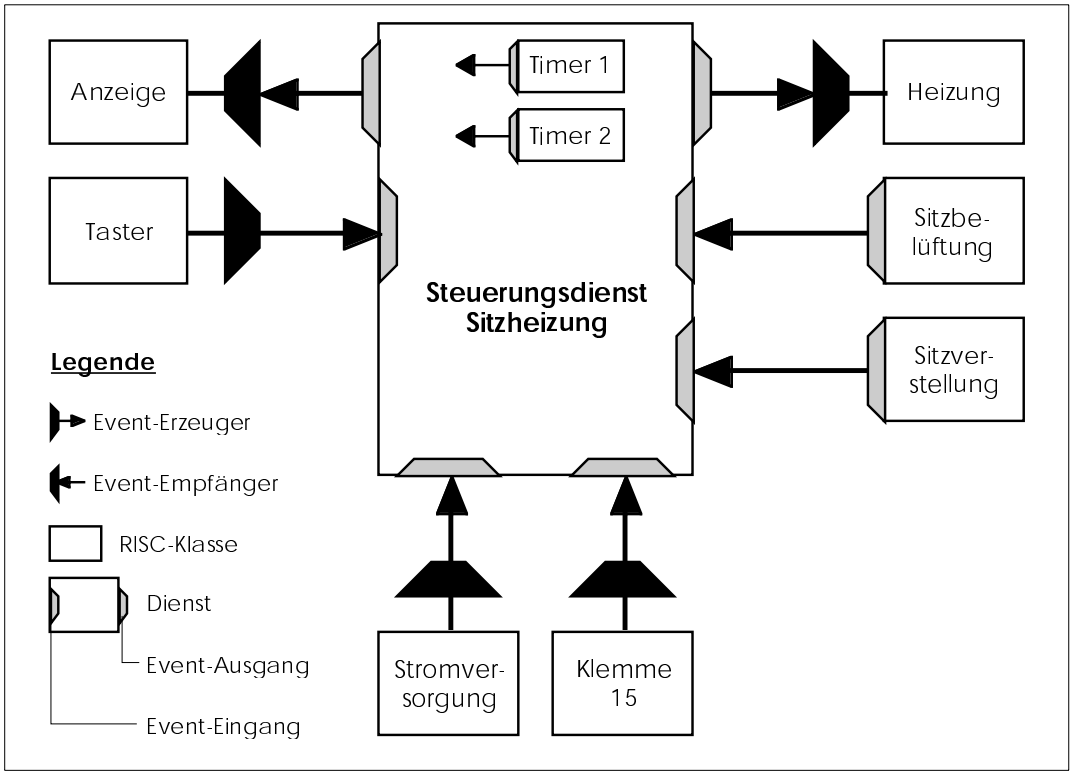


Abbildung 3: Das Steuergerät und seine Umgebung

Die Dienste können Ereignisse direkt erzeugen und verarbeiten; für die Kommunikation mit der Umgebung befinden sich an der Schnittstelle zwischen Steuergerät und seiner Umgebung sogenannte Event-Wrapper, welche eine Umwandlung von kontinuierlichen Signalen bzw. von bestimmten Werten in Ereignisse und umgekehrt vornehmen (siehe Abbildung 4).

```
behaviour Taster_Event_Erzeuger
  reads Taster

variable
  state: 0..1 initial 0;

do
  each 100ms do
    if Taster.read_val <> state then
      state := not state;
      case state of
        0: signal Release_Taster;
        1: signal Press_Taster;
      end case
    end if
  end each
end
```

Abbildung 4: Eventwrapper für einen Taster (Pseudocode)

Abbildung 5 zeigt einen Auszug aus dem Steuerungsdienst für die Sitzheizung. Dieser Auszug modelliert die Reaktion der Sitzheizungssteuerung auf Veränderungen der Zündschloßstellung, da in der modellierten (fiktiven) Sitzsteuerung nach dem Ausschalten der Zündung die

Sitzheizung zwar abgeschaltet wird, bei einem kurzen Fahrzeugstop den Betrieb in der bisherigen Leistungsstufe jedoch wieder aufnimmt.

```
behaviour Heizung is
  reads klemme_15, ...
  writes Heizung, Diagnose...

declare
  state, save_state : 0 .. 2;

local services
  timer1, timer2 : Timer;

initial
  state:=0;
  save_state := 0;
  ...
end initial

...

on klemme_15.ein
  if (state > 0) then
    signal Heizung.stufe_0
    save_state := state;
    state := 0;
    timer2.start(5);
  end if
end -- klemme_15.ein

on klemme_15.aus
  if (save_state > 0) then
    state := save_state;
    save_state := 0;
    case state of
      0: signal Diagnose.heizungsfehler
      1: signal Heizung.stufe_1
      2: signal Heizung.stufe_2
    end case
  end if
end -- klemme_15.aus

on timer2 do
  timer1.reset;
  save_status := 0;
end --timer2

...

end behaviour
```

Abbildung 5: Auszug Steuerungsdienst Sitzheizung (Pseudocode)

Da u. U. mehrere Dienste auf dasselbe Ereignisse reagieren, muß eine entsprechende Kommunikationsinfrastruktur eingerichtet werden. Prinzipiell wäre eine Broadcast-Lösung möglich, bei der alle Ereignisse an alle Empfänger versendet werden. Allerdings führt eine solche Lösung schon sehr bald zum Verlust des Überblicks und begünstigt ungewollte Seiteneffekte,

was Wiederverwendung und Wartbarkeit stark behindert bzw. einschränkt. Deshalb wurden als Kommunikationsmittel Ereigniskanäle vorgesehen, d.h. eine Multicast-Lösung mit statisch typisierten Ereignissen.

Dienste können sich für den Empfang und die Weitergabe von Ereignissen entweder bei anderen Diensten oder bei den Event-Wrappern anmelden und erhalten dann beim Auftreten der abonnierten Ereignisse eine entsprechende Benachrichtigung.

Das gewünschte Verhalten wird durch die Gesamtheit aller Dienste und die Grundoperationen der RISC-Klassen beschrieben. Die Zuordnung der einzelnen Dienste auf potentielle Steuergeräte ist nicht statisch festgelegt. Durch die stärkere Entkopplung der Dienste voneinander sowie zwischen den Diensten und den RISC-Klassen kann eine flexible Targetierung auf mehrere Steuergeräte vorgenommen werden.

3. Offene Fragen

Auf Basis der hier vorgestellten Architektur wurde die Funktionalität einer Sitzsteuerung prototypisch modelliert. Einige der dabei gewonnenen Erkenntnisse sind bereits in dieses Papier eingeflossen. Trotzdem sind noch einige Fragen zu klären, die im folgenden kurz geschildert werden:

- Was ist eine gute Größe für eine RISC-Klasse?
- Wie können Dienste optimal entkoppelt werden? Wie können zur Erleichterung der Wiederverwendung die Dienste so beschrieben werden, daß bei Spezialisierungen (Vererbung) nur die betroffenen Methoden angepaßt werden müssen?
- Wie können Dienste zu einem mächtigeren Dienst zusammengefaßt werden?
- Wie kann der getypte Multicast umgesetzt werden? Wie sieht dabei der Zusammenhang zwischen Ereignis und Methodenaufruf aus? Wie können Abarbeitungsfristen beschrieben und garantiert werden?
- Wie können Probleme bei der Vererbung von Automaten vermieden werden?
- Wie sieht eine geeignete Werkzeugumgebung aus?

Die Klärung dieser Fragen ist Gegenstand aktueller Forschungsarbeiten.

4. Literatur

[Awad96] M. Awad, J. Kuusela, J. Ziegler: *Object-Oriented Technology for Real-Time Systems: a Practical Approach Using OMT and Fusion;* Prentice-Hall, 1996

[Lebo 98] S. Leboch: *Wiederverwendung auf der Basis von Anwendungsdiensten*; Proc. 5 Berichtskolloqium des Graduiertenkollegs GKPVS, Stuttgart Juli 1998

Reflections on the Object-Oriented Design of Embedded Systems

Antje von Knethen, AG Software Engineering, University of Kaiserslautern,
D-67653 Kaiserslautern, Germany, vknethen@informatik.uni-kl.de,
Barbara Paech, Fraunhofer Institute for Experimental Software Engineering (IESE)
Sauerwiesen 6, D-67661 Kaiserslautern, Germany, paech@iese.fhg.de

Abstract

The methodical development of object-oriented embedded systems is still an open issue. We discuss the design of complex control functionality through object interactions. Borrowing from the domain of information systems, we argue for the use of independent control classes.

1 Introduction

While the object-oriented paradigm is widely accepted in the domain of information systems, the object-oriented modeling of embedded systems is still in its infancy. There are only few established object-oriented methods and they leave many unresolved questions. This holds particularly for the question how a complex system functionality can be distributed among objects.

In the domain of information systems, the introduction of independent function objects has proved to be an answer to this question [Jac92, Pae99]. Function objects control the modification of data in different objects. In the following sections, we argue for transferring this idea to embedded systems, where instead of data modifications complex dependencies among sensors and actuators are controlled.

We discuss our ideas based on a case study taken from the domain of building automation.

2 The Case Study: Temperature Control Within a Building

The case study was developed with support of the SFB 501 "Development of large systems with generic methods" at the University of Kaiserslautern.

The following list describes a subset of the requirements for a temperature control system within a building:

- If a person enters an empty room, the room has to be heated-up to an adjustable comfort temperature in an adjustable time slot.
- If the last person leaves a room, the comfort temperature has to be maintained during an adjustable time slot.
- If the window in a room is open, the adjustable minimal temperature has to be maintained irrespective of the presence of persons in the room.
- If someone is present in a room, the comfort temperature is exceeded, and the outdoor temperature is lower than the current room temperature, then the window has to be opened.
- The adjusted temperature of the heating boiler must not exceed the maximum of the radiator temperatures requested by the different rooms.

3 Object-Oriented Design

The aim of object-oriented design is the realization of the requirements through interacting objects. Two kinds of designs can be distinguished: an application-oriented design that encompasses objects resulting from the application and a technical design that reflects the concrete hardware and the execution platform (e.g., with parallel or sequential processing). Here, we look at the application-oriented design only.

Figure 1 shows a first class diagram for the case study. We repeat classes in the diagram for layout reasons. To indicate particular members of a class (e.g. to distinguish the outdoor temperature sensor from the room temperature sensor) we use comment boxes.

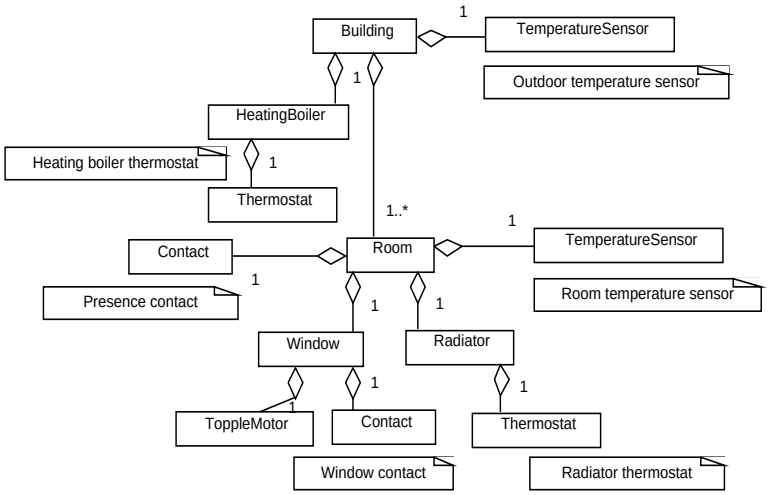


Figure 1: Class diagram for the case study

The class diagram mirrors the physical structure (the building architecture). A class is formed for each construct of the building (e.g., room or window). With aggregations, the physical structure is represented (e.g., a room has a window and a radiator). Furthermore, classes representing sensors and actuators are introduced (e.g., temperature sensor or thermostat). They are related through aggregations to the building constructs, in which they are located (e.g., each room has a temperature sensor or the building owns an outdoor temperature sensor). The sensor classes pick up the current values of the hardware sensors. The actuator classes adjust the hardware actuators.

This first class diagram results from requirements documents not given here, which describe the building architecture, the sensors, and the actuators. It does not contain any classes or operations implementing the behavior of the system (e.g., a class temperature control).

In the following sections, we discuss whether the behavior of a system should be implemented by one independent class, various independent classes or by operations of physical classes. The following criteria are important:

- The behavior of the system consists of controlling different dependencies among several sensors and actuators. The control of a dependency (called *control task* in the following) requires two decisions: The first decision determines whether an action(i.e., the control of actuators) is needed based on the (combined) sensor values (e.g., it is determined that cooling is necessary based on the values of the room temperature sensor and the presence contact). The second decision determines what action should be performed (e.g., the window has to be opened to cool off, if the outdoor temperature is lower than the comfort temperature, otherwise, nothing should be done).

- Various control tasks could conflict (e.g., the temperature control and a safety control: the temperature control wants to open the window to cool off and the safety control wants to close the window because of hazardous conditions). Conflicts have to be solved by priorities.
- Other classes can be informed of changing sensor values in an active or passive way (i.e., the values are requested periodically or the sensor class reports changes to controlling classes). Likewise, the control of an actuator class can be active or passive (i.e., an actuator class asks for new signals periodically or it gets the signals directly by controlling classes). Typically, passive sensors and actuators are preferred because periodical requests lead to a high number of object interactions.
- The determination of an action that should be performed does not only depend on the values of sensors, but also on properties of physical classes. For example, the control algorithm to calculate the heating-up temperature (the temperature guaranteeing that the comfort temperature in a room will be reached within an adjustable time slot) depends on the size of the room.
- The aggregation among physical classes will be interpreted as follows: the super class is able to call operations of the sub classes but not vice versa. This means in particular, that sensor values first get known to classes that aggregate a sensor class and that actuators are controlled by aggregating classes.
- Typically, the number of communication relations among classes should be reduced.

Strict observance of the given criteria leads to the following system structure: Classes on the top level of the hierarchy contain the controlling operations because they have the easiest access to all sensors and actuators. For example, a controlling operation "temperature control" could be assigned to the class *Building* because the outdoor temperature and the heating boiler can be accessed directly. This assignment is not intuitive because the temperature control takes place for each room separately. Furthermore, this is not realistic because various rooms of a building have to be maintained. This strong concentration of functionality and interaction only to few classes is not desirable.

If the control is moved to classes on lower levels of the hierarchy - in the example to the class *Room* or to the actuator classes directly - the following fundamental problems arise:

- If the control task is assigned to classes on upper levels (e.g., in the class *Room*), there is still the risk of overloading when additional control tasks (e.g., light) are integrated.
- If the control task is assigned to classes on lower levels, a high number of messages is necessary to transmit values from and to sensors and actuators (e.g., the value of the outdoor temperature sensor have to be transmitted to the building and from the building to the room or the radiator).
- If the hierarchy is taken strictly, classes on upper levels have to request values of sensors and actuators periodically and have to deliver the values to classes on lower levels (e.g., the building requests the value of the outdoor temperature sensor and delivers the value to the room). This implies a further overhead of communication.
- Transfer of values or periodical requests for values could be avoided by an introduction of communication relations among sensors and actuators. The relations can lead to an overload of these classes.

This discussions shows that the assignment of a whole control task to one physical class is unsatisfying. Thus, a partitioning of the control task could be useful.

In our example, one class could monitor the occupancy of the room and the room temperature. Based on these values, the class could determine whether the current temperature is too high, too low or fine. If the current temperature is too high, another class could monitor the outdoor temperature and determine whether the window can be used to cool off the room. A third class could activate the topple motor of the window to cool off the room.

These partial functions could be integrated in the physical classes. But still, there is the risk of an overload of the central class. In addition, it is difficult to locate the control task as a whole later (e.g., in the case of modifications).

Since also the assignment of partial control functions to physical classes does not lead to a satisfying solution, the introduction of control classes seems to be useful. This is comparable with the domain of information systems where the assignment of complex functionality to data classes has to be handled. In this case, the following guidelines should be applied:

- To locate the control functionality easily, a class should be introduced for each control task (e.g., a class *TemperatureControl*).
- To reduce communication relations among classes to sensors and actuators, auxiliary classes should be introduced. These auxiliary classes combine sensor values and bundle actuator signals. In our example, a class *Presence* could be introduced that stores the time a person has occupied a room and compares the time with the predefined time slots. Also, a class *RoomTemperature* could be introduced that compares the current room temperature with the aspired (comfort or minimal) temperature. Typically, the auxiliary classes are used by different control tasks (e.g., the class *Presence* is important for the light and the temperature control).
- The control classes should not be integrated in the aggregation hierarchy, because they communicate with the physical classes as well as with the sensors and actuators (through the auxiliary classes).
- If the control of actuators is complex, it is useful to introduce further auxiliary classes. These auxiliary classes allow to distinguish between the determination of the action that should be performed and the actual control of the actuators (not part of our case study).
- Priorities could be implemented by the actuator class, if only one actuator is affected. Otherwise, further auxiliary classes have to be introduced (not part of our case study).

Figure 2 shows the additional classes of our example.

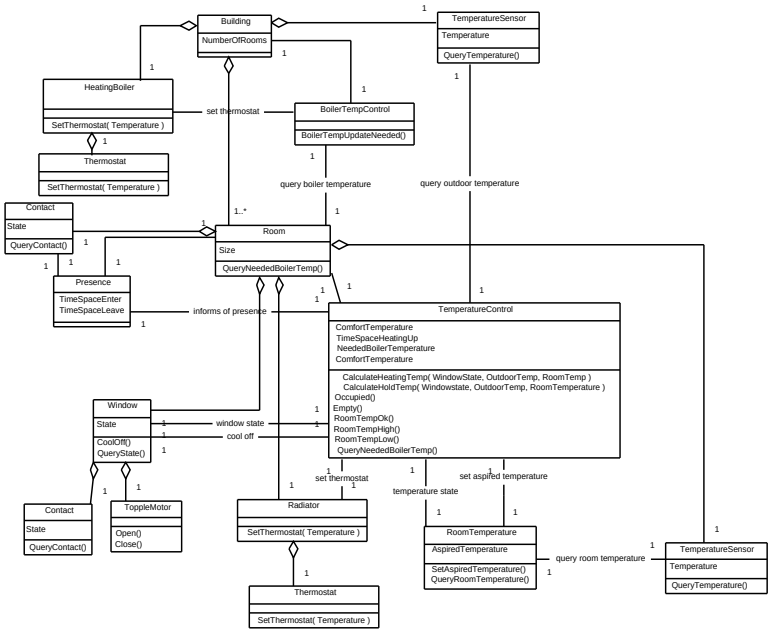


Figure 2: Class diagram extended by additional classes

The class *TemperatureControl* implements the control of the temperature in one room. The class decides which action should be performed. Furthermore, it controls the radiator to heat up and the

window to cool off. It requests the outdoor temperature to decide whether the window could be used to cool off. If the outdoor temperature is lower than the comfort temperature, the class will send a signal to cool off, otherwise, the class will do nothing.

The class *TemperatureControl* is activated by the auxiliary classes *Presence* and *RoomTemperature*. The class *Presence* requests the presence contact periodically and delivers the state of occupancy (occupied and empty) taking into account the adjustable time slots. The class *RoomTemperature* requests the room temperature sensor periodically taking into account the aspired temperature (comfort or minimal) and delivers the temperature state of the room (RoomTempOk, RoomTempHigh, RoomTempLow).

The class *BoilerTempControl* controls the temperature of the heating boiler. The class adjusts the thermostat of the heating boiler to the needed temperature. Therefore, the class requests all needed boiler temperatures from the different rooms and determines the maximum of the temperatures. Then, it adjusts the thermostat of the heating boiler to this value.

The class *BoilerTempControl* is activated by a room (initiated by the class *TemperatureControl*) that needs a new boiler temperature.

4 Related Work

OCTOPUS [AKZ96] is one of the few well-known object-oriented development methods in the domain of embedded systems. OCTOPUS combines the methods Fusion [CAB+93] and OMT (Object Modeling Technique) [RBP+91]. It starts with a description of the system functionality with use cases. Then, the system is decomposed into sub-systems. For each sub-system, an object model is developed during an analysis and design phase - similar to OMT. However, no guidelines on the distribution of complex system functionality is given. It is recommended to build a sub-system "hardware wrapper" to isolate the used hardware.

Another well-known object-oriented development method for embedded systems is ROOM(Real-Time Object-Oriented Modeling) [SGW94]. Basic modeling constructs are actors. Actors are system components which act independently and communicate with each other. The behavior of an actor is described with ROOMcharts. Similar to OCTOPUS, ROOM does not give guidelines on the distribution of complex system functionality.

In STATEMATE [HP98], a decomposition of activities as well as the control and data flow among different activities of the system are modeled. In the given examples, activities are only parts of control tasks between one sensor and one actuator. It is not clear how to proceed in the case of complex control tasks.

References

[AKZ96] M. Awad, J. Kuusela, and J. Ziegler. *Object-oriented Technology for Real-Time Systems.* Prentice Hall, 1996.

[CAB+93] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development - The Fusion Method.* Englewood Cliffs, NJ: Prentice Hall, 1993.

[HP98] D. Harel and M. Politi. *Modeling reactive Systems with Statecharts.* McGraw-Hill, 1998.

[Jac92] I. Jacobson. *Object-Oriented Software Engineering.* Addison-Wesley, 1992.

[Pae99] B. Paech. *Aufgabenorientierte Softwareentwicklung - Integrierte Gestaltung von Unternehmen, Arbeit und Software.* Springer, im Erscheinen, 1999.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modeling and Design.* Prentice-Hall, 1991.

[SGW94] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-oriented Modeling.* Wiley, 1994.

Developing Reactive Objects with SDL in UML Projects

Philippe Leblanc
leblanc@verilog.fr
CS VERILOG
150, r. Vauquelin, BP 1310
31081 Toulouse Cedex, France
www.verilogusa.com

The purpose of this Position Paper is to present the OO SDL constructs – basic and advanced –, how they can be applied to the development of reactive objects, and how SDL design can be introduced in UML projects.

Keywords: SDL, UML, reactive object, reuse, inheritance, redefinition

Reminder on UML Diagrams and Application Scope

The UML is an object-oriented modeling technique suited for the analysis and development of information systems for which the Class diagram is the core model. Additional diagrams, mainly Use cases, Sequence diagrams and State diagrams, help defining the dynamic behavior of the system and its objects, therefore making UML applicable to real-time development.

In this paper, objects whose behavior is defined with a State diagram will be called *reactive objects*. The engineering technique described here focuses on these objects.

As far as real-time design is concerned, UML has some weaknesses:

- There is not yet any standardized Action language associated to UML diagrams to formally describe transition actions or operation bodies. This is however required when a user wants to check the model before its implementation and also for automatic code generation.
- The resulting semantics of a complete model made of Class, Sequence and State diagrams is not known: diagrams are partially redundant, leading to inconsistencies in some places. You can use several interaction mechanisms within the same application (non-blocking asynchronous interaction, remote procedure call, etc.), what is the resulting global behavior?
- Usual UML objects provide operations which can be inherited and redefined locally to match current application needs. In case of reactive objects, the redefinition of their behavior can also require the modification of their related State diagram when reactions to events greatly differ from the original behavior. This is not possible in basic UML.

In the real-time domain, e.g. in telecom applications, model testing, automatic implementation and building and reuse of adaptive concurrent objects are key-issues in today's challenges. The OO concepts available in the latest versions of the SDL notation – entity types, gates, packages, inheritance of entity type, redefinition of transition, remote procedure call – now allow software developers to build adaptive reactive objects and generic architectural patterns, and to reuse them by specializing their behaviors. The resulting applications can be statically and dynamically checked, and can be converted to executable code.

The SDL Design Notation

SDL Modeling Principles

SDL (Specification and Description Language) is an ITU-T (International Telecommunications Union - Telecommunication sector) recommendation, referenced Z.100. A steady and complete version has been issued in 1988, known as SDL-88. We first present this version, the OO enhancements will be introduced later.

SDL-88 is dedicated to the specification and design of real-time distributed systems. SDL allows, first, the description of the system architecture in the form of a set of autonomous reactive classes, called *processes*. Then each process is detailed with an extended finite state machine. At run-time, processes are dynamically instantiated similarly to UML classes.

Communication between objects is a point-to-point asynchronous communication (a new synchronous communication mode has been introduced later; this will be detailed hereafter). SDL also includes the concept of logical time, mainly to handle timers.

In addition, as SDL has been designed for the engineering of large-size complex systems, e.g. public network switching systems, constructs for hierarchical refinement have been added to ease iterative development.

Description of the System Architecture with SDL-88

The system is described in the form of a hierarchy of entities: the root entity is called *system*; intermediate entities are called *blocks*; and terminal entities are called *processes*.

A block is an autonomous entity exchanging *signals* – with or without data – with its environment, through *channels*. Block refinement is described by means of Interconnection diagrams. Signals must be declared in accordance with their use, and signal flows must be specified on channels.

Blocks are then refined in an iterative way, down to the level where the identified structural entities can be described by means of finite state machines. For each new Interconnection diagram, the designer must specify how the interfaces described for the block being refined are supported by its components.

Description of the Behavior with SDL-88

An SDL process is a finite state machine extended with data. It possesses:

- a common input queue storing all the received signals not yet consumed,
- a set of local variables with related functions,
- a set of state-to-state transitions, described in Transition diagrams.

At run-time, a process can be dynamically instantiated several times: the SDL system therefore corresponds to a flat network of distributed autonomous reactive objects, evolving during the system life (intermediate structuring levels have disappeared).

Variables, as well as signal parameters, must be typed. For this purpose, SDL provides a set of predefined types: *boolean*, *integer*, *real*, *charstring*, etc. Furthermore, to manipulate more complex data, SDL provides type generators to build arrays (*array)*, lists (*string*) or structures (*struct*). Similarly to UML passive classes, new types (*newtype* or *Abstract Data Type*, *ADT*) can be built encapsulating data (similar to attributes) and *operators* (similar to operations).

In case of complex or repetitive behavior, parts or sets of transitions can be described in *procedures*. SDL procedures are designed in the same way as processes are: procedures share the common input queue of their processes and can access their variables.

Compared to UML, Harel's Statechart, ROOM or Shlaer-Mellor, SDL provides a complete range of graphical symbols to detail the content of a transition: *signal output*, *timer set/reset*, *task* (contains an SDL expression), *procedure call*, *process instance creation*, *decision*, etc.

Reuse in SDL-88

Reuse of reactive objects

The top-down iterative approach can also be mixed with a bottom-up approach when low-level structural entities are already available. In this case, the designer will build new Interconnection diagrams by connecting existing components together with newly developed objects. This reuse mechanism however does not allow the redefinition of components.

Reuse of abstract data types

If the designer has already available some parts of the application code, e.g. in C source code, the best way to integrate them is to create dedicated ADTs encapsulating the existing functions in operators. Legacy code is so integrated in a proper way and its use within transition bodies can be semantically checked (at the type checking stage). However, operators cannot be redefined.

Reuse of procedures

In SDL-88, another way to reuse reactive parts is to declare these transitions in dedicated procedures. Similarly to ADT operator calls, procedure calls can be semantically checked. However in SDL-88 this mechanism is limited as procedures must be local; SDL-96 solves this with the RPC mechanism.

The New SDL Concepts for Reuse

OO concepts were introduced in SDL-92 (issued in 1992): structural types, packages, inheritance and redefinition of transition. The most recent literature on SDL gives complete

presentations of these concepts. A new version of SDL was issued in 1996 which provides, compared to SDL-92, corrections and minor modifications only.

Structural Types

Structural types have been introduced in SDL to describe entities, like processes or blocks, independently of their use: a structural type is a separate partial SDL model which can be integrated into new SDL architectures with possible adaptation.

Process type and reusable behavior

A *process type* corresponds to a Transition diagram with declaration of variables and procedures if needed, to which we have added *gates*: a gate is a communication port through which signals can be received and transmitted (received signals are still stored in a single common input queue). Finally, a process type can be considered as a class with a set of public interfaces and a private behavior.

To use a process type within an SDL model, you must create a local empty process and declare this process as based on this process type. Then gates of the process type must be connected to the channels specified in the SDL model, in order to enable exchange between the process type-based local process and its environment.

Block type and architectural patterns

Similarly to process types, concepts of *block type* and *system type* have been introduced in SDL. A block type is a component made of blocks and processes, and has a set of gates; system type is the same concept at system level.

Using a block type is similar to using a process type: a block must be created locally and must be declared as based on the block type. Gates must be connected to channels to allow communication within the model. Thus thanks to block types, an SDL model can be made by simply combining architectural patterns.

Packages

In order to make the description of partial behavior, partial architectures or patterns easier, the concept of *package* has been added to SDL. A package is an SDL model containing any kind of types: data types, signal types, process types, block types or system types. A package corresponds to a library of reusable types (reactive classes, passive classes and types). A package is imported with the *USE* clause.

Inheritance and Redefinition of Behavior

The concept of reusable reactive objects has been extended with capabilities to inherit and redefine transitions. For example, it is possible to create a new process type based on an existing one, and to add states, to redefine existing transitions or to add new ones.

Transition redefinition is a controlled mechanism: in the super-process type, a transition must be declared as *virtual* to allow its redefinition, in the sub-process type, the redefined transition must be declared as *redefined*. If a transition is declared as *finalized*, it is no longer possible to redefine it.

If new signals are processed by the sub-process type, then they must be added to the inherited gates.

Compared to UML, the SDL inheritance mechanism is clearly defined in the language and there is no ambiguity when using it.

Remote Procedure Calls

Communication between SDL-88 objects was only made by an asynchronous exchange of signals. Then to model a client-server interaction where the client waits for the server reply before continuing its work, SDL-88 imposed to create two signals: one for the request, another one for the reply, with an intermediate state added to the client.

SDL-92/96 now includes the concept of *remote procedure call* or *RPC*: a procedure declared in the server process as *exported* can be called by a remote process, the client (procedure declared *imported* in the client). This interaction looks like a synchronous rendez-vous for the client process: it is suspended until the request is completely processed by the server.

This RPC mechanism can also be used to reuse parts of behavior. When parts of behavior are of general-purpose enough, they can be described in procedures declared exported in a server process. Other processes may call them without duplication of code.

Similarly to transitions, exported procedures can be redefined (with *virtual* and *redefined* protection mechanisms).

UML-SDL Process

How SDL design is introduced in a UML-centric process? On the one hand, UML is adequate at analysis level for a large scope of applications and at design level to develop data-oriented software (non-reactive objects). On the other hand, SDL is efficient when designing real-time architectures and developing reactive objectives. Therefore the engineering process we promote for a consistent use of SDL with UML is made of the following phases, activities and diagrams:

- **Analysis**: OO analysis with Class diagrams, Use case analysis with system-level Sequence diagrams
- **Architectural design**: system and software architecture with Interconnection diagrams
- **Test design**: test cases described with internal Sequence diagrams
- **Object design**: reactive objects designed with Transition diagrams, passive objects designed with Class diagrams
- **Implementation**: automatic code generation from reactive objects, hand-written code (C++, Java) for passive objects
- **Test execution**: automatic conversion of test cases into executable test suites, e.g. TTCN, TCL or C

Tool Support

ObjectGEODE fully adheres to this approach. The toolset supports UML for OO analysis and data design and SDL for reactive object development. In addition, it integrates the complementary ITU-T recommendations ASN.1 (Z.105) for the telecom data modeling and MSC (Z.120) for the Sequence diagrams.

ObjectGEODE integrates tools to specify, design, simulate and generate software systems: notation-oriented graphical editors, static semantic checker, simulator, validator, C/C++ code generators.

The strength of SDL and ObjectGEODE compared to UML and tool support available so far is that the rules governing safe reuse are formally defined (in SDL) and checked (by ObjectGEODE). Misreuse is detected both from a static point of view – through type checking (data and structural entities) – and from a dynamic point of view – through model simulation and validation.

As far as code generation is concerned, mature and efficient C code generators are provided by tool vendors since many years. Successful deployments of automatically generated code have been achieved for various applications, such as software embedded in mobile phones, switching systems, ATM equipment, on-board software for aircraft and satellite.

Future of UML and SDL

Current SDL can be consistently combined with UML at the engineering process level. At the software component level, differences in the language constructs make the mixing of UML and SDL objects more difficult. However the works undertaken by the respective standardization bodies give the opportunity to tightly integrate SDL with UML.

Several RFPs have been opened by the OMG early in 1999 to improve UML in the engineering of complex and real-time systems. In parallel, the ITU-T is preparing a next version of SDL, SDL-2000, (to be released by end of 1999) which will incorporate most of the remaining OO concepts available in UML, Java or C++ (interfaces, type-based dynamic instantiation, etc.) in order to deliver an SDL language compliant with these design and programming techniques, mainly for allowing joint development.

The solution presented in this paper takes these works into account. The ObjectGEODE methodology and toolset will be improved continuously in accordance with the achievements obtained by the OMG and ITU-T.

---

*Philippe Leblanc is **Senior Consultant** in CS VERILOG. He works in the real-time software engineering domain for 16 years. He has accompanied major telecom manufacturers in the appropriation of SDL and OO techniques.*

Verhaltensmodellierung eingebetteter Systeme mit dem *OCoN*-Ansatz

Holger Giese, Jörg Graf und Guido Wirtz
Institut für Informatik, Westfälische Wilhelms-Universität
Einsteinstraße 62, 48149 Münster, GERMANY
Telefon: +49 (0)251 - 83 – 33 75 0
jgraf@math.uni-muenster.de
http::/wwwmath.uni-muenster.de/cs/u/jgraf

Hintergrund und Einführung

Mit der fortschreitenden Entwicklung von Fahrzeugen, Gebäuden, Maschinen und Kommunikationsmitteln in den letzten Jahren sind auch die Anforderungen an Funktion, Sicherheit und Wartung gestiegen, die durch ihren alltäglichen Gebrauch an diese Produkte gestellt werden. Diese Steigerung von Anforderungen durch die Prozesse und Techniken der Benutzer übertragen sich unmittelbar auch auf die Prozesse und Techniken der Konstrukteure, die versuchen, den gesteigerten Anforderungen durch verbesserte Produkte gerecht zu werden. Hierbei gilt es, die mit der Anforderungssteigerung einhergehende Zunahme an Produktkomplexität durch weiterentwickelte Prozesse und Techniken während der Produktkonstruktion zu bewältigen. Aus diesem Grund wird im Fahrzeug- und Maschinenbau sowie in der Gebäude- und Telekommunikationstechnik zunehmend Software integriert, die in ihren Produkten statt Hardware zur Steuerungen und Kontrolle dient. Dadurch kann der steigenden Komplexität und dem permanenten Änderungsdruck dieser Produkte flexibeler begegnet werden, als dies mit Hardwaresteuerungen möglich ist.

Der Einsatz von vollintegrierter Steuerungssoftware in industriell gefertigen Produkten zeigt allerdings, daß diese Software Eigenschaften besitzt, die mit konventionellen Softwareentwicklungsmethoden entweder nur rudimentär oder überhaupt nicht entwickelt werden können. Die Ursache hierfür liegt darin, daß diese Entwicklungsmethoden meist Softwaresysteme voraussetzen, die schwerpunktmäßig Daten verwalten und häufig Computer oder Computernetzwerke als Laufzeitumgebungen besitzen. Softwaresysteme mit hohen Steuerungsanteilen, die durch reaktives und/oder zeitabhängiges Verhalten charakterisiert sind und in Minimalumgebungen von Produkten ablaufen sollen, sind damit nur unzureichend erfaßbar. Als Folge hiervon hat sich in der Softwaretechnik der Begriff der *„eingebetteten Systeme" (engl. embedded systems)* herausgebildet, der die besonderen Eigenschaften dieser Steuerungssoftware in einer eigenständigen (jedoch sehr weitläufigen) Systemklasse diskutiert. Doch obwohl hier die Eigenständigkeit dieser Systemklasse angesprochen wird, teilen eingebettete Systeme viele Eigensschaften mit verteilten Systemen (engl. distributed systems), da bei ihnen ebenfalls Systemaspekte wie z.B. *Nebenläufigkeit*, *Verteilung*, *Zeitabhängigkeit*, *Fehlertoleranz*, usw. relevant sind. Es können daher viele Erfahrungen aus dem Entwurf verteilter Systeme auch auf den Entwurf eingebetteter Systeme übertragen werden.

Auf der Realisierungsebene eingebetteter Systeme finden sich ebenfalls solche Gemeinsamkeiten. So kommen z.B. auch hier zunehmend Techniken zum Einsatz, die bereits für verteilte Systeme erfolgreich sind (vgl. z.B. echtzeitfähige CORBA-Busse [1] und Echtzeiterweiterungen für JAVA [2]). Dies ist u.a. deshalb möglich, weil sich die Laufzeitumgebungen von verteilten Systemen und eingebetteten Systemen immer mehr einander angleichen.

Im Hinblick auf die Softwareentwicklung für eingebettete Systeme haben sich aus den oben angeführten Wechselwirkungen zwischen Produkt- und Entwicklungsanforderungen und aus den Unzulänglichkeiten bisheriger Entwicklungsmethoden spezielle Softwareentwicklungsmethoden für eingebettete (und echtzeitfähige) Systeme entwickelt. Die von diesen Methoden angebotenen Ausdrucksmittel zur Spezifikation solcher Systeme sind jedoch immer noch sehr unzureichend (vgl. z.B. SA/RT [3]). Ein generelles Problem stellt hierbei die Inkonsistenz zwischen modellierten Systemeigenschaften in Analyse- und Entwurfsmodellen dar und innerhalb dieser Systemmodelle zwischen modellierten Systemstrukturen und modelliertem Systemverhalten. Um dieser Konsistenzproblematik entgegenzuwirken, nutzen viele Entwicklungsmethoden für eingebettete Systeme mittlerweile die Objektorientierung (vgl. ROOM [4] und OCTOPUS [5]). Sie ermöglicht durch ihren verantwortlichkeits-orientierten Systembegriff nicht nur konsistenzerhaltene Übergänge zwischen Analyse-, Entwurfs- und Realisierungsmodellen, sondern auch eine konsistentere Darstellung der Verhaltenseigenschaften (Operations-, Nachrichten-, Signalschnittstellen) in dafür verantwortlichen Struktureinheiten (Klassen und Objekte). Mit der UML [6] steht mittlerweile ein standardisiertes, in seiner Notation akzeptables Ausdrucksmittel zur Verfügung, das auch zunehmend von objektorientierten Entwicklungsmethoden für eingebettete Systeme genutzt wird (vgl. Real-Time UML [7] oder UML/RT als Weiterentwicklung von ROOM [8]).

Neben der angedeuteten Konsistenzproblematik zwischen modellierten Systemstrukturen und modelliertem Systemverhalten stellt die Modellierung der Interaktions- und Ablaufdynamik von Systemverhalten ein weiteres Problem nicht nur für eingebettete Systeme dar. Dies wird noch dadurch verstärkt, daß neben Reaktivitätsaspekten auch zunehmend Nebenläufigkeits- und Fehlertoleranzaspekte usw. berücksichtigt werden müssen. Gerade hier versucht der nachfolgend vorgestellte *OCoN*-Ansatz, eine integrale und konsistente Technik anzubieten.

Der OCoN-Ansatz

Das Akronym „*OCoN*" (Abk.: „*Object Coordination Nets*") steht für eine visuelle Modellierungsmethode, die sich in ihrer Notation und Modellierungsweise an High-level Petri-Netze anlehnt. Hierbei wird aber keine High-level Petri-Netz-Sprache (z.B. Prädikat-Transitions-Netze [9] oder objektorientierte Petri-Netze [10,11,12]) verwendet, sondern ein eigenständiger Sprachformalismus, der sich sehr eng an der objektorientierten Strukturmodellierung orientiert. So wird z.B. mit der *OCoN*-Sprache im Unterschied zu objektorientierten Petri-Netzen eine Systemstruktur nicht durch eine Menge von Netzspezifikationen beschrieben. Vielmehr

wird mir ihr die Verhaltensdynamik von Schnittstellen, Teilsystemen, Klassen, Komponenten und Operationen *komplementär* zu ihren strukturellen Eigenschaften beschrieben. Ihre strukturellen Eigenschaften werden hingegen weiterhin netzunabhängig durch die Systemstruktur dargestellt.

Der *OCoN*-Ansatz [13,14] umfaßt eine visuelle Modellierungssprache zur Modellierung von objektorientiertem Systemverhalten und eine Menge von Vorgehensweisen, die vorgeben, wie das modellierte Systemverhalten in eine objektorientierte Systemstruktur integriert wird. Die Idee der *OCoN*-Sprache geht hierbei von einer *Implementierungsabstraktion* aus, die die Verhaltensdynamik eines Objektsystems und seiner kooperierenden Objekte bis auf die *Koordinationsaspekte* reduziert. Diese können visuell dargestellt werden, ohne daß dabei auf eine konkrete Implementierungssprache wie z.B. C++, JAVA oder EIFFEL Bezug genommen werden muß.

Ein Schwerpunkt der *OCoN*-Sprache liegt u.a. in der *nahtlosen* Darstellung des modellierten externen Reaktionsverhaltens von Objekten einer Klasse und dessen Benutzung in spezifizierten Abläufen von Methoden anderer Klassen (vgl. [15]). Da die *OCoN*-Sprache hierdurch visuelle Ausdrucksmittel zur Modellierung externer wie interner Systemdynamik zur Verfügung stellt, kann mit ihr das Verhalten eines Objektsystems bezogen auf seine Koordinationsaspekte vollständig *visuell* spezifiziert werden. Weitere Eigenschaften der *OCoN*-Sprache sind:

- ***Reaktive Systemsicht:*** Die Systemdynamik wird mit Hilfe von Ereignissen, Zuständen und Zustandsübergängen über Transitionen modelliert.
- ***Kontrakt-Prinzip:*** Das extern benutzbare Verhalten von Objekten wird durch Kontrakte festgelegt. Mit ihnen sichert eine Klasse ihren Benutzern bestimmte externe Interaktionsabfolgen zu, die ihre Objekte leisten (Kontrakt = Schnittstelle + zugesicherte Interaktionsabfolgen).
- ***RPC-Synchronisation:*** Als Hauptinteraktionsmechanismen zwischen Objekten dient der *synchrone* und *asynchrone Prozedurfernaufruf (engl. remote procedure call, Abk.: RPC)*. Es kann aber auch ein *einfacher* und *wechselseitiger Signalübermittlungsmechanismus* verwendet werden[1].
- ***Nebenläufigkeit:*** Sie ist durch die notationelle und semantische Nähe der *OCoN*-Sprache zu Petri-Netzen kanonisch integriert. In Verhaltensspezifikationen wird sie als „Normalfall“ angesehen und kann durch Hinzufügen von Synchronisationskonstrukten wie z.B. Zuständen, RPCs, Triggern und Ressource-Sperren eingeschränkt werden.

Die Struktur einer *OCoN*-Spezifikation besteht im wesentlichen aus einer Menge von drei Diagrammtypen (Netztypen), die jeweils unterschiedliche Verhaltensaspekte eines Teilsystems, einer Klasse oder einer Komponente festlegen. Die Diagramme dieser drei Diagrammtypen sind bei einer Verhaltensspezifikation in gewisser Weise optional, da sie auch entfallen können, wenn z.B. bestimmte Verhaltensaspekte nicht berücksichtigt werden sollen (z.B. das Verhalten trivialer Klassen oder Operationen). Folgende drei Diagrammtypen werden unterschieden:

(1) ***Protokoll-Netze*** beschreiben bestimmte Einschränkungen auf den Benutzungsabfolgen von Operationen, die in Kontrakten extern angeboten werden. Bieten Teilsysteme, Klassen oder Komponenten ihren Benutzern solche Kontrakte an, müssen sie das Verhalten der darin aufgeführten Operationen und ihre Benutzungsreihenfolgen realisieren.

(2) ***Ressource-Allokations-Netze*** beschreiben das Reaktionsverhalten von Teilsystemen, Klassen oder Komponenten, wenn ihre Instanzen auf eigene Zustandsänderungen und auf Zustandsänderungen der von ihnen benutzten Instanzen (Ressourcen) reagieren sollen. Dieses Reaktionsverhalten realisiert einerseits das extern in Kontrakten angebotene Verhalten und andererseits auch das interne Eigenverhalten von Instanzen.

(3) ***Servive-Netze*** beschreiben die Abläufe der Instanz- und Kontraktbenutzungen, die in Methoden von Teilsystemen, Klassen oder Komponenten notwendig sind, um ihr Operationsverhalten zu realisieren.

Nachfolgend werden die oben eingeführten Netztypen der *OCoN*-Sprache und deren Anwendung in Strukturspezifikationen anhand eines Fallbeispiels erläutert.

Fallbeispiel Produktionszelle

Als Fallbeispiel zur Erläuterung der Modellierungsmöglichkeiten der *OCoN*-Sprache dient auszugsweise eine Referenzfallstudie zur Steuerung einer Produktionszelle (vgl. [17]).

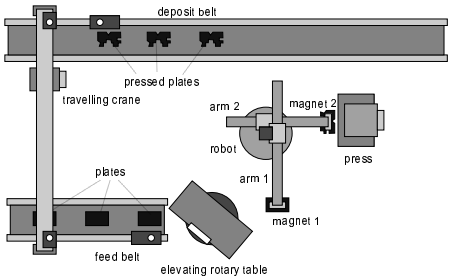


Abbildung 1: Produktionszelle

Der grundsätzliche Aufbau der Fertigungszelle ist in Abbildung 1 dargestellt. Im Folgenden wird nur der Roboter (engl. robot) betrachtet. Seine Aufgabe ist es, Platten, die ihm von einem Drehtisch (engl. rotary table) zur Verfügung gestellt werden, mit seinem Arm 1 zu nehmen, um damit anschließend eine Presse (engl. press) zu laden. Nach jedem Pressungsvorgang kann er mit Arm 2 die Presse wieder entladen und die dabei von ihm aufgenommenen gestanzten Platten auf ein Lagerfließband (engl. deposit belt) legen.

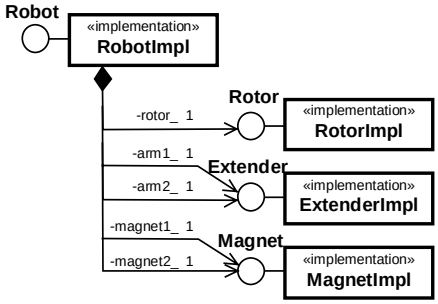


Abbildung 2: Strukturmodell des Roboters

[1]Dies wird dadurch möglich, daß die formale Semantik der *OCoN*-Sprache auf einem Nachrichtenübermittlungsmechanismus beruht, der RPCs und Signalübermittlungen gleichermaßen auf Nachrichten abbildet (vgl. [16]).

Die Systemstruktur des Roboters ist mit Drehmotor (Rotor) und den beiden Greifarmen, an denen sich Magnete befinden, in Abbildung 2 dargestellt. Die einzelnen Systemteile sind dabei über entsprechend spezifizierte Kontrakte mit der eigentlichen Robotersteuerung RobotImpl verbunden.

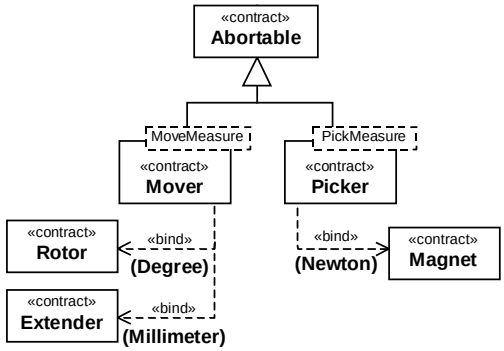


Abbildung 3: Kontraktmodell

Das zu diesen Kontrakten gehörende Kontraktmodell ist in Abbildung 3 dargestellt. Der in Abbildung 4 detailliert zu sehende Kontrakt Abortable bildet hierbei das Basisverhalten einer zu steuernden Einheit bzgl. eines spontanen internen Versagens und des externen Abruchs von Steuerungsanweisungen. Dazu werden drei Basiszustände [Base], [Active] und [Aborted] unterschieden, die durch sechseckige Stellen visualisiert sind. In jedem Zustand kann die zu kontrollierende Einheit mittels Aufruf einer abort-Operation zum Abbruch (Nothalt) ihrer Aktivität veranlaßt werden. Ein Aufruf der reset-Operation kann dann benutzt werden, um die Einheit vom Zustand [Aborted] wieder in den regulären Ausgangszustand ([Base]) zurückzusetzen. Im Zustand [Active] ist zusätzlich jederzeit ein spontaner Übergang durch die T-Aktion in den Zustand [Aborted] möglich. Die schattiert dargestellte T-Aktion legt hierbei für den Kontraktanbieter ein *willkürliches Eigenverhalten* fest, während die als zweigeteilt dargestellten Operationen *extern aufrufbares* Verhalten beschreiben.

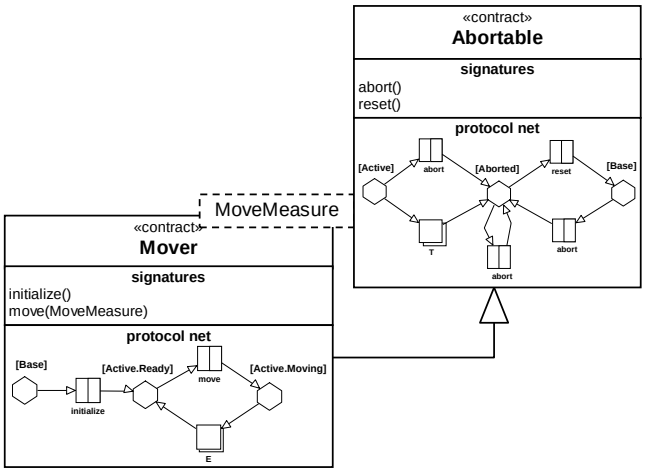


Abbildung 4: Kontrakt Abortable *und* Mover

Die Kontraktschablone Mover wird in Abbildung 4 durch eine Spezialisierung des Abortable-Kontrakt gewonnen. Die im Oberkontrakt Abortable beschriebenen Zustandsübergänge sind in jedem spezialisierten Zustand des Unterkontraktes Mover ebenfalls möglich (z.B. [Active.Ready] durch [Active]). Auf diese Weise ergibt sich das Protokoll der Kontraktschablone Mover aus der Kombination der beiden Protokollnetze. Vom Zustand [Base] kann mittels Aufruf der initialize-Operation in den Zustand [Active.Ready] gelangt werden, wo der Einheit mit einem Aufruf der move-Operation angewiesen werden kann, eine bestimmte Raumposition einzunehmen. Dabei wechselt die Einheit in den Zustand [Active.Moving]. Diesen Zustand verläßt sie erst dann wieder über den spontanen Zustandsübergang E in Richtung [Active.Ready], wenn sie die beauftragte Raumposition erreicht hat. Die schattiert dargestellte E-Aktion beschreibt hierbei im Gegensatz zu einer T-Aktion ein *zugesichertes Eigenverhalten.* D.h. jede Raumposition wird nach endlicher Zeit erreicht, da E zugesichert irgendwann schaltet. Unabhängig davon ist jederzeit auch das im Oberkontrakt spezifizierte spontane Versagen durch das willkürliche Schalten der T-Aktion in den Zustand [Aborted] möglich.

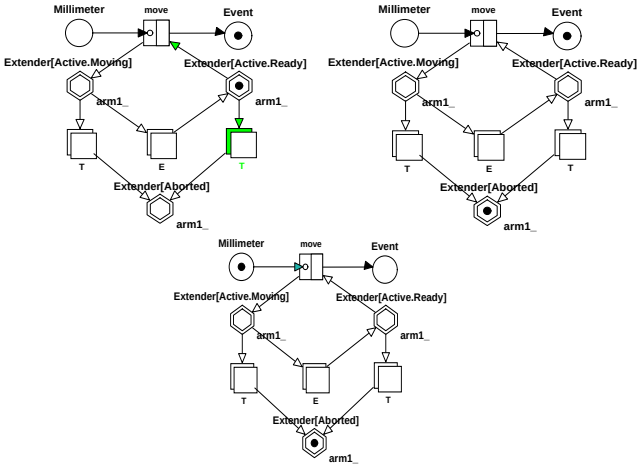


Abbildung 5: Benutzungsszenario eines Kontrakts

Die in Abbildung 5 beschriebenen drei möglichen Ergebnisse bei der Benutzung der move-Operation eines Mover-Kontraktes können durch interne Steueroperationen eines Kontraktbenutzers in eine *aufrufbasierte* Form gebracht werden, in dem die modellierten Methoden dieser Operationen entweder einen erfolgreichen Abschluß zurückmelden (linker Fall) oder ggf. eine AbortException werfen.

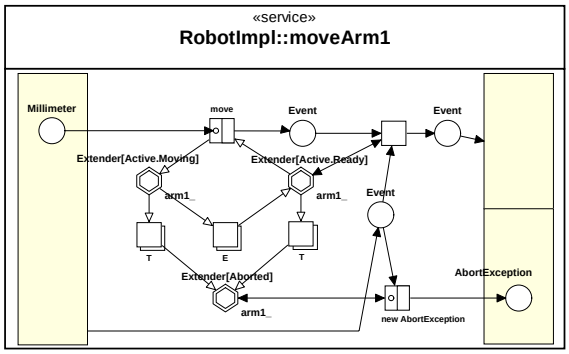


Abbildung 6: Realisierung der moveArm1*-Operation*

Im Fall der Steuerung von Arm 1 kann so die Bewegung zu bestimmten Raumpositionen durch ein Service-Netz, das die Methode der Operation moveArm1 darstellt, modelliert werden (siehe Abbildung 6).

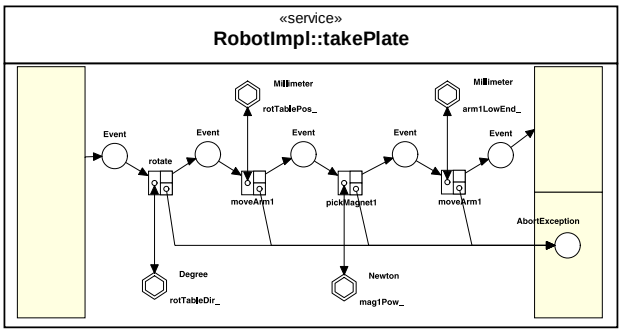


Abbildung 7: Realisierung der takePlate*-Operation*

Die oben realisierte Operation moveArm1 kann nun zusammen mit analogen Operationen moveArm2, rotate, pickMagent1, pickMagnet2 in extern benutzbare Steueroperationen zur *phasenorientierten* Ablaufsteuerung verwendet werden (vgl. RobotImpl::takePlate, Abbildung 7).

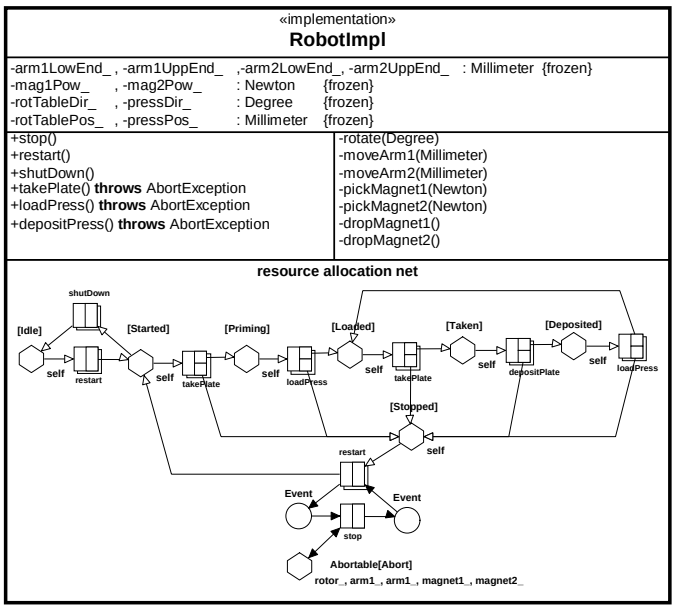


Abbildung 8: Implementierungsklasse RobotImpl

Durch die «implementation» RobotImpl kann nun ein korrekter Steuerungsablauf in Form eines Automaten mit Initialisierung (restart), Vorlauf (takePlate, loadPress) und regulärem zyklischen Ablauf (takePlate, depositPlate, loadPress), wie in Abbildung 8 beschrieben, einem externen Benutzer des Roboters angeboten werden. Die hierbei extern im Kontrakt aufrufbaren Operationen sind im Ressource-Allokations-Netz durch schattierte Aktionen dargestellt, die Übergänge zwischen den self-Stellen beschreiben. Zusätzlich kann jederzeit die stop-Operation parallel zu den anderen Operationen aufgerufen werden, um so den Roboter zum Abbruch der aktuellen Steuersequenz zu bringen. Ebenso führt das spontane Versagen eines der Roboterteile (Rotor, Arm, Magnet im Zustand [Aborted]) unmittelbar dazu, daß die im Protokoll-Netz (unten) spezifizierte stop-Operation im Ressource-Allokations-Netz (oben) ausgelöst wird. Dadurch wird jede aktuell ablaufende externe Steueroperation abgebrochen bzw. die als nächstes aufgerufene Steueroperation direkt mit einer AbortException beantwortet.

Somit ergibt sich der in Abbildung 9 beschriebene Kontrakt Robot, der eine rein aufrufbasierte Kontrolle des Roboters erlaubt.

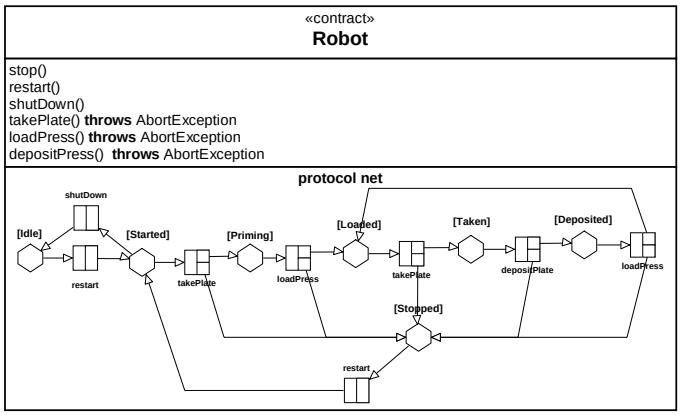


Abbildung 9: Robot*-Kontrakt*

Zusammenfassung

Durch die durchgängige Verwendung der *OCoN*-Sprache, wie in Abbildung 10 verdeutlicht, kann eine nahtlose Verhaltensspezifikation erreicht werden, in der Kontrakte bei ihrer Benutzung in Service- oder Ressource-Allokations-Netzen ebenso visuell eingebetteten werden können wie die Wechselwirkungen zwischen Service-Netzen und Kontrakten im Ressource-Allokations-Netz.

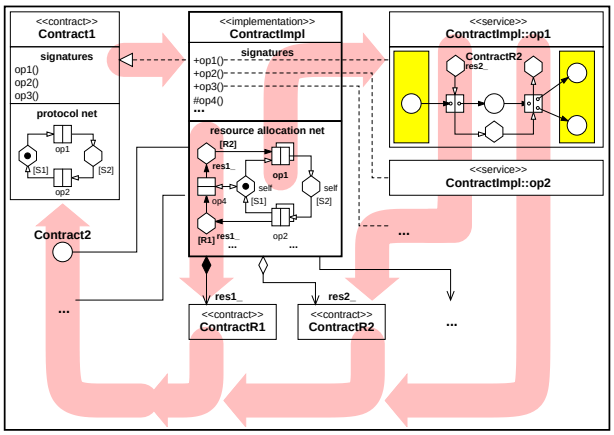


Abbildung 10: Nahtlosigkeit des OCoN-Ansatzes

Der *OCoN*-Ansatz erlaubt, durch die Unterstützung von signal- und aufrufbasierten Spezifikationen, die unteren Ebenen einer Steuerung signalbasiert zu modellieren (siehe Kontrakt Abortable und Mover), um auf höherer Steuerungsebene die benötigte stärkere Abstraktion mittels aufrufbasierter Interaktion und Teilautonomie bzgl. Ausnahmesituationen zu erreichen (siehe RobotImpl). Somit kann auch ein nahtloser Übergang von stärker echtzeitbehafteten Steuerungseinheiten hin zu komplexen, mehr die Logistik betreffenden, Steuerungseinheiten erreicht werden.

Ausblick

Da die *OCoN*-Sprache auf High-level Petri-Netzen basiert, stehen verschiedene Modellierungs- und Analysekonzepte aus der Petri-Netz-Technik zur Verfügung. So besteht z.B. mit zeitbehafteten Petri-Netzen [18] die Möglichkeit, Echtzeitanforderungen durch Zeitangaben an Stellen und Transi-

tionen miteinzubeziehen. Mit Hilfe stochastischer Petri-Netze [19] können auch Leistungsaspekte wie z.B. Durchsatzanalysen berücksichtigt werden. Es ist geplant, diese Konzepte im Rahmen der Weiterentwicklung des *OCoN*-Ansatzes in die *OCoN*-Sprache zu integrieren.

Literatur

[1] Douglas C. Schmidt, David Levine, and Sumedh Mungee. The Design and Performance of Real-time Object Request Brokers. Computer Communications, 21(4), April 1998.

[2] Robert. W. Atherton. Moving Java to the Factory. IEEE Spectrum, 35(12):18-23, December 1998.

[3] P. T. Ward and S. J. Mellor. Structural Development for Real-Time Systems, Volume 1: Introduction and Tools. Engelwood Cliffs: Yourdon Press, 1985.

[4] Bran Selic, Garth Gullekson, and Paul Ward. Real-Time Object-Oriented Modeling. Wiley, 1994.

[5] Maher Awad, Juha Kuusela, and Jurgen Ziegler. Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion. Prentice Hall, 1996.

[6] Rational Software Corporation. Unified Modelling Language 1.1, September 1997.

[7] B. P. Douglas. Real-time UML: Developing Efficient Objects for Embedded Systems. Addison-Wesley, 1998.

[8] Bran Selic and Jim Rumbaugh. Using Uml for Modeling Complex Real-time Systems. Techreport ObjecTime. 1998.

[9] H. J. Genrich. Predicate/transition Nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986 Part I, number 254 LNCS, pages 207-247. Springer Verlag, 1987.

[10] Olivier Biberstein and Didier Buchs. An Object Oriented Specification Language based on Hierarchical Petri Nets. In IS-CORE Workshop (ESPRIT), Amsterdam, September 27-30 1994. Also available as Technical Report (EPFL-DI-LGL No 94/76).

[11] C.A. Lakos. Object Petri Nets - Definition and Relationship to Coloured Nets. Techreport, Network Research Group, Department of Comupter Science, University of Tasmania, April 1994.

[12] Christoph Maier and Daniel Moldt. Object Colored Petri Nets - a Formal Technique for Object Oriented Modelling. Workshop PNSE'97, Petri Nets in System Engineering, Modelling, Verification, and Validation, Hamburg, Germany, September 1997.

[13] Guido Wirtz, Jörg Graf, and Holger Giese. Ruling the Behavior of Distributed Software Components. In H. R. Arabnia, editor, Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'97), Las Vegas, Nevada, July 1997.

[14] Holger Giese, Jörg Graf, and Guido Wirtz. Modeling Distributed Software Systems with Object Coordination Nets. pages 107-116, July 1998. Proc. Int. Symposium on Software Engineering for Parallel and Distributed Systems (PDSE'98), Kyoto, Japan.

[15] Holger Giese, Jörg Graf, and Guido Wirtz. Seamless Visual Object-oriented Behavior Modeling for Distributed Software Systems. Techreport, University Münster, March 1999. 04/99-I.

[16] Holger Giese. Object Coordination Net Specification 2.0. Techreport, University Münster, May 1999. 0X/99-I (to be published).

[17] Claus Lewerentz and Thomas Lindner. volume 891 of LNCS, chapter Task Description, pages 7-19. Springer Verlag, 1994.

[18] Robert H. Sloan and Ugo Buy. Analysis of real-time programs with simple time Petri Nets. In International Symposium on Software Testing and Analysis, pages 228-239, 1994.

[19] Christoph Lindemann. Performance Modelling with Deterministic and Stochastic Petri Nets. Wiley, 1998.

A Components Model based on Interaction-Nets

Juan M. Cordero and Miguel Toro

University of Seville
{cordero,mtoro}@lsi.us.es

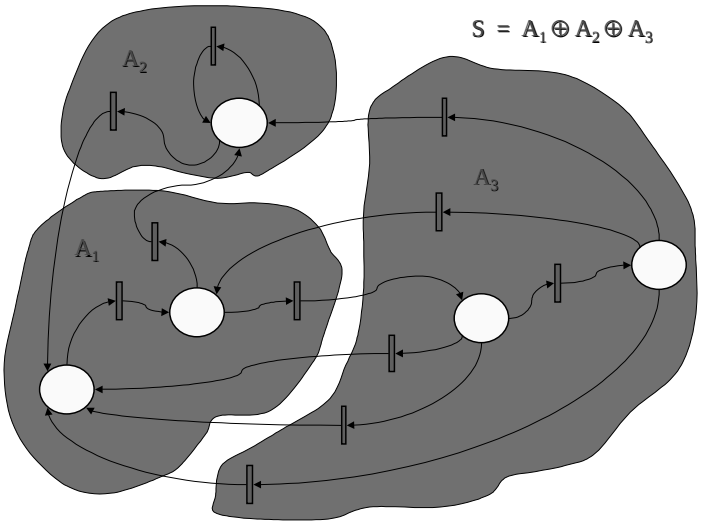
Abstract

Realtime systems have inherent complexity related to their control that makes it necessary to have a component-based approach that allow to apply the divide and conquer principles. In this paper, we give an overview of a paradigm for components-based specification. We propose a model and a language for modeling event-based interactions between components: the system is formed by components that interact through asynchronous communication.

Keywords: Realtime, Component, Agent, Petri Net, Event.

1. Introduction

We can view a system as a collection of subsystems that interact through messages. Let us suppose that we model the entire system as a large Petri net. We can divide the net in subnets where each subnet represent a part of the system. Each subnet must have an interface that define the inputs and outputs with the rest of the entire net. We are going to consider each subnet as a *component*. In this way each component has its own encapsulated Petri net that models the component's concurrent and reactive behaviour. So, systems are made of independently built components which can be composites of other components [1]. This is illustrated in the following figure.



We have chosen *coloured Petri net* [5] as the formalism for modelling concurrency because is a formal technique that enables formal reasoning about the system being modeled in terms of behaviour, properties, equivalence notions between system models, and so on. However, they have two drawbacks. First, large systems result in huge nets, even when using high-level Petri nets. Second, nets are very static. Complex and large distributed applications require flexibility and extendibility because they are dynamic.

2. Interaction-Net

An *Interaction-Net* (INet) is a high-level Petri net that models an event we are naming as INet-event. An INet makes known, through an INet-event, an interactive behaviour determined by the context. The context of an INet is the set of INets whose INet-events could produce a state change in the INet. We make use of the term "interactive behaviour" ([6]) to indicate the high-level behaviour ([9]) due to the interactions between INets through INet-events. The state of an INet is the set of encapsulated objects into the INet.

We can consider two types of events in the system. The produced events by the objects that are encapsulated in the INets, the primitive events, that is the events that cannot be produced by the INets. And the produced events by the INets, the INets-events or composite events, because they are formed by the primitive events and others INet-events by applying a few mechanisms of composition. In this way we can talk about an events algebra [7].

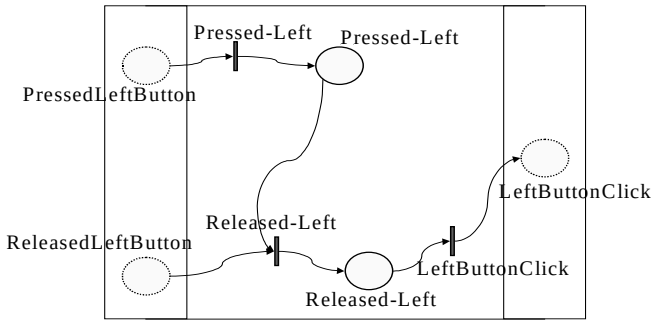
This view of an INet like an event define a new mechanism of composition of Petri nets, by allowing to make very large dynamic Petri nets in a modular way.

Let us see an example of an INet. Let us suppose that we want to model an event produced by a mouse, that is one interactive behaviour of a mouse:

```
event LeftButtonClick {
  var time : Clock;
  trans Pressed-Left ( c : short )
        when [ PressedLeftButton ]
        being [ c := time.getTime(); ];
  trans Released-Left ( c1 : short )
        when [ Pressed-Left ( c2 ), ReleasedLeftButton ]
        being [ c1 := time.getTime() - c2; ];
  fire when [ Released-Left( c ) ]
        if [ c <= 500 ];
}
```

LeftButtonClick is the name of the INet that we are modelling. There are two transitions. Pressed-Left is fired when the INet-event PressedLeftButton is present in the context, then we run the time object's getTime method. Released-Left is fired when the transition Pressed-Left and the ReleasedLeftButton INet-event are present. When the fire section is held, that is when the Released-Left is present, the LeftButtonClick INet-event is sent to the context.

The input INet-events are modeled as places. Really we model the input INet-events as tokens in the input places that have the same label that the input INet-event. The INet-event that is modeling by the INet is represented by a place with the same label that the INet. Each transition has a place with its same label. This is illustrated in the following figure.



Events are an alternative to the limitations of pre and post conditions widely known in object-orientation. Pre and post conditions are not sensitive to the history of a component [2]. They only determine a given state the component must be in to handle a message. If the component is not in the state, the incoming message can not yet or no longer be handled. They are not appropriate for open systems for that reason. We are considering an event as asynchronous message passing. Event-driven behaviour offers a promising approach for understand how systems behave. An INet-event is present, absent or undefined. It cannot be both present and absent during the same instant (coherency property).

3. Agent-Net

An Agent-Net is a context for a set of INets, determining the environment in that the INets act. An Agent-Net encapsulates objects and INets, allowing that the INets can act over the objects. These objects cannot be seen out of the INet, except when an object is a token. Such INets describe the agents' internal concurrent behaviours (multiple concurrency activities may be performed within one agent). The Agent-Nets are proactive, that is, they can decide themselves what actions they perform.

They act independently and concurrently and they cooperate through INet-events. Each agent-net encapsulate a behaviour and its interface is formed by inputs and outputs. We adopt the notion of *contracts* realized by *offered* and *required* events.

The Agent-Net formalism uses concepts borrowed from the object-oriented approach to describe the structural or static aspects of systems, and uses high-level Petri nets (Interaction-Nets) to describe their dynamic or behavioural aspects.

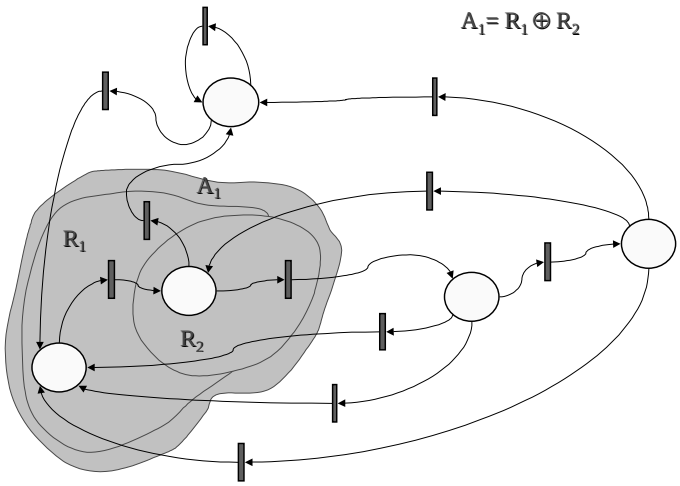
Also an Agent-Net can be described as a composition of others Agent-Nets without to declare any INet explicitly. Only declaring the inputs and outputs that take part in the interface of the agent and which one belong to each sub-agent-net.

3.1 Agent-Net like a Component

A component is an entity which manages some resources in the system and offers services to manipulate them [4]. A resource is an item with a state and procedures to manipulate it. Each component describes therefore its own functionalities and the correct way to use them. The functionalities are described as events which can model interrogations or announces according to the ODP classification [3]. An interrogation is an operation which produces a result for the caller. An announce is an operation which returns an acknowledge when the request is received and proceed latter the computation without sending a result. Beside the operations, a component can have automatically triggered operations which are not accesible from the environment. A resource is

encapsulated into a component and can be used only by the services of the component. A resource is equivalent to the notion of attribute in object-orientation.

We are going to identify a component as an Agent-Net. A component receive inputs of an context and emit outputs in function of a determined behaviour. The behaviour is fixed independent of the context by allowing to be used for to build new components. In this way we can build systems as the composition of agent-nets. This is illustrated in the following figure.



In the figure we can see that an agent-net A1 is divided in two nets: R1 and R2. The R1 and R2 INets share a place it is internally represented like an internal event into the agent-net.

4. Conclusion

We have solved the two drawbacks we had mentioned in the introduction section. On one hand, the new mechanism of composition that we have present in this paper is the solution for the construction of hierarchical nets. On the other hand, the model of interactions allow us to solve the dynamic properties of the system.

We also have to keep in mind that:

- We can carry out structural and behaviour analisis of the system.
- We can detect causality relations between events.
- The specification is based on components.
- We can carry out the description of pro-actives behaviours.
- The model allow us to reuse the features of object-oriented technologies.

References

[1] M. Bidoit and R. Hennicker. *A General Framework for Modular Implementations of Modular System Specifications*. In Proceedings of TAPSOFT'93, Springer Verlag, LNCS vol. 668.
[2] F. Puntigam. *Coordination Requirements Expressed in Types for Active Objects*. In Proceedings of ECOOP97, Springer Verlag, LNCS, vol. 1241, pages 367-388.
[3] ITU X.901..904, ISO/IEC 10746-1..4, *Basic Reference Model of Open Distributed* Processing. Draft International Standards, Geneva, Switserland, 1995.

[4] P. Wegner. *Models and Paradigms of Interaction*. In Tutorial Notes, ECOOP'94, Bologna, Italy, 1994.
[5] K. Jensen, *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical use. Volume 1 : Basic Concepts.* EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1992.
[6] P. Wegner. *Models and Paradigms of Interaction*. In Tutorial Notes, ECOOP'94, Bologna, Italy, 1994.
[7] E. Kindler, *A Compositional Partial Order Semantics for Petri Nets Components*, In Proceedings of ICATPN'97, Springer Verlag, LNCS vol. 1248.
[8] *Proceedings of the Fourth International Workshop on Software Specification and Design*. IEEE Computer Society Press, 3-4 April 1987.
[9] B. Selic, G. Gullekson and P.T. Ward. *Real-Time Object-Oriented Modeling*. Wiley, 1994.

SDL & UML, eine vollständige Lösung für die Entwicklung von Embedded Realtime Systems, von der OO-Analyse bis hin zur Target Verifikation.

Ulf Pansa, Telelogic Germany

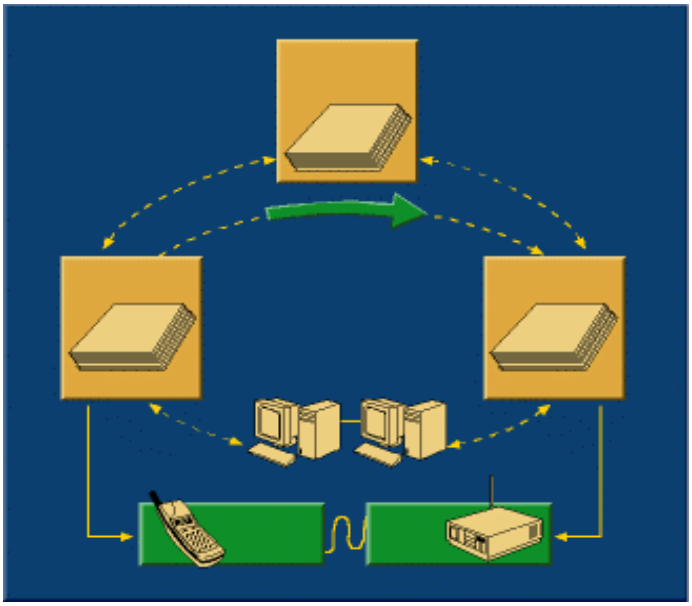
Abstract:

Thema des Vortrages sind die Spachen UML und SDL und deren methodisch-technische Anwendung bei der Entwicklung von Realzeit Systemen im Embedded Bereich. Es hat sich in der Vergangenheit gezeigt, daß sowohl UML als auch SDL Stärken wie auch Schwächen hinsichtlich der für den Entwickler notwendigen Durchgängigkeit aufweisen.

Während UML eine höhere Abstraktion im Bereich der Objekt Orientierten Analyse aufweist, hat sich SDL seit vielen Jahren in der High Tech Industrie als Standard im Objekt Orientierten Design und allen darauf folgenden Entwicklungsphasen bis hin zur Target Testen und Debug etabliert. Gerade die Beschreibung und das Testen komplexer Echt- Zeit- Systemen stehen im Vordergrund bei dem Einsatz von SDL.

SDL bietet hier die uneingeschränkten Vorteile einer formalen Technik, welche die Automatisierung von Verifikation, Validierung sowie die Erzeugung von Test Cases ermöglicht.

Für den Anwender bedeutet die Intergration von UML und SDL eine durchgängige Lösung, von der Analyse bis hin zur Implementierung und zum Testen.



UML / SDL

- Anmerkung:
Das hier beschriebene Vorgehensmodell beinhaltet keine vollständige Beschreibung ver, sonder vielmehr einen vereinfachter Ablauf bezogen auf die Umsetzung.
Im Rahmen dieses Vortrages geht es in erster Linie um einen roten Faden, welcher dem Anwender den Weg einer durchgängigen Lösung veranschaulichen soll.

UML (Unified Modelling Language) ist die jüngste Zusammenführung traditioneller Objekt Orientierter Methoden durch die OMG (Object Management Group).
UML stellt eine Standardisierung verschiedener Diagramme und Betrachtungsweisen eines komplexen Systems dar. Der Schwerpunkt liegt dabei in der Modellierung komplexer Systeme.
Neben der anfänglichen Euphorie in welcher UML als ein Synonym für die Lösung aller Probleme innerhalb der Software Entwicklung stand, folgte allmählich die Ernüchterung. Gerade in der Entwicklung von Embedded Echt-Zeit- Systeme stellt sich die Frage nach dem praktischen Nutzen, da gerade die Aspekte der Realzeit nur unzureichend berücksichtigt werden. Tatsächlich stehen dem Anwender eine ganze Reihe unterschiedlicher Diagramm Typen zur Verfügung. Mit Hilfe dieser Diagramme lassen sich komplexe Zusammenhänge sehr anschaulich darstellen, was nicht zwingend bedeutet daß auch alle Darstellungsformen verwendet werden müssen. Vielmehr soll dem System Analytiker ein Pool von Beschreibungsformen zur Verfügung stehen, mit der er die unterschiedlichen Betrachtungsweisen in definierter Form beschreiben kann.

Im Gegensatz zu UML ist SDL eine Sprache mit historischem Hintergrund. Bereits 1976 wurden international die ersten Vereinbarungen darüber getroffen, wie beispielsweise graphische Prozeßsymbole zu spezifizieren sind. Inzwischen sind für die Weiterentwicklung dieser Sprache mehr als 2.000 Mann- Jahre an Entwicklungszeit aufgewendet worden.
Die Schwerpunkte von SDL (Specification and Description Language) liegen im Gegensatz zu UML in den Bereichen System Design, Detailed Design und der Implementierung. Ein weiterer Fokus, welchen man dem Bereich des Detail Design zuordnen könnte ist das Testen (Verification, Validation, Target Testen & Debug). Hier kommen die Stärken der formalen Sprache SDL voll zur Geltung.
SDL ist mehr als eine Programmiersprache, im Gegensatz zu Sprachen wie C, C++ oder Java ist SDL auch eine formale Technik. Die Bedeutung einer „formalen -Systembeschreibung“ im Gegensatz zu einer „informellen“ wird spätestens dann bewertet, wenn ein Projekt sich aufgrund von Mißverständnissen verzögert oder gar scheitert.

SDL bietet in einer entsprechenden Entwicklungsumgebung die Möglichkeit Dokumentation, Systemdesign und Implementierung auf einer Ebene zu halten und ein Auseinanderdriften von Dokumentation und Implementierung zu vermeiden. SDL als solches ermöglicht ein Verfahren, das sehr stark einem Spiralmodell ähnelt, jedoch ohne der üblicherweise anfallenden Reibungsverluste an den Phasenübergängen.

Prozeßabbildung

Um einen vollständigen Entwicklungsprozeß zu beschreiben erscheint es sinnvoll diesen in Bereiche zu unterteilen. UML ist ohne Zweifel in den oberen Bereichen (Requirement Capture und System Analyse) sehr mächtig. Hier finden vor allem Use Case Diagram, Class Diagram und Sequence Diagram Verwendung. In den Bereichen des Design werden am häufigsten Class Diagrams, State Charts und Sequence Diagrams verwendet.

Des weiteren stehen hier das Deployment Diagram, das Component Diagram, das Collaboration Diagram und das Activity Diagram zur Verfügung. Die verschiedlichen Diagramm Typen dienen zur Beschreibung der unterschiedlichen points of view. Man unterscheidet logical view, physical view und process view voneinander.

Die von SDL am meisten verwendeten Darstellungsformen unterteilen sich wie folgt:

- SDL Block (Type) Diagramme zur Darstellung der Systemarchitektur *[Physical View]*
- SDL Prozeß (Type) Diagramme zur Darstellung der Parallelität *[Prozess View]*
- SDL Zustandsautomaten CEFSM (Comunication Oriented Extended Finite State Machine) *[Logical View]*
- SDL Procedures *[Logical View]*
- HMSC (Hierarchische Message Sequence Chart)

Note „Type“:
Ähnlich wie in C++ und Embedded C++ gibt es in SDL zwei Ebenen der Objektorientierung. So unterscheidet man beispielsweise zwischen *Block* und *Block Type*.
Es bleibt dem Anwender überlassen, ob er die (volle) Objektorientierung (Typisierung und Vererbung) einsetzen möchte oder nicht. Für sehr kleine Systeme gilt das Gleiche wie für C++ bzw. EC++. Objektorientierung Ja, aber aus Gründen von Speicherplatz und Performance nur mit Einschränkungen.

Schnittstellen und Diagramme (UML / SDL)

Um eine effiziente und durchgängige Projektktierung zu ermöglichen ist es erforderlich die notwendigen Beschreibungsformen zu definieren und sicher zu stellen daß keine unnötigen Redundanzen bestehen. Der Anwender sollte auch nicht von zu vielen Darstellungsformen erschlagen werden.
Hinsichtlich der Anzahl der zu Anwendung kommenden Diagramme läßt sich sagen: „So viel wie nötig und so wenig wie möglich!“ – Schließlich soll dem Anwender noch ein bißchen Zeit für seine eigentliche Aufgabe bleiben, dem Entwickeln.

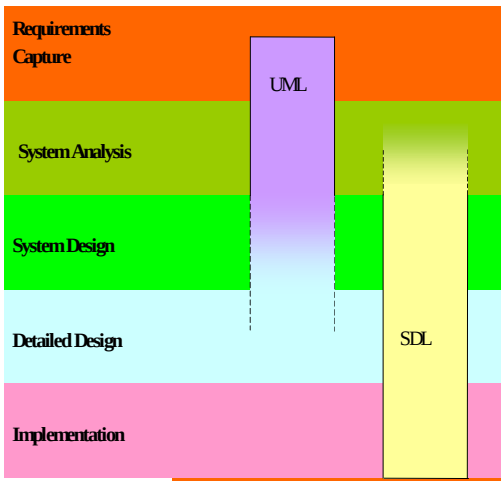
Ziel sollte es daher sein einen optimalen Übergang vom Modell bis zur Implementierung zu schaffen. In dem anschließenden praktischen Beispiel sind daher folgende Darstellungen gewählt worden.

Requirement Capture:
- Text Dokumente
- Sequenze Diagramms (MSC/ HMSC)

System Analysis:
- Class Diagramms

System Design:
- Class Diagramms
- State Charts
- SDL Block Diagramms
- SDL Prozess Diagramms
- SDL Procedures

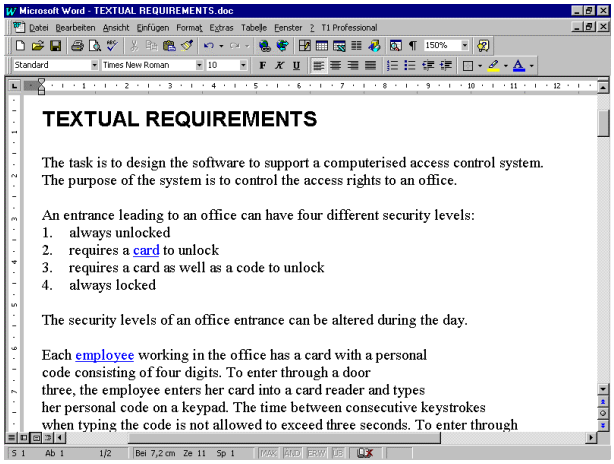
Implementation:
- Automatische Codeerzeugung
- ANSI-C
- RTOS Integration



Praktisches Beispiel „Access Controll“

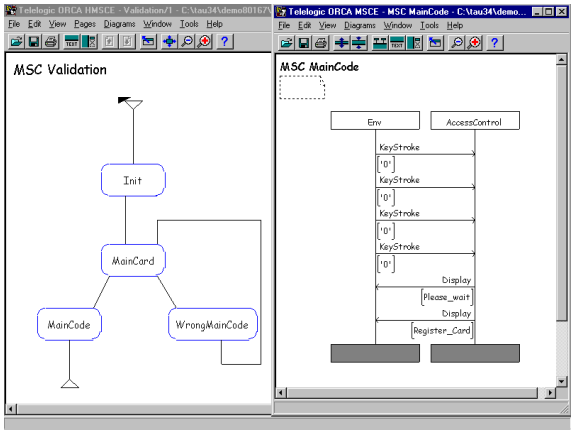
Um die kombinierte Handhabung von UML und SDL zu verdeutlichen werden die einzelnen Entwicklungsschritte anhand eines konkreten Beispiels erläutert. Das hier beschriebenen Beispiel ist die Zugangs Kontrolle für eine Gebäudeüberwachung – „Access Controll“.

Das System besteht aus 1-n Bediener Konsolen (Local Station) und einer Zentraleinheit (Central Unit) welche letztendlich die Freigabe für einen Besucher entscheidet.
Die Local Station bestehen jeweils aus einem Kartenleser einem Display und einer Tastatur (Panel). Nach Eingabe einer Magnetkarte kann der Benutzer entweder mittels eines „Super User Pin Code“ eine neue Karte inklusive Pin Code registrieren lassen oder aber mit einer bereits registrierten Karte das Öffnen einer bestimmten Tür veranlassen.



Requirement Capture

An erster Stelle einer jeden Entwicklung steht die Analyse und Erfassung der Anforderungen.
Dazu ist es nach wie vor unumgänglich textuelle Dokumente zu erstellen um mit informellen Umschreibungen die Aufgabenstellung zu definieren. Ein sehr rudimentäres Beispiel hierfür ist obige Beschreibung des Access Controll. Im nächsten Schritt werden Schlüsselwörter identifiziert welche später über einen Requirement Link Manager die Verbindung zwischen den unterschiedlichen Dokumenten (Diagrammen) bilden. Diese Schlüsselwörter ziehen sich wie ein roter Faden durch sämtliche Entwicklungsprozesse bis hin zur Implementierung.

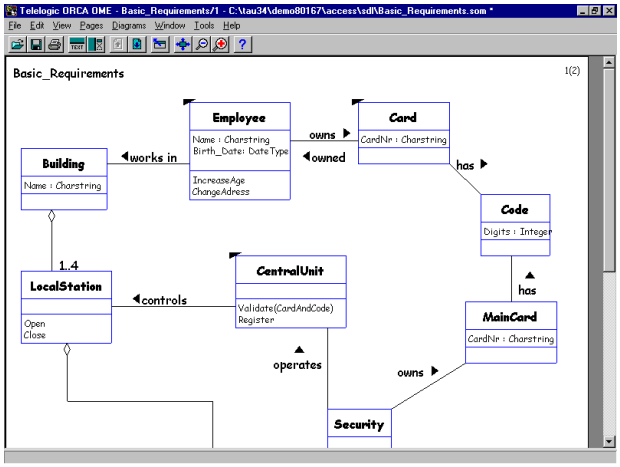


Die zweite Form der Anforderungsbeschreibung bilden die HMSC (Hierarschische Message Sequence Chart). Mit Ihnen werden funktionale zeitliche Abläufe beschrieben. Mit Hilfe der Hierarchisierung lassen sich Teilsequenzen modularisieren und somit in kleine übersichtliche Einheiten verpacken. Ein weiterer Vorteil von HMSCs ist die Darstellbarkeit von alternativen Abläufen und funktionalen Rekursionen – wie etwa bei der falschen Eingabe eines Pin Codes (WrongMainCode).

HMSCs sind der erste Schritt um eine informelle Beschreibung zu formalisieren. Später, am Ende der Designphase werden diese HMSC als Use Cases für die System Validierung verwendet.

System Analyse

Während die Requirement Analyse (Requirement Capture) das zu entwickelnde System von außen betrachtet (black box), bezieht sich die System Analyse auf die Frage der Umsetzung. Es geht hier nicht darum wie man die Umsetzung durchführt sondern vielmehr um die Frage was umzusetzen ist um die Anforderungen zu erfüllen. Hierbei spielen vor allem die UML Class Diagramme eine zentrale Rolle.
Mit dieser Darstellung lassen sich Beziehungen, Assoziationen, Generalisierungen und Aggregationen beschreiben.



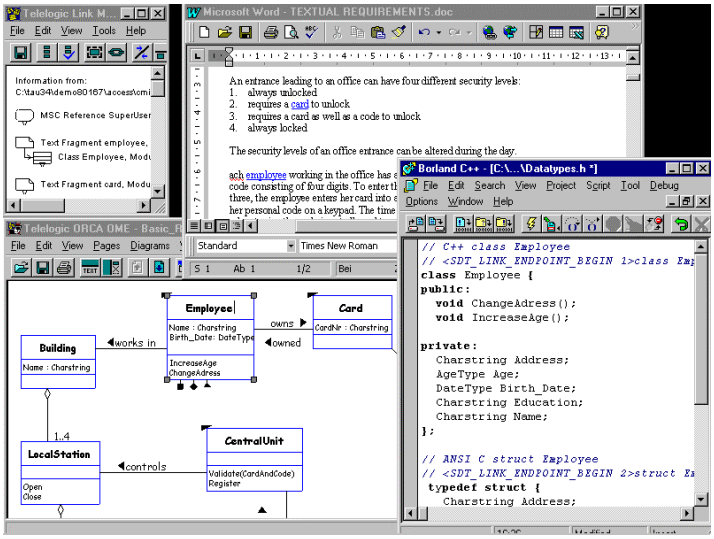
System Design

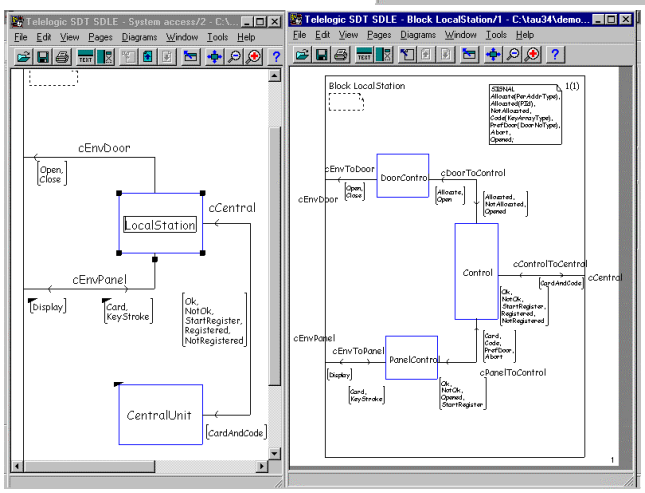
Eine elementare Frage gilt der Umsetzung von der System Analyse hin zum System Design.
Hier gibt es mehrere Möglichkeiten die Schwerpunkte zu legen, da sich UML und SDL in dieser Phase der Entwicklung teilweise überschneiden.

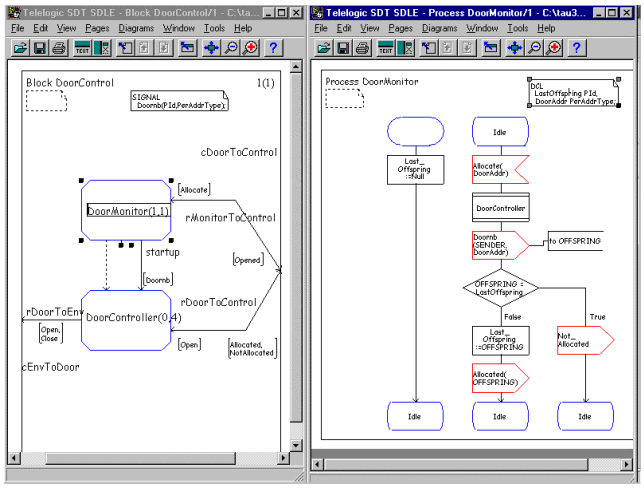
Für die automatische Umsetzung aus UML heraus stehen mehrere Möglichkeiten zur Verfügung:

Class Diagramm	**und/oder**	**Class Diagramm**
→ C++ Class Declaration		→ SDL Block Type
→ ANSI C Struct Declaration		→ SDL Process Type
→ IDL Module (CORBA)		→ SDL Block
→ IDL Interface (CORBA)		→ SDL Process
→ ASN.1 Sequence		→ SDL NEWTYPE (with Operator)

Wichtig für eine durchgängige Umsetzung, unabhängig ob das tiefere Design nun in SDL, C, oder C++ durchgeführt wird, ist die Unterstützung durch einen Linkmanager. Dieser soll dem Anwender ermöglichen von jeder Darstellungsform aus bidirektional zu navigieren. Ansonsten würde man bei der Fülle der im context stehenden Dokumente sehr leicht den Überblick verlieren. Die hierfür notwendige Linkverbindung wird bei der jeweiligen Konvertierung automatisch gesetzt. Dabei kann vom Anwender definiert werden ob es sich um einen Requirement-, einen Implementation- oder um einen Behavior Link handelt. Linkverbindungen können auch nachträglich definiert, geändert oder mit Kommentaren versehen werden.







Im Beispiel „Acess Controll“ hat man sich dafür entschieden die aus den Aggregations Modellen abzuleitende Systemarchitektur in SDL über Block Diagramme darzustellen. Block Types wurden in diesem Fall nicht verwendet, da die geplante Zielplattform (ein 8 Bit Controller – 8051 Derivat) nur über begrenzte Speicherkapazitäten verfügt.

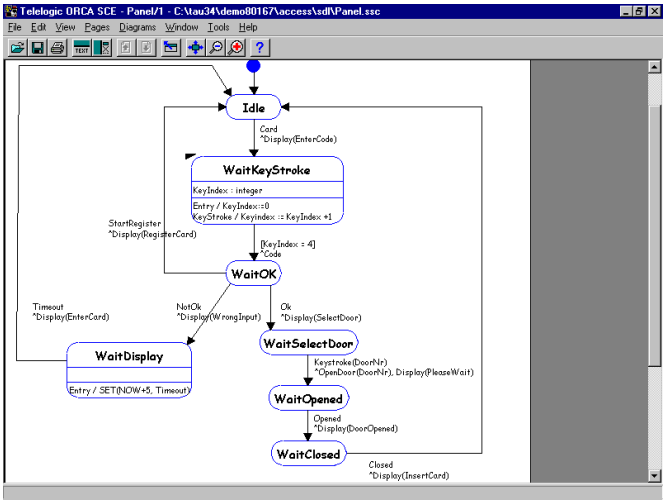
Die Darstellung links zeigt die Modularisierung des Systems (SDL Blöcke) in Local Station und Central Unit. Innerhalb von Local Staion befinden sich die Sub Module DoorControl, Control und PanelControl. Die Verbindung zwischen den einzelnen Blöcken erfolgt über Kanäle (Definition der Schnittstellen) und Signale.
In einer separaten Text Box erfolgt die Signaldeklaration. Hier wird deutlich, daß Signale sowohl einfache Events als auch komplexe Datenpakete transportieren können. Im Falle von Events enthalten die Signale keinerlei Parameter. Die schwarzen Ecken beispielsweise beim Signal Display zeugen von einer bestehenden Linkverbindung. Einen gesamtheitliche Darstellung der Systemarchitektur (Structure Tree) wird in dem Organizer vermittelt.

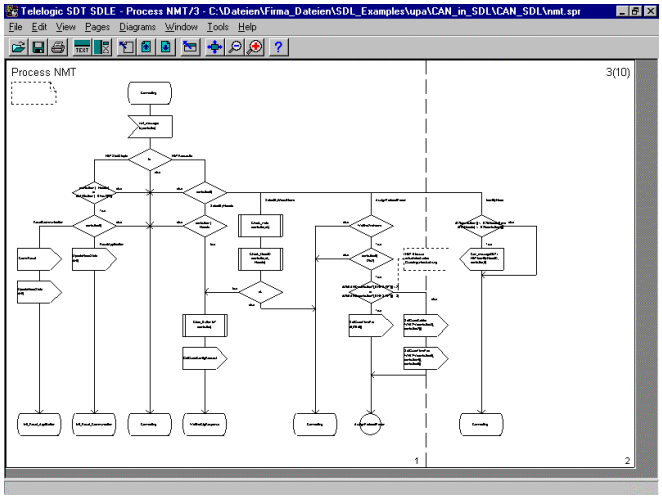
Bewegt man sich nun tiefer in die Systemarchitektur, gelangt man früher oder später auf die Ebene der Prozeßdarstellung. In dem hier dargestellten Fall innerhalb des Blockes DoorControl. Dieser Block enthält die beiden Prozesse DoorMonitor und DoorController (linke Bildhälfte). Prozesse dienen in SDL zur

Beschreibung von Parallelitäten, schließlich handelt es sich bei SDL um eine Beschreibungs Sprache für Echtzeitsysteme.

Es ist läßt sich erkennen, daß DoorMonitor statisch (1,1) und DoorController dynamisch (0,1) ist. In der Praxis bedeutet dies, daß D.M. bei Systemstart (Initialisierung) einmal und nur einmal vorhanden ist, D.C. hingegen erst zur Laufzeit des Systems bis zu vier mal kreiert werden kann.
Es ist weiterhin zu ersehen, daß die dynamische Erzeugung des Prozesses D.C. von D.M. ausgeht (gestrichelter Pfeil). Rechts im Bild erkennt man die in SDL beschriebene Logik (Behavior) von D.M.. Das Symbol (Rechteck mit zwei Linien) mit der Bezeichnung DoorController ist das hierfür notwendige Create Symbol.

Zwangsläufig stellt sich hier die Frage nach der Verbindung zischen SDL Prozess Behavior (SDL Logik) und UML. Die Antwort ist in der Darstellung rechts zu sehen. UML kennt zwar keine direkte Parallelität, dafür aber die Beschreibung von Behavior und Funktionalität in Form von hierarchischen Zustandsautomaten. Diese lassen eindeutig in die Darstellungsform der SDL Logik konvertieren und als SDL Prozeß oder SDL Prozeß Type in das System einbinden. Dem Anwender bleibt es somit frei gestellt, welche Form der Funktionalen (logischen) Beschreibung er bevorzugt.
Für reine Zustandsautomaten mit hierarchischer Gliederung (Verteilung der Komplexität auf mehrere Hierarchie Ebenen – State Level) bieten sich vorzugsweise die UML State Charts an.
Bei einer Darstellung von Zustandsautomaten mit dem Schwerpunkt in der Kommunikation oder bei Automaten mit komplexen bedingten Abläufen (siehe oben CAN NMT), eingebetteten Reglermodulen etc. gibt es keine bessere Darstellungsform als SDL. SDL bietet hier eine Kombination von FSM (Finite State Machines) und Ablaufdiagrammen. Die ausgeprägte symbolische Darstellung in SDL ermöglicht dem Anwender auch bei umfangreichen Diagrammen den Überblick zu bewahren.
Ein weiteres Feature von SDL sind die SDL Prozeduren. Sie bieten eine weitere Stufe der Modularisierung.





Noch ein wichtiger Aspekt in diesem Zusammenhang sind die Data Type Definition. Wie anfänglich erläutert gibt es eine ganze Reihe von Umsetzungsmöglichkeiten bezogen auf UML Class Diagramms. Eine wesentliche Rolle spielen die UML Classes in Hinblick auf die Beschreibung komplexer Datentypen. Diese lassen sich beispielsweise in C Strukturen konvertieren und anschließend in das SDL Design includieren. In SDL durchgeführte Declarations nehmen somit Bezug auf die in UML definierten Datentypen – der Kreis schließt sich.

Verifikation und Validierung

Wenn auch bereits im System Design die Vorteile von SDL und die damit verbundene Durchgängigkeit klar hervorging, um so mehr werden diese Vorteile ersichtlich wenn es um die Überprüfung eines entwickelten Systems geht. In UML werden leider keinerlei Testkonzepte berücksichtigt, was zum einen auch daraufhin zurückzuführen ist, da die Schwerpunkt von UML nun mal nicht in der Umsetzung sondern in der OO- Analyse und in der Modellierung liegen. Dennoch lassen sich in der Kombination UML/SDL auch Testkonzepte durchgängig gestalten. Die im Requirement Capture erstellten Szenarien Diagramme (HMSC Use Cases) dienen hierbei als Test Scripte für den SDL Validator. Dieser generiert neben einem detailierten Report der durchgeführten Testschritte auch eine Coverage View, eine grafische Darstellung der von den Testläufen abgedeckten Systemkomponenten (Abbildung auf der folgenden Seite). Der Anwender kann nun erkennen, ob alle Teile des Systems durchlaufen wurden. Ist das beispielsweise nicht der Fall, so bedeutet dies, daß entweder in der Requirement Analyse nicht alle Möglichkeiten berücksichtigt wurden oder aber daß sich Code- Leichen im System befinden – Re Use von Software.

Ersteres läßt sich durch Ergänzungen der Use Cases recht schnell beheben – no body is perfekt. Zweiteres sollte sehr genau untersucht werden. Erfahrungen in der Vergangenheit (Beispiel Ariane) haben gezeigt welche Auswirkungen so etwas haben kann.

Aufgrund der formalen Eigenschaften, die SDL besitzt, gibt es auch entsprechend formale Vorgehensweisen eine in SDL durchgeführte Spezifikation zu überprüfen. Nicht zuletzt aufgrund dieser Möglichkeiten gibt beispielsweise der TÜV Essen die Empfehlung bei der Spezifikation von sicherheitskritischen Systemen SDL zu verwenden.
Für eine vollständige Überprüfung eines in SDL spezifizierten Systems können folgende Überprüfungen statt finden:

- **Design-Verification „Statische Analyse“**
  Das SDL System wird lexikalisch, syntaktisch und anschließend semantisch überprüft.

- **Design-Verification „dynamische Analyse“**
  hierbei wird durch das Konzept der „State Space Exploration“ dynamisch die Lauffähigkeit des SDL Systems überprüft. Analysiert wird das System hinsichtlich:
  deadlocks, signal race conditions, implicit signal consumption, create errors, output errors, data operator errors, import errors, view errors.

- **Design-Validation „Simulator“**
  Mit Hilfe des Cbasic Codegenerators wird für die Hostumgebung eine Codegenerierung durchgeführt. Dieser instrumentierte C- Code wird für den Host Compiliert (Win NT oder Unix) und in einer Simulatorumgebung als Executable gestartet.
  Debugging erfolgt neben textueller Reports über MSCs oder direkt in SDL.

- **Design- Validation „Validator“**
  Inputs hierfür sind User defined Rules, MSCs oder SDL-Observers.
  Outputs für den Validation- Report sind MSC Diagramme, Log files, Backanotations nach SDL bzw. Test-Coverages. Wie bei der Simulation wird ein Executable gestartet.

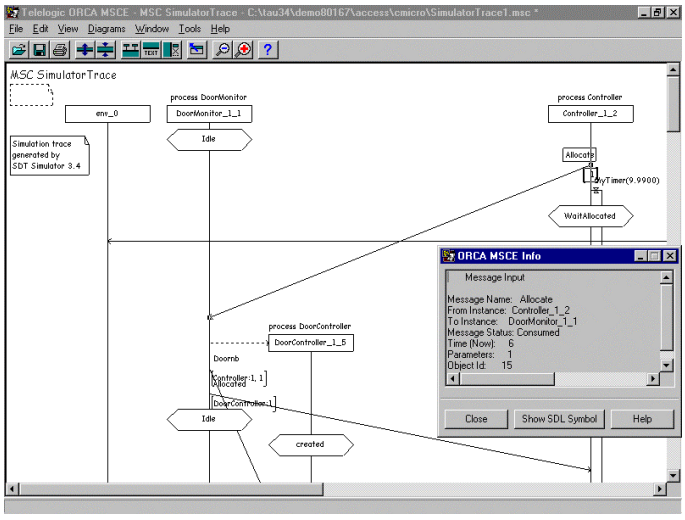
Verification: „You know that the system will work before implementation!“

Validation: „You know that you have the system you require before the implementation!“

MSCs (Message Sequence Charts)

Ebenfalls von großem praktische Wert ist die Simulation eines SDL Systems. Der dynamische Ablauf läßt sich zeitgleich in den SDL Diagrammen oder aber mit grafischem MSC Traces (Bild rechts) verfolgen.
Es können im SDL Diagramm grafisch Breakpoints gesetzt, Variablenwerte zur Laufzeit beobachtet oder über Time Stemps das Laufzeitverhalten untersucht werden.

Im MSC Trace lassen sich Interprozeß Kommunikation, Zustandsübergänge (Rauten) und allgemeine zeitliche Abläufe beobachten. Eine direkte Back Annotation vom MSC zur SDL Spezifikation ist per Mausklick möglich.
Eine weiterer Abstraktionsschritt verbirgt sich hinter der Notation TTCN (Tree and Tabular Combined Notation). Mit diesem internationalen Standard (ISO 9646-3) für Test Case Specification lassen sich modulare und parallele Testsuiten aus SDL heraus generieren. Schwerpunkte von TTCN sind ausführliche Conformance und Regressionstests.
Der hier zur Verfügung stehende Simulator kann über eine TTCN-SDL-Link Verbindung direkt gegen den SDL Simulator gestartet werden. Die sich hieraus ergebenden Erkenntnisse lassen keine Fragen mehr offen.



Resümee

UML und SDL bieten nicht nur eine Methode sondern inzwischen auch eine vollständige
Technik und ein Verfahren für die Entwicklung von embedded realtime systems.
Mit dem Fokus auf der Verwendung von internationalen Standards und Ihrer zweckdienlichen Kombination werden zukünftige Entwicklungen wesentlich vereinfacht.

A Suggestion for Real-Time Performance Annotations in Extension of the ROOM Concept

Peter Graubmann *Axel Klein*
Siemens AG, Corporate Technology, Dept. ZT SE 2, D-81730 Munich
☎ +49 (0)89 636-44181 ☎ +49 (0)89 636-41006
peter.graubmann@mchp.siemens.de axel.klein@mchp.siemens.de

The ROOM Concept for Real-Time Object-Oriented Modeling [1] provides an attractive approach to the systematic and consistent design and implementation of large embedded systems. Through the object-oriented paradigm it allows high-level abstractions for requirements and architecture specifications as well as consistent refinement down to the implementation level and encourages re-use of design elements on all levels. The well-integrated, semantically complete and unambiguous modeling formalism avoids the usual variety of independent, yet informally related specifications and the resulting consistency problems. Thus it makes possible to execute the models for early requirements or design validation, and to automatically generate target code from the model. With a single model for the entire system life cycle, it is also much easier to maintain and evolve the architectural view during implementation development and successive redesign. The behavioral specification through state charts lends itself well to the reactive behavior of embedded real-time systems, and the graphic-based model description building on well-established UML terminology [2] provides an intuitive means to specify and communicate the system architecture and design.

Starting to explore the suitability of the ROOM approach for various design scenarios in the practice of industrial development, we have come across some open questions and desired features which are (in our current understanding) not yet sufficiently answered by the ROOM method and its current realization. In the subsequent paragraphs we present such issues and discuss some ideas how to deal with them, arising from current approaches in other specification methods used in the telecommunications domain.

Component-Based Design

In order to be executable, the behavior of a design component in ROOM must be modeled as a state machine (ROOMChart). In a behavioral simulation, sequence diagrams can then automatically be generated and compared to previously defined sequence charts specifying the required (external) behavior. The modeling of a state machine exposing the required behavior is a major creative act in a top-down design process. It does not only describe *what* the component does, but in addition *how* this is achieved by defining appropriate internal states and transitions which generate the desired reaction on external events. In other words, even in the first step of an incrementally refined state machine design, some design decisions must be taken which extend the original functional requirements specification.

Assuming a development scenario where we employ already existing "off-the-shelf" components, however, we want to use these components as "black boxes", i.e. we know their external behavior, but not their internal implementation. Also, for a legacy or commercially supplied component a specification of the internal state machine behavior is usually not available. What must be available for the component to be re-usable, however, is a description of the external (observable) behavior, which in its semantic content is more or less equivalent to a collection of sequence charts (together with the structural definition of the protocols involved). Note that a single UML sequence diagram is generally not sufficient when e.g. the component participates in different use cases.

In contrast to the UML (and ROOM) sequence diagrams, the standardized ITU-T Message Sequence Charts [3] include powerful composition/decomposition features to express the complete interaction pattern of design components within a single modeling construct. Such a specification should be well suitable as starting point to *generate* a functionally equivalent state chart for a black-box component rather than explicitly designing one. In this way, one could produce an executable functional representative of a component as it is needed for early system design validation through host simulation, when the component implementation itself is not available on the host.

Performance Validation

While in general IT systems performance is a non-functional issue of service quality, real-time systems are different in that they demand timeliness as a functional requirement. Functions which start or finish too late (or, sometimes even too early) may be worthless, if not harmful and thus must be regarded as faulty. The validation of real-time performance behavior is therefore just as important as that of other functional behavior, and should be done as early as possible in the design process to avoid critical design faults which are discovered only after late tests on the target.

Such early validation requires some sort of performance model for analytic calculations or simulation of the system's performance behavior. Unless this performance model is integrated with the functional model, there arises a consistency problem between the different models during design refinement and evolvement, which lessens the original advantage of the life-spanning ROOM design model. Thus the question is, if and how real-time performance can be evaluated (or, at least estimated) within the functional design model itself.

A possible approach would be to add time annotations to the behavioral model elements, i.e. specify assumed real-time behavior for transitions, entry and exit actions and communication actions. These assertions could then be evaluated in the course of a functional simulation to provide information on the dynamic system performance. To be useful in practice, we would need not only singular time values, but also time intervals and stochastic distributions to describe and simulate the performance behavior. This must go together with corresponding evaluation arithmetic to calculate relevant performance indices, such as average, minimum, maximum and variances of execution time. Verilog has recently introduced an extension of this kind to executable SDL specifications in ObjectGeode [4].

Performance Requirements

In the context of performance simulation, the crucial question is not only how to specify the behavior requirements but also how to countercheck the simulation runs against the requirements. Time annotations associated with state chart transitions, as discussed in the previous section, may well serve during different states of the design as both requirements or first estimates for the duration of a transition. Annotations to state transitions provide an intra-actor behavior description. Yet, we are interested at least as much in the validation of the inter-actor behavior.

Sequence charts are a well suited means to describe this inter-actor behavior. In particular, Message Sequence Charts (MSCs) as defined by the ITU-T [3,5] with their extension of compositional mechanisms (HMSCs) are not only capable to describe some individual behavior sequences but allow a complete description of the interaction of one actor with it's communication partners. Furthermore, in the current study period, MSCs are enhanced by rather rich time annotation facilities which will be formally defined (see the preliminary MSC2000 document [6]). These time annotations, which will comprise the possibilities provided by UML Sequence Charts [7], are based on an event trace semantics with associated time intervals between events. Thus, MSCs can be employed to specify the temporal aspect of the external behavior of an actor.

During a simulation run, the time annotations provided for the state transitions can be used to calculate the time intervals elapsing between the various incoming and outgoing messages for each actor. From this information, it is not difficult to generate the MSC with the related time annotations mirroring the simulation run. A countercheck with the MSC behavior specification for this actor shows whether the timing requirements are met or not.

This scenario is still somewhat simplified, however, it seems to be feasible and may provide a helpful instrument for the system designer. It is our intention to investigate not only into the mapping of state transition time annotations to MSCs and of these MSCs into the collection of MSC specifications but also to analyze the time annotations provided for MSC2000.

References

[1] Selic, B.; Gullekson, G.; Ward, P.: Real-Time Object-Oriented Modeling. John Wiley & Sons, New York, NY, 1994

[2] Selic, B.; Rumbaugh, J.: Using UML for Modeling Complex Real-Time Systems. ObjecTime and Rational White Paper, 1998.

[3] ITU-T Recommendation Z.120 (1996). Message Sequence Chart (MSC). Geneva, 1996.

[4] Roux, J.-L.: SDL Performance Analysis with ObjectGEODE. Technical paper presented at the "Performance and Time in SDL & MSC" Workshop, University of Erlangen, Germany, Feb 98

[5] ITU-T Recommendation Z.120 (1998). Message Sequence Chart (MSC), Annex B. Geneva, 1998.

[6] ITU-T Recommendation Z.120 (MSC2000). Message Sequence Chart (MSC). Temp. Doc., Geneva, 1999.

[7] Douglass, B. P.: Real-Time UML. Developing Efficient Objects for Embedded Systems. Addison-Wesley, Reading, Massachusetts, 1998.

OMOS

Objektorientierte Modellierung von Embedded-Software

W. Hermsen und K.J. Neumann, Robert Bosch GmbH, Frankfurt

Kurzfassung
Die Entwicklung von Kfz-Steuergerätesoftware ist in hohem Maße durch kundenspezifische Anforderungen an Funktionalität, Echtzeitfähigkeit und Speicherbedarf gekennzeichnet. Die Anforderungen variieren dabei nicht nur für verschiedene Kfz-Hersteller sondern auch für unterschiedliche Projekte eines Herstellers. Zudem ist die kunden- und projektspezifische Software im Hinblick auf Kosten, Zeit und Qualität effizient zu entwickeln.

Um diesen Anforderungen eingebetteter Echtzeitsysteme auch in Zukunft gerecht zu werden, bedarf es einer strukturierten und leicht erweiterbaren Software-Architektur zur Beschreibung ganzer Produktfamilien. Diese muß insbesondere die Variantenbildung und Wiederverwendung von Kode effektiv unterstützen, ohne dabei unnötigen Laufzeit- und Speicherplatzbedarf zu verursachen. Das in diesem Artikel vorgestellte Modellierungskonzept für Steuergerätesoftware (OMOS[1]) nutzt hierfür objektorientierte Techniken unter gezielter Vermeidung von laufzeit- und speicherplatzintensiven Mechanismen objektorientierter Sprachen, wenn diese nicht notwendig sind.

1 Einleitung

Die Anforderungen an die Funktionalität und Sicherheit von Steuergerätesystemen im Kraftfahrzeug nimmt stetig zu. Dies hat ein signifikantes Wachstum des Softwareanteils in diesen Systemen und damit auch der Komplexität der Software zur Folge. Zudem stellt sich, insbesondere für die Zulieferindustrie von Kfz-Steuergeräten, das Problem, die ebenfalls zunehmende Variantenvielfalt der Steuergerätesoftware für die verschiedenen Automobilhersteller und deren Modelle mit ihren unterschiedlichen Ausstattungen bzgl. der Steuergerätefunktionalität auch zukünftig beherrschbar zu halten. Diesen Anforderungen kann nur durch den Einsatz einer strukturierten und erweiterbaren Software-Architektur sowie einer effizienten Softwareentwicklung begegnet werden, die es ermöglicht, schnell und kostengünstig neue Software zu erstellen, bei gleichzeitiger Verbesserung der Softwarequalität [6].

Die Erreichbarkeit dieser Ziele verlangt ein hohes Maß an Wiederverwendbarkeit von existierenden Komponenten und Kode, wie sie gerade durch die Vererbung in der objektorientierte Technologie unterstützt wird [7]. Wiederverwendung bedeutet dabei, daß bestehende Komponenten, die in einem bestimmten Kontext schon eingesetzt wurden, in einem neuen Kontext, z.B. anderen Projekten, verwendet werden. Durch Wiederverwendung von getesteter und bereits eingesetzter Software steigt die Qualität, d.h. Komponenten, die mehrfach eingesetzt werden, werden immer stabiler. Zudem führt die Wiederverwendung zu Zeit- und Kostenersparnis bei der Softwareerstellung. Darüber hinaus ermöglicht die Vererbung Variantenbildung und Austauschbarkeit von Komponenten. Ziel der Variantenbildung ist es, aufbauend auf bestehenden Komponenten, neue Komponenten zu definieren, wobei einige Details geändert oder ergänzt werden. Durch eine „mechanisierte Variantenbildung“ erhöht sich die Reaktionsgeschwindigkeit auf neue Anforderungen. Die Austauschbarkeit ermöglicht schließlich, an Stelle einer Komponente auch Varianten dieser Komponente zu benutzen, ohne Änderungen an der Kodierung der nicht ausgetauschten Komponenten vornehmen zu müssen.

Bei der Entwicklung von Steuergerätesoftware sind aufgrund der Forderungen an Laufzeit und Speicherplatz allerdings nicht alle Konzepte der objektorientierten Technologie direkt einsetzbar, z.B. Mehrfachvererbung. Die konsequente Fortführung der objektorientierten Modellierung durch Implementierung in einer objektorientierten Sprache, wie C++ (s. [8]), wird nicht zuletzt aus diesem Grunde nicht betrieben. Zudem sind objektorientierte Sprachen für Echtzeitsysteme, wie z.B. Embedded C++ (s. [2]), derzeit noch zu wenig ausgereift, so daß für die Implementierung meist auf die Programmiersprache C zurückgegriffen wird [3].

[1] OMOS steht für **O**bjektorientiertes **MO**dellierungskonzept für **S**teuergerätesoftware

Diesen einschränkenden Bedingungen trägt das Modellierungskonzept OMOS, das in diesem Artikel vorgestellt wird, Rechnung. OMOS ist ein auf die Entwicklung von Steuergeräte-Software zugeschnittenes objektorientiertes Modellierungskonzept, das die Variantenbildung durch Vererbung unterstützt und flexiblen und wiederverwendbaren Kode liefert, der die harten Anforderungen an Speicherplatz und Laufzeit von Steuergerätesoftware im Kfz berücksichtigt. Aus diesem Grund beschränkt sich OMOS auf die Teilmenge der Konzepte der objektorientierten Modellierung, die eine optimierte Umsetzung in C-Kode ermöglicht. Die aus der Objektorientierung bekannten Konzepte: Objekt, Klasse und Vererbung sowie Inline- und virtuelle Methoden, etc., werden hierfür in C nachgebildet. Diese Nachbildung ist lohnenswert, weil dadurch Mechanismen bereitgestellt werden, die eine Wiederverwendung und Variantenbildung aktiv unterstützen.

2 OMOS Konzept

In der objektorientierten Modellierung wird eine Software durch eine Menge miteinander kommunizierender Objekte modelliert [7]. Jedes Objekt kann Attribute und Methoden besitzen. Die Attribute beschreiben die internen Daten eines Objekts, in denen der Zustand des Objekts festgehalten wird. Die Methoden realisieren die Funktion eines Objekts und greifen auf dessen Daten zu. Gleichartige Objekte werden zu Klassen zusammengefaßt, die abstrakte Datentypen darstellen und können quasi als Schablonen für die Objekterzeugung betrachtet werden. Eine Klasse legt dabei für ihre Objekte eine einheitliche Struktur bestehend aus Attributen und Beziehungen zu anderen Klassen sowie ein einheitliches Verhalten, das durch die Methoden beschrieben wird, fest. Jedes Objekt ist Instanz einer Klasse und besitzt damit alle durch die Klasse deklarierten Attribute, Methoden und Beziehungen.

In OMOS werden zwei Arten von Klassen unterschieden: 1-Klassen und N-Klassen. N-Klassen entsprechen dem aus der Objektorientierung bekannten Klassenbegriff, d.h. sie können insbesondere beliebig oft instanziert werden. 1-Klassen hingegen sind Klassen, von denen in einer konkreten Systemkonfiguration maximal eine einzige Instanz existiert. Diese Eigenschaft von 1-Klassen bietet bei der Realisierung in C gegenüber N-Klassen bzgl. Laufzeit und Speicherplatzbedarf eine großes Optimierungspotential. So lassen sich beispielsweise in C sämtliche Attribute von 1-Klassen-Objekten als eigenständige Variablen realisieren, so daß eine Dereferenzierung der Objekte sowie die Übergabe von Objektreferenzen an Methoden zur Laufzeit entfällt. Die Unterscheidung von 1-Klassen und N-Klassen zur Ausnutzung dieses Optimierungspotentials ist um so begründeter, wenn man berücksichtigt, daß die Zahl der 1-Klassen in der Kfz-Steuerungssoftware sehr groß ist (z.B. Motor, Getriebe, Fahrpedal).

Zu den von OMOS unterstützten Beziehungen gehören die Komposition, die Kommunikation und die Variantenbeziehung basierend auf der Vererbung.

Durch Kompositionsbeziehungen werden mehrere "einfachere" Objekte zu einem "komplexeren" bzw. abstrakteren Objekt zusammengefaßt. Im Zuge eines Top-Down-Entwurfs drückt die Komposition die Zerlegung oder auch Verfeinerung eines Besitzobjekts in Teilobjekte aus, und damit die Zerlegung einer komplexeren Aufgabe in mehrere einfachere Teilaufgaben. Die Teilobjekte hängen dabei von der Existenz ihres Besitzobjekts ab, d.h. mit dem Vorhandensein eines Besitzobjekts sind auch alle seine Teilobjekte in einem Modell präsent. Selbstverständlich können auch Teilobjekte selbst wieder verfeinert werden. Durch die Kompositionsbeziehungen wird somit die Menge aller Objekte in Form eines Teilobjektbaums festgelegt, aus dem ein Gesamtsystem besteht. Die Wurzel dieses Teilobjektbaums bildet die Instanz einer durch den Stereotype "Root" ausgezeichnete 1-Klasse.

Über Kommunikationsbeziehungen wird festgelegt, daß sich Teilobjekte gegenseitig benutzen können. Dabei sendet ein Teilobjekt (Senderobjekt) an ein anderes Teilobjekt (Empfängerobjekt) quasi eine Nachricht, indem das Senderobjekt eine Methode des Empfängerobjekts aufruft. Durch eine Kommunikationsbeziehung wird jeder Instanz der Senderklasse ermöglicht, auf alle Methoden eines Empfängerobjekts zuzugreifen.

Die Variantenbeziehung wird durch eine Vererbungsbeziehung im objektorientierten Sinne ausgedrückt. Eine neue (Klassen-) Variante wird als Unterklasse einer allgemeineren Klassenvariante, der Oberklasse, modelliert. Hierdurch erbt die neue Variante die Struktur und das Verhalten, und damit auch die Schnittstelle der allgemeineren Variante. Insbesondere ist es durch die Vererbung der Schnittstelle möglich, Instanzen der neuen Variante überall dort zu verwenden, wo Instanzen der Oberklasse verwendet werden. Die neue Klassenvariante kann darüber hinaus um zusätzliche Attribute, Methoden und Beziehungen erweitert werden sowie die Realisierung von geerbten

Methoden variantenspezifisch überschreiben. Einmal geerbte Eigenschaften können jedoch nicht wieder abgegeben werden, da hierdurch die Konformität zur Schnittstelle der Oberklasse verletzt würde. Durch die Variantenbeziehung ergibt sich, daß durch ein Klassenmodell quasi ein Pool von möglichen Varianten der Steuerungssoftware beschrieben wird.

Abbildung 2-1 stellt den grundsätzlichen Ablauf der Entwicklung von Steuergerätesoftware unter Verwendung von OMOS vor. Zunächst ist die Steuerungssoftware durch ein Klassenmodell zu beschreiben, das alle Klassen und deren Beziehungen untereinander enthält, die für die Umsetzung der Steuerungsaufgabe benötigt werden. Diese erfolgt weitgehend UML-konform (Unified Modeling Language) mit Hilfe des Tools Rational Rose [1]. Anschließend wird durch OMOS für jede modellierte Klasse je eine Deklarations- und Definitionsdatei generiert. Bei diesen Dateien handelt es sich um C-Koderahmen, die durch den Systementwickler mit den eigentlichen Steuerungsfunktionen in Form von C-Kode auszufüllen sind. Die beiden Schritte Anpassung des Klassenmodells und Koderahmengenerierung können beliebig häufig durchlaufen werden. Dabei wird der bereits aus einem vorausgehenden Iterationsschritt existierende Methodenkode bei der Koderahmengenerierung unverändert übernommen (Abgleich), sofern die Methode noch existiert.

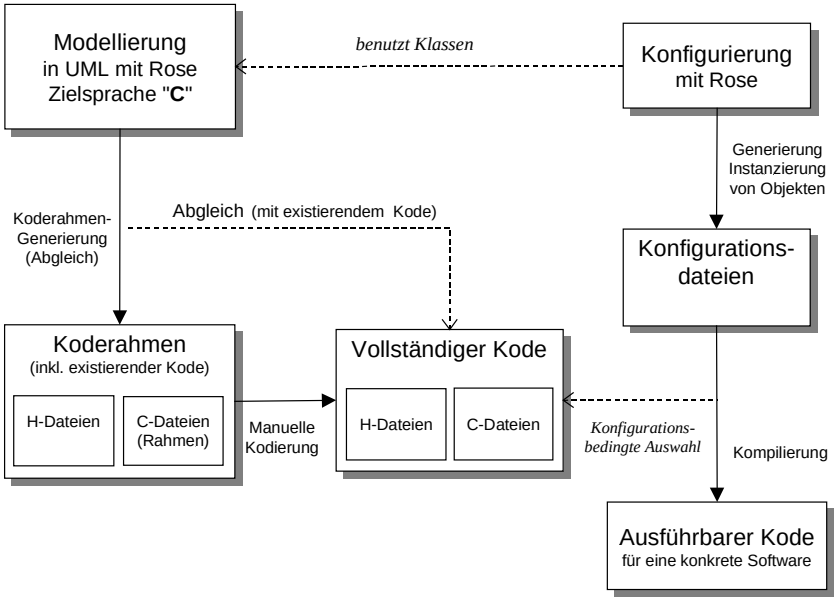


Abbildung 2-1 Entwicklung von Steuergerätesoftware mit OMOS

Enthält ein so erstelltes Klassenmodell Varianten, so wird hierdurch nicht nur eine einzige sondern eine Menge von Softwarevarianten beschrieben. Eine konkrete Softwarevariante wird dann durch eine Konfiguration festgelegt. Für eine Konfiguration generiert OMOS schließlich alle konfigurationsabhängigen Dateien. In diesen werden sämtliche konfigurierten Objekte angelegt und ggf. initialisiert. Zusammen mit den vollständig kodierten Klassendateien, die in der konkreten Konfiguration der Software verwendet werden, ergibt sich der compilierbare Kode.

3 Zusammenfassung

Die zunehmenden Anforderungen an die Entwicklung von Steuergerätesoftware und die Variantenbildung erfordern eine effiziente Softwareentwicklung und eine strukturierte und erweiterbare Software-Architektur. Das vorgestellte Modellierungskonzept OMOS basiert auf objektorientierten Ansätzen, die sich bzgl. Laufzeit und Speicherbedarf effizient in C-Kode, gemäß dem ANSI-Standard, umsetzen lassen. Die von OMOS generierten und manuell erweiterten Dateien sind "normale" C-Quelldateien, die sich problemlos mit anderer Software, die nicht mit OMOS entwickelt wurde, integrieren lassen.

OMOS bildet die objektorientierten Konzepte: Objekt, Klasse und Vererbung in C nach. Hierdurch ist es möglich, eine Software-Architektur, bestehend aus miteinander in Beziehung stehenden Klassen, festzulegen. Jede Klasse definiert dabei eine Schnittstelle für den Zugriff auf ihre Instanzen bzw. Objekte, aus denen schließlich die Steuergerätesoftware besteht. Durch die Vererbung wird sowohl die Variantenbildung und Austauschbarkeit von Komponenten, wie auch die Wiederverwendung von existierendem Kode in geeigneter Weise unterstützt. Darüber hinaus bildet die Variantenbildung mittels Vererbung eine hervorragende Basis für die Entwicklung verteilter Steuergerätesoftware [4, 5].

Die Modellierung ist dabei weitgehend unabhängig von der gewählten Zielsprache für die Koderahmengenerierung. Diese Unabhängigkeit wird zusätzlich dadurch unterstützt, daß sich auch die Kodierung selbst an eine objektorientierte Programmierung anlehnt. Bei einem späteren Umstieg auf eine andere Zielsprache, wie z.B. Embedded C++, ist somit sowohl bei der Modellierung als auch bei der Kodierung nur mit einem geringen Änderungsaufwand zu rechnen.

Hinsichtlich der Produktivitätssteigerung, die durch eine projektübergreifende Wiederverwendung von Komponenten und Kode zu erwarten ist, kann derzeit noch keine Aussage gemacht werden, da in den laufenden Projekten der Wiederverwendungsprozeß von OMOS-basierter Software noch in den Anfängen liegt.

4 Literatur

[1] Booch, G., Jacobson, I. und Rumbaugh, J.: "The Unified Modeling Language for Object-Oriented Development". Version 1.0, Rational Software Corporation, 1997.

[2] Embedded C++, 1998. Homepage: http://www.caravan.net/ec2plus/

[3] Kernighan, B.W. und Ritchie, D.M.: "Programmieren in C". 2. Ausgabe, Carl Hanser Verlag, München, Wien, 1990.

[4] Kiencke, U. und Neumann, K.J.: "Modellierung und Partitionierung verteilter Echtzeitanwendungen". VDI-Berichte Nr. 1415 (1998), Elektronik im Kraftfahrzeug, S. 691-707.

[5] Kiencke, U., Kytölä, T und Neumann, K.J.: "Hierarchical Partitioning of Real-Time Applications based on Object-Oriented Modeling". SAE Convergence '96, Dearborn, MI, USA, Oktober 1998.

[6] Kytölä, T., Mathony, H.-J. und Münich, A.: "Objektorientierte Software-Entwicklung für Karosserieelektronik-Anwendungen". VDI-Berichte Nr. 1415 (1998), Elektronik im Kraftfahrzeug, S. 571-595.

[7] Rumbaugh, J. et al.: "Object-Oriented Modeling and Design". Prentice-Hall, Englewood Cliffs, NJ, USA, 1991.

[8] Stroustrup, B.: "Die C++ Programmiersprache". 2. Auflage, Addison-Wesley, Bonn, München, Paris, 1992.

Objektorientierte Entwicklung eingebetteter Echtzeitsysteme im Automobil

Petra Geretschläger, Dr. Peter Hofmann

DaimlerChrysler AG
Forschung und Technologie, Stuttgart

1 Einführung

Was hat die Konstruktion von Automobilen mit der Entwicklung von Software gemeinsam? Vordergründig betrachtet nicht viel, Automobile sind Produkte traditioneller Ingenieursarbeit: Karosseriebauer gestalten eine sowohl optisch ansprechende als auch technisch ausgeklügelte Form. In dieses Gebilde aus Blech und Stahl werden nach und nach alle relevanten Komponenten integriert. Angefangen beim Kabelbaum über die Lichtanlage, die Sitze bis hin zum Motor.

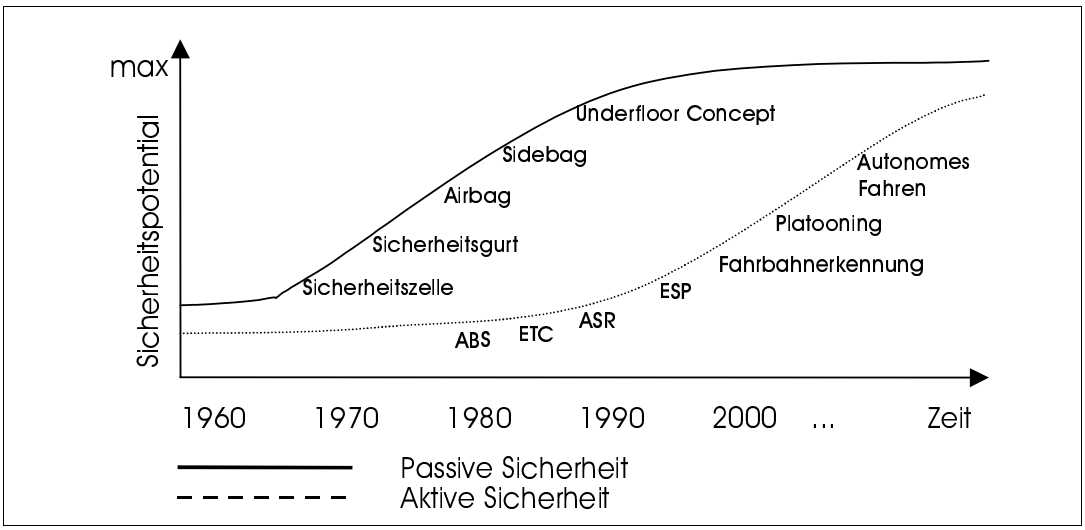


Abb. 1: Passive- und Aktive-Sicherheit

Heute stehen wir an einem Wendepunkt: Immer häufiger werden mechanische und elektrische Komponenten durch elektronische Systeme ergänzt oder ersetzt. Mit dem Ziel die Sicherheit kontinuierlich zu erhöhen und den Fahrkomfort zu verbessern. Ersteres kann beispielsweise durch den Einsatz eines Anti-Blockiersystems (ABS) oder eines elektronischen Stabilitätsprogramms (ESP) erreicht werden. Telefon, Radio mit CD-Player und Navigationssystem sind nur einige Beispiele für die Verbesserung des Fahrkomforts, die mittlerweile nicht mehr aus den Mittel- und Oberklassefahrzeugen wegzudenken sind.

Vor allem im Bereich der Sicherheit konnte, wie in Abbildung 1 dargstellt, ein beinahe optimales Sicherheitspotential erreicht werden. Insbesondere im Bereich der passiven Sicherheit konnte die Unfallgefahr für die Autoinsassen durch die Einführung von Systemen, wie beispielsweise des Airbags, des Seitenairbags oder den Seitenaufprallschutz deutlich reduziert werden. Verbesserungen in diesem Bereich sind nur noch mit sehr hohem technischen Aufwand realisierbar, daher sind zukünftige Verbesserungen vor allem im Bereich der aktiven Sicherheit zu finden. Durch den Einsatz der Elektronik können hier neue Systeme entwickelt werden, die den Fahrer aktiv unterstützen und so Unfälle vermeiden können, wie beispielsweise ABS (Anti-Blockiersystem), ETC (Elektronische Traktionskontrolle) und ESP (Elektronisches Stabilitätsprogramm).

Diese neuen Aufgaben können mit rein mechanischen oder elektrischen Systemen, wie in der Vergangenheit, nicht mehr gelöst werden. Für derartige Entwicklungen werden sehr komplexe elektronische Systeme benötigt, die einen hohen Anteil an Steuerungssoftware enthalten. Die Erstellung dieser Steuerungssoftware erfordert einen veränderten Entwicklungsprozess, der die Besonderheiten der Softwareentwicklung berücksichtigt.

2 Objektorientierte Systementwicklung

Die oben beschriebenen elektronischen Systeme werden in der Informationstechnik als reaktive eingebettete Echtsysteme (EES) bezeichnet, da sie steuernd und regelnd in ihre technische Umwelt eingreifen. Die Charakteristiken dieser Systeme - komplexes Verhalten, harte Echtzeitanforderungen und Hardware-Beschränkungen (8-16 Bit CPU, geringer Speicher) - müssen im Entwicklungsprozess berücksichtigt werden und führen zu sehr spezifischen Anforderungen.

Zusätzlich findet ein Wandel von homogenen Systemen hinzu heterogenen Systemen statt. Wurden bisher entweder regelungs- oder steuerungstechnische Systeme eingesetzt, so werden zukünftig die unterschiedlichen Systemtypen eng miteinander kooperieren und kommunizieren.

Mit zunehmender Anzahl von elektronischen Systemen im Fahrzeug ergibt sich die Notwendigkeit, diese ursprünglich autonomen Systeme zu einem Gesamtkonzept zu integrieren.

Neben diesen technischen Anforderungen gewinnen nichtfunktionale Anforderungen, wie Wiederverwendung, Erweiterbarkeit der Funktionen und Wartbarkeit immer mehr an Bedeutung, da sie maßgeblich zur Qualität der zu entwickelnden Produkte beitragen. Insbesondere dynamisch nachladbare Funktionen, die vergleichbar den Plug- and Play-Konzept nur noch ins Fahrzeug geladen werden, tragen damit zur Differenzierung von anderen Wettebewerbern bei.

Insgesamt zeigt sich, daß die bisher eingesetzten Entwicklungsmethoden weder die vielfältigen Anforderungen erfüllen können, noch die steigende Komplexität der elektronischen Systeme beherrschen. Aus diesem Grund untersucht die DaimlerChryler Forschung inwieweit die Anwendung objektorientierter Konzepte und Technologien neue Lösungswege auf-

zeigen. Ziel der Aktivitäten ist die Erstellung einer auf Objektorientierung basierenden Entwicklungsmethode speziell für automobile Systeme.

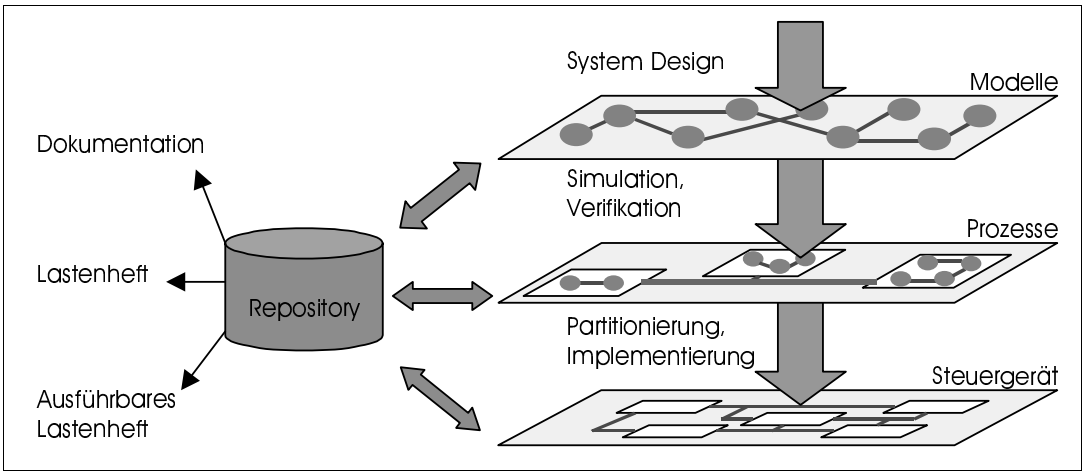


Abb. 2: **Schematischer Entwicklungsprozess**

Die Idee dieser Methode basiert auf der Strukturierung eines eingebetteten Systems auf der Basis kooperierender Teilsysteme. Jedes dieser Teilsysteme wird als sogenanntes Modellobjekt aufgefaßt. Modellobjekte haben ein klar definiertes Verhalten, das über mehrere Hierarchieebenen hinweg und mit Hilfe unterschiedlicher Beschreibungssprachen spezifiziert wird. Für die Beschreibung des Verhaltens wird gezielt auf objektorientierte Beschreibungstechniken zurückgegegriffen, wie sie beispielsweise in UML angeboten werden. Da diese Beschreibungstechniken bei weitem nicht alle notwendigen Aspekte abdecken, muß ein Konzept gefunden werden, das es erlaubt, daß Modelle aus unterschiedlichen Beschreibungssprachen kooperativ zusammenarbeiten können.

Ausgangspunkt der Entwicklung ist die Idee eines System, das wie in Abbildung 2 dargestellt, in der Regel in mehrere miteinander kommunizierende Teilsysteme zergliedert werden kann. Jedes dieser so entstandenen Teilsysteme kann wiederum in eine Menge von Teilsysteme mit zugehörigen Interaktionen zerlegt werden. Dieser Schritt wird solange wiederholt bis nur noch atomare Objekte übrig bleiben, bei denen eine weitere Zergliederung keinen Sinn ergibt.

Die Kommunikation zwischen den einzelnen Teilsystemen spielt bei diesem Prozess eine zentrale Rolle. Nur wenn es gelingt, diese Kommunikationsbeziehungen über sämtliche Hierarchieebenen hinweg konsistent zu beschreiben, kann die Funktion des Gesamtsystems gewährleistet werden. Daher wird besonderer Wert auf die Spezifikation der Schnittstellen der einzelnen Teilsysteme gelegt, da diese Schnittstellen die Kommunikation erst ermögli-

chen. Der eben beschriebene Prozess des hierarchischen Dekomposition ist in der nachfolgenden Abbildung schematisch dargestellt.

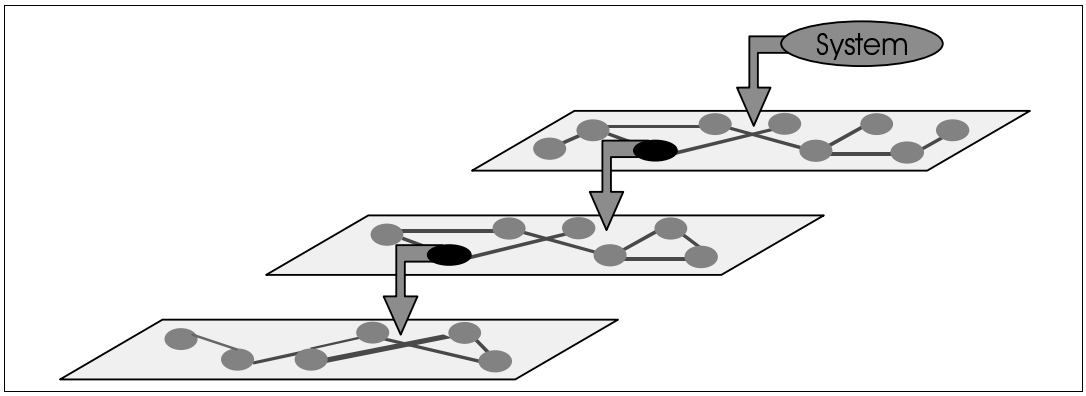


Abb. 3: Hierachische Dekomposition des Systems

Nach der Zerlegung des Systems erfolgt nun die Modellierung der einzelnen Teilsysteme, sowie die Präzisierung der bisher nur sehr grobgranular beschriebenen Kommunikation. Diese kontinuierlich präziser werdenden Kommunikationsinformationen werden Schritt für Schritt an die darüber liegenden Hierarchieebenen weitergegeben.

Für die Modellierung der Teilsysteme werden unterschiedliche Modellierungssprachen benötigt, so daß jeder Aspekt des Systems mit der geeigneten Sprache beschrieben werden kann. Aufgrund ihrer umfangreichen Notation und der Standardisierung durch die OMG bietet sich die Unified Modeling Language als Grundlage für eine zukünftige automobilsspezifische Sprache an. Die Vorteile einer solchen Modellierungssprache gegenüber der bisher eingesetzten natürlichen Sprache liegen in der konsistenten und eindeutigen Beschreibung der Funktionalität und der Möglichkeit das bisherige Lastenheft um ausführbare Modelle zu ergänzen (ausführbares Lastenheft).

Um einen durchgängigen Entwicklungsprozess zu erreichen, wird eine zentrale Verwaltung der Entwicklungsdaten benötigt, die es ermöglicht, zu jedem Zeitpunkt der Entwicklung auf alle bereits erstellten Daten zurückzugreifen, diese zu ergänzen oder zu ändern. In unserem Konzept wird ein Repository vorgeschlagen, das ein universelles Datenmodell bereitstellt in dem kooperativ zusammenarbeitende Modelle unabhängig von ihrer Modellierungssprache abgelegt werden können. Für die Implementierung wird eine objektorientierte Datenbank eingesetzt, die es ermöglicht die einzelnen Modelle in Form von gekapselten Objekten mit Meta-Informationen zu speichern. Ein solches Repository kann in nachfolgenden Projekten auch als Basis für wiederverwendbare Komponenten dienen.

Nach den oben beschriebenen Schritten ist das zu entwickelnde System vollständig modellbasiert beschrieben. Die Modelle sind in einem Repository abgelegt und können nun für weitere Entwurfsschritte verwendet werden.

In der nun folgenden Phase des Entwicklungsprozesses geht es darum, die Qualität der formalen Modelle sicherzustellen. Hierzu werden Simulations- und Verifikationstechniken, wie

beispielsweise Model Checking eingesetzt. Die Generierung von entsprechenden Testfällen zählt ebenfalls zu den notwendigen Aktivitäten in diesen Phasen.

Bereits heute stehen Werkzeuge zur Verfügung, die aus den formalen Modellen Simulationscode erzeugen können. Treten während der Simulation Fehler oder Mängel in den beschriebenen Funktionen auf, so müssen diese in den formalen Modellen des Repositorys gekennzeichnet werden. Die damit verbundenen rückwärts gerichteten Entwicklungsschritte müssen von einem Werkzeug überwacht und koordiniert werden, damit die Konsistenz der Entwurfsdaten im Repository jederzeit gewährleistet werden kann.

In der sich anschließenden Partitionierung und Implementierung wird aus der formalen Spezifikation der eigentliche Code erstellt und auf die Steuergeräte verteilt. Zeichnete sich das zu entwickelnde Systeme in den ersten Phasen der Entwicklung noch durch komplexe hierarchische Strukturen aus, so ist auf der Ebene der Steuergeräte nur noch ein Hierarchieebene vorhanden.

3 Roadmap

Die oben beschriebenen Entwicklungsmethode auf der Basis objektorientierter Konzepte, liefert die Grundlage für einen durchgängigen System-Design Prozess. Sie erlaubt die Erweiterung existierender Ansätze um automobilspezifische Beschreibungsmöglichkeiten, die hinsichtlich der Durchgängigkeit, der Erweiterbarkeit und der Wiederverwendung neue Akzente setzt. Darüber hinaus ermöglicht sie die Spezifikation von verteilten kooperierenden Systemen, die mehr und mehr Einzug ins Fahrzeug halten, analog zu dem Trend „das Netzwerk ist der Computer“ der bereits vor einigen Jahren in der IT-Welt zu beobachten war. Generell werden sich im Bereich der eingebetteten Systemen ähnliche Entwicklungen abzeichnen wie in klassischen IT-Welt.

Die Aufgabe besteht nun einerseits in der Bereitstellung adäquater Beschreibungstechniken und Werkzeuge für die einzelnen Subsysteme in Verbindung mit der Beherrschung des gesamten Entwicklungsprozesses. Brüche in der Werkzeugkette, wie sie heute an der Tagesordnung sind, können zukünftig nicht mehr toleriert werden, aufgrund der Tatsache, daß dadurch nicht nur die Effizienz und die Qualität des Entwurfs leiden, sondern letztendlich der Einsatz des Entwicklungsprozesses an der Akzeptanz der Anwender scheitern wird.

Hier liegt eine große Herausforderung für die Industrie und die Hochschulen. Es muß gemeinsam gelingen, die heute nicht optimalen Werkzeuge und die inhomogene Methodenlandschaft gemeinsam so zu verändern, daß ein Mehrwert für die Ingenieure herauskommt, um letztendlich für den Kunden optimale Produkte zu schaffen.

Wir haben aus diesem Grund begonnen, unsere Aktivitäten in diesem Bereich in der DaimlerChrysler Forschung zu verstärken und beginnen ein Konsortium aus kompetenten Partnern aufzubauen, indem wir enge Kooperationen mit einzelnen Hochschulen eingegangen

sind. Ziel dieser Aktivität ist die Entwicklung einer passenden Modellierungssprache - Automotive-UML.

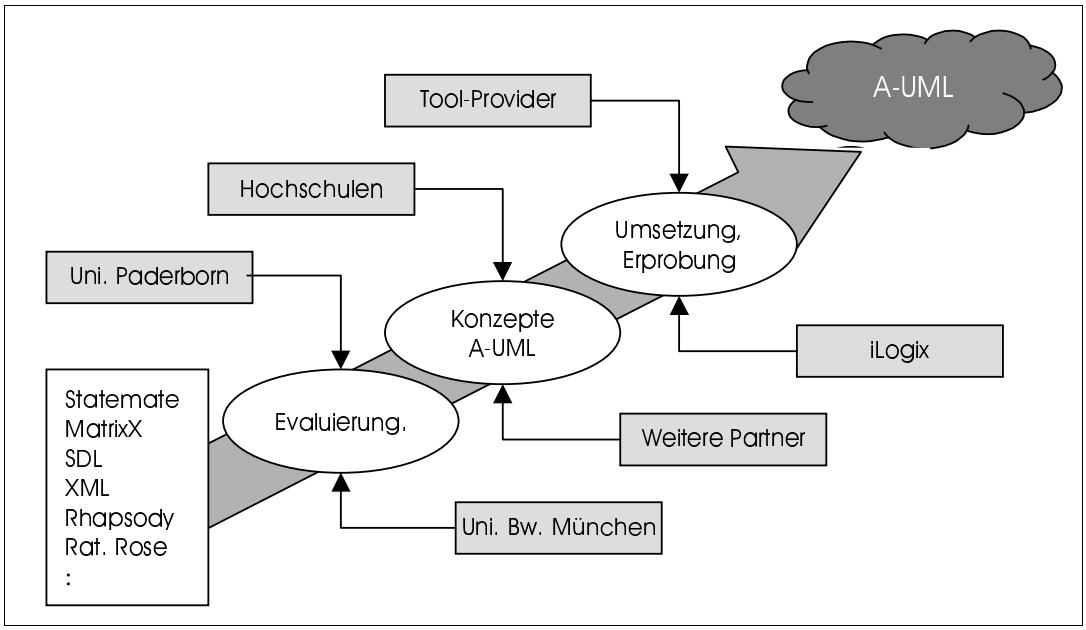


Abb. 4: **Roadmapzur Entwicklung von Automotive UML**

Darunter verstehen wir nicht nur eine neue Variante von UML wie beispielsweise UML-RT, sondern eine auf die speziellen Bedürfnisse der automobilen Systementwicklung abgestimmte Menge an Werkzeugen und Methoden, die den Besonderheiten echtzeitfähiger Systeme in gleicher Weise gerecht werden sollen, wie den mehr und mehr aufkommenden graphisch-interaktiven, multimedialen Fahrerinformationssystemen.

4 Literatur

[Bal 96] Helmut Balzert: Lehrbuch der Software-Technik; Spektrum Akademischer Verlag GmbH, 1996

[BoRuJa 99] Grady Booch, James Rumbaugh, Ivar Jacobson: the Unified Modeling Language User Guide; Addison Wesley Longmann, Inc., 1999

[FaHo 98] Johannes Fasolt, Dr. Peter Hofmann: Echtzeitsysteme objektorientiert modelliert; Elektronik, Ausgabe 16, 18, 20, 1998

UML is great for Embedded Systems – Isn't it?

Theodor Tempelmeier[1]
Fachhochschule Rosenheim

Preamble. Object-Oriented Design is definitely the right way of building systems, and this also applies to embedded systems in the view of the author. Encapsulation, abstract data types, etc., and other concepts such as generics or templates, are a must when developing complex embedded systems. This is in fact also the way the more progressive companies do already develop embedded systems.[2] The author is reluctant, though, to accept inheritance as a dominating design principle per se (see appendix). But at least an object-based approach is the right choice in the author's opinion, without any doubt. However, this contribution does not deal with object-orientation in itself, but only with the application of the Unified Modeling Language within this context.

1. Introduction

Thinking about the application of the Unified Modeling Language (UML) to embedded systems development actually involves questions on different issues:

1. Can something be modelled in UML in principle?
2. Can something be modelled in base UML (i.e. UML without extensions)?
3. Can something be modelled in UML in a "better" way, seen from a pragmatic point of view, i.e. concerning tool availability, etc.?
4. Can something be modelled in UML in a better way with respect to the concepts in UML?

The answer to the first three questions is probably a plain "yes". If something can be modelled with certain concepts (say in the framework of ROOM) then it can be modelled in UML by defining extensions (stereotypes) which exactly exhibit the same behaviour as the original concepts. UML just serves as an implementation vehicle, in this case. Concerning the second question, one may well assume that anything can be modelled with UML, given the plethora (and vagueness) of concepts in UML. And, of course, availability of tools will profit from the unification process. This contribution only deals with the fourth question, both from a practical and theoretic point of view.

2. Software Requirements Specification

From an actual project in the aerospace industry, three sample issues are used to evaluate the Unified Modeling Language for the requirements specification phase (see appendix). The conclusion from this is as follows.

Appropriate models for specification have been used in the engineering domain for quite some time (perhaps in contrast to the domain of business and administrative applications). It can not easily be seen how the Unified Modeling Language would provide any fundamental improvement as compared to the current modelling approaches.

---

[1] Prof. Dr. Theodor Tempelmeier, Fachbereich Informatik, Laboratory of Real-Time Systems, Fachhochschule, Marienberger Str. 26, 83024 Rosenheim, Germany, phone +49-8031-805-510, fax -105, e-mail tt@extern.lrz-muenchen.de, world wide web http://www.fh-rosenheim.de.

[2] By the way, the call for papers for this particular workshop seems to be wrong in claiming that "software for embedded systems is mostly *designed* according to the concepts of Structured Analysis." It is true that Structured Analysis is very often used for describing and analysing(!) the software requirements, but using Structured Analysis for *design* is rather rare (only in Ellis: Objectifying Real-Time Systems, for instance).

3. Software Design

The role of a modelling language in the design phase may be manifold. The two most important roles are

- the modelling language is used as design language down to coding, i.e. the modelling language is also used for "programming"
- the modelling language is used for design and the programming language is used for coding

The problem with the first approach is that most modelling languages do not have semantics as precise as it is necessary for programming. The problem with the second approach is that there may be a semantic discrepancy between the modelling language and the programming language (the programming language is in ultimate authority).

From the experience of applying object-based designs to embedded systems and from experience with various modelling languages and tools (e.g. HOOD), the author takes the following position.

- "Programming" in the modelling language is neither practicable nor reasonable nor desirable.
- The modelling language can be and should be used for a "visualisation" of the design. *This also implies that the modelling language resembles the design concepts within the programming language on a one-to-one, or "isomorphic" basis.* (Naturally, this is only valid if the programming language includes sound design concepts, e.g. programming languages such as assembler are ruled out.)

Obviously, the Unified Modeling Language fits the concepts of mainstream languages such as C++ or Java nicely. The author could accept UML for representing designs targeted to these languages, though not all concepts of embedded systems development (outside C++ and Java) can perhaps be described adequately.

If other target languages are used, e.g. Ada for safety-critical systems, the situation is different. Ada has sound design concepts (which are sometimes superior to C++ or Java concepts) which seem to be without counterpart in UML. This gives rise to the weird situation that the language for modelling a design is less powerful (in terms of design concepts) than the programming language. The following three examples are given.

- *Protected objects.* Protected objects are not supported by UML. The keyword "protected" has a totally different meaning to the C++ or to Ada95 programmer. What would be the meaning of a keyword "protected" in UML?
- *Hierarchical libraries.* Hierarchical libraries can be realised by nesting, which causes some problems concerning recompilation, etc. Ada95 has a much smarter scheme for hierarchical libraries which avoids such problems. How are hierarchical libraries handled in UML? Even if semantics in UML were simply tied to the library model of Java, one would still face the question of how to handle the generic hierarchical library units of Ada95, as there are no generics in Java. Further, it would still be open how to distinguish nested hierarchies from "smart" hierarchies.
- *Abstract data types (in the meaning of a class without inheritance) as distinct from classes (with inheritance).* Ada95 offers both classes without and with inheritance (keyword "tagged"). There are good reasons for this distinction, for instance that one sometimes wants to avoid certain aspects of inheritance in safety-critical systems (see ISO/IEC), while there is no reason to refrain from using abstract data types. Suggestions to use the Java concept of "final" classes for the purpose of this distinction may be acceptable, but this seems not to be a concept within base UML, version 1.3alpha.

It is possible, of course, to enrich UML with special notes and comments to reflect some additional concepts. Additionally, tools may employ a variety of switches to steer code generation in the desired way. But, this is exactly a repetition of the well-known situation with earlier notations and tools: The designer has in mind an exact idea of his design (in terms of the concepts of the target language, say, Ada). He then has to understand the design notation (UML), its semantics (maybe defined in terms of another language, say, C++ or Java), and the effects of the code generation switches of the tool (for instance about 50 switches for generating C++ classes in the Rational Rose tool). And then, maybe, the designer will get the design (tailored to his target language) he had exactly in mind from the very beginning. Such a procedure his hardly acceptable in practice.

As a conclusion, a one-to-one (or "isomorphic") correspondence of design concepts in the modelling language and in the programming language is required (if the programming language itself includes a set of adequate design concepts as it is the case with Ada). This requirement does not seem to be fulfilled by the combination "Ada and UML".

Appendix:
Excerpt with modifications[3] from a presentation at the 24th IFAC/IFIP Workshop on Real-Time Programming, Schloß Dagstuhl, Germany, May 30 - June 2, 1999.

Aspects of Flight Control Software - A Software Engineering Point of View

Alfred Roßkopf
DaimlerChrysler Aerospace AG

Theodor Tempelmeier
Fachhochschule Rosenheim

...

Software Requirements Specification

Requirements for flight control law software are usually defined using control block diagrams, which essentially describe blocks and data flows between them. Each block represents a transformation of input data to output data. The detailed control law computations are described in an algorithmic language, e.g. in FORTRAN. It is essential that this specification is executable, in order to validate the control law design.

In a small case study the Unified Modeling Language UML (Version 1.0) has been investigated with respect to specifying requirements for typical flight control software. The results are presented here in the form of three examples:

- definition of control surfaces
- control law block diagrams
- definition of external interfaces

Definition of Control Surfaces.

The main outputs of a flight control system are commands to the control surface actuators. Therefore, a requirements specification for flight control software would typically include a definition of these control surfaces. A possible UML definition for this is given in figure 1. A control surface “is a” primary or a secondary surface (inheritance relation) and so on. And a delta-canard aircraft “has” one rudder, four flaperons, and so on (aggregation or composition relation).

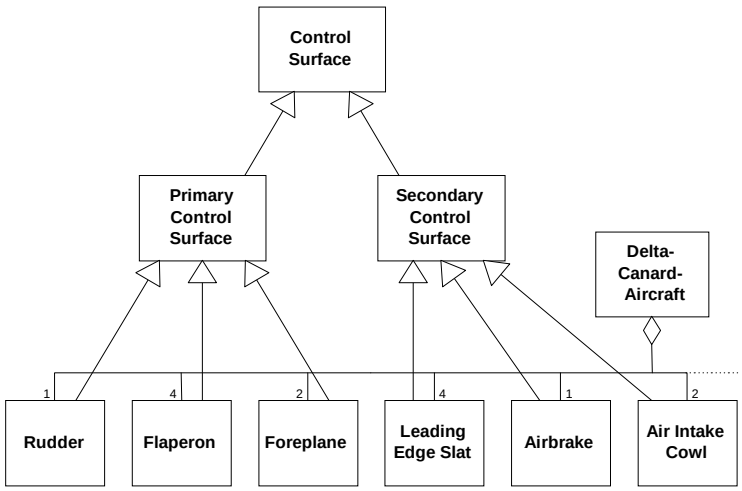


Fig. 1: Control Surfaces of an Aircraft in Delta-Canard-Configuration (UML Diagram)

[3] Modifications are shown as footnotes and are in the sole responsibility of the second author (T. Tempelmeier).

Figure 2 shows as an alternative a sketch of an aircraft with the primary control surfaces emphasised in white and the secondary ones in dark. It can be seen that a simple figure is sufficient to convey at least the same information as the UML diagram. The authors would not consider it very helpful to re-phrase such parts of the requirements in UML.

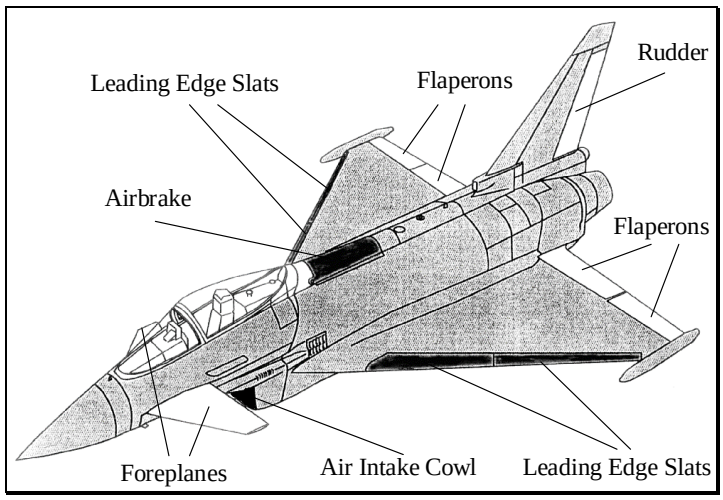


Fig. 2: Control Surfaces of an Aircraft in Delta-Canard-Configuration

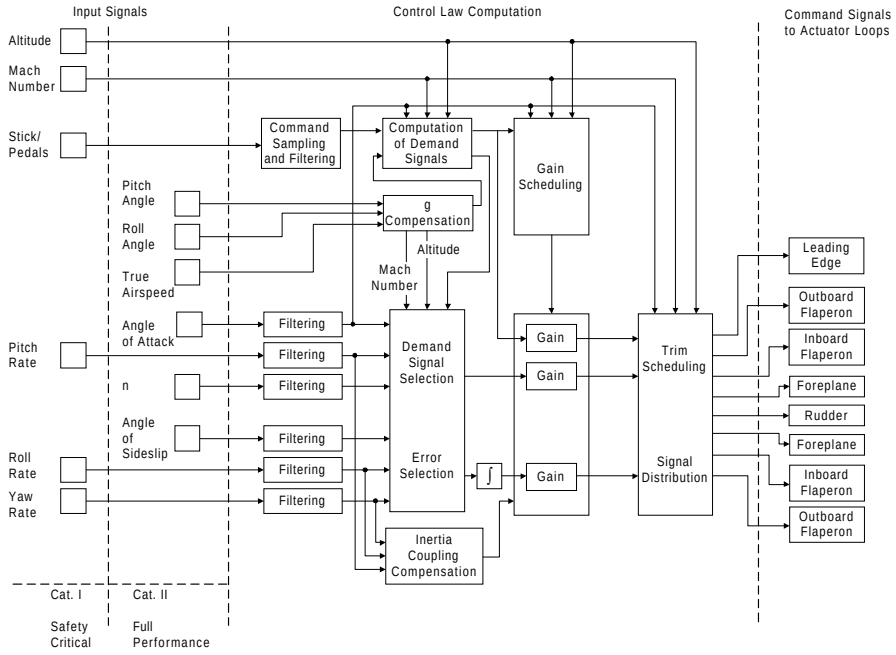


Figure 3: A block diagram of flight control laws according to (Kaul, 1992)

Control Law Block Diagrams.

As said above, control block diagrams contain fundamental parts of the requirements for control law software. Figure 3 shows an example. If one would try to rephrase such diagrams in UML, the following difficulties arose.

- Obviously, the control block diagram is not a class diagram. Instead, the blocks constitute

multiple instances of classes, e.g. the "Filtering" objects. The control block diagram thus must rather be seen as an "object diagram". While it is possible to draw object diagrams in UML, only the association relation can meaningfully be used in this case. However, the association relation is only a line drawn between rectangles with almost no semantic information. UML's object diagrams are thus not a suitable alternative for control block diagrams.

- Class diagrams do not help very much in this case, either. They would reveal surprisingly little information, for instance that a notch filter "is a" filter, etc. The complexity of the control block diagram lies in the functionality, i.e. in the contents of the block diagram elements and in their interdependencies. In contrast, UML seems to be more suitable for applications where the complexity lies in the class relationships, like in database applications, for instance.
- One could try to (mis)use other diagrams of UML, e.g. collaboration diagrams or activity diagrams, but the authors do not see any advantage in doing so, as compared to using well established control block diagrams.
- [4]

Definition of External Interfaces.

Requirements for flight control software must include a definition of the external interface of the software, including the formats of the input and output values. The authors consider Ada a good choice for specifying these formats and advocate its use starting with the requirements definition. The following examples repeat some of these probably well-known features of Ada.

```
type switch_values is (neutral, on, off);
  for switch_values use (neutral => 1,
                         on => 2, off => 4);

  C_Small_180 : constant  := 180.0 * 2**-15 ;
type  T_Fixed_180  is  delta  C_Small_180
                         range  -180.0 .. 180.0 ;
  for T_Fixed_180 'small use C_Small_180 ;
  for T_Fixed_180 'size   use 16 ;
```

Such Ada definitions, firstly, have precise semantics according to the language definition. Secondly, the use of the representation clauses ("for ...") allows a precise format definition down to the bit level. For instance, the first definition ties the switch values to their bit representations in say a digital input or output register, the second definition associates fixed point values with the format of an analog digital converter, for instance. Note the use of a delta which is not a power of two in the latter case. This represents a very common situation in practice.

Most importantly, using such definitions in the software requirements specification automatically guarantees consistency between (this part of) the requirements with the final code. The authors feel this method to be superior to rephrasing such requirements in some formal notation, and then perform proofs of consistency with the final Ada code. On the other hand, using UML would almost certainly be inferior, because there is no semantics and not even a syntax for describing types in UML[5].

Software Design and Implementation

In a research and technology project of Daimler-Chrysler Aerospace, an integrated process for the development of control laws for complex aircraft configurations has been investigated. As part of this project control laws for a typical fighter aircraft have been implemented in Ada. The resulting software is called COLAda (Control Law Software in Ada). In a companion publication, various architecture and design aspects of COLAda are reported (Roßkopf, 1999). Here, the focus is on the following three general issues:

- using object-oriented design techniques
- design for multi-processor targets
- using finite state machines

Using Object-Oriented Design Techniques.

In the design of COLADA object-oriented techniques have been used, where appropriate. Certain control law elements, e.g. filters, have been identified as candidates for objects, and corresponding abstract data types. (e.g. filter types) have been defined. Several objects of these types may be created as required by the control law application–the objects are just instances of abstract data types. The external behaviour of these objects is defined by the operations associated with the abstract data type. Of course all internal data are encapsulated in the objects.

According to a certain terminology, the design might be termed to be object-based, as no type exten-

[4] Finally, use case diagrams could be used. It is still unclear to the author, whether use cases are just a recurrence of the once condemned functional decomposition models such as Structured Analysis, or whether they also contain some fundamental new ideas (in UML version 1.3, eventually).

[5] A rudimentary type system and some other formalisations have been added to UML in version 1.1 with the help of the so-called Object Constraint Language OCL, though.

sion (“inheritance”) is used. This is for the following reasons:

- Full use of inheritance, in particular polymorphism and dynamic binding (i.e. the use of class-wide types in Ada terminology), may cause certain problems in safety-critical systems (cf. ISO/IEC, 1998).
- There is hardly a need for using inheritance in the context of this project. The cases where variants of certain object types occur (e.g. first order and second order filters) can easily be covered without inheritance.

Though certain aspects of flight control software can be designed and implemented in an object-oriented way, and though the authors very strongly favour such an approach, there must be a warning against hypocrisy with respect to object-orientation: The approach “everything is an object” does not seem very helpful. Over-simplistic approaches like in Coad et. al., where a case study of an object-oriented auto-pilot system is reported on, seem to be impractical for the implementation of real flight control software. And, of course, normal arithmetic operations and static typing are to be used in control law software. This is in contrast to the philosophy of radical object-oriented languages such as SmallTalk and Lisp, for instance, which are considered unsuitable to real-time safety-critical systems.

... ...

References

Coad, P., North, D., Mayfield, M.: Object Models. Strategies, Patterns, and Applications. Yourdon Press, Englewood Cliffs, 1995.

Ellis, J.R.: Objectifying Real-Time Systems. SIGS Books, New York 1994.

ISO/IEC: Working Draft 3.8 - Programming Languages - Guide for the Use of the Ada Programming Language in High Integrity Systems. ISO/IEC PDTR 15942, ISO/IEC JTC 1/SC22/WG9, October-29, 1998.

Kaul, H.J.: Flugsteuerungssystem Jäger 90 (In German). In: G. Bürgener (Schriftleitung). Jahrbuch 1992 III der Deutschen Gesellschaft für Luft- und Raumfahrttechnik e.V. (DGLR). Deutscher Luft- und Raumfahrtkongreß 1992. DGLR Jahrestagung, Bremen, 29. September – 02. Oktober 1992. Deutsche Gesellschaft für Luft- und Raumfahrt e.V. (DGLR), Bonn 1992.

Rosskopf, A.: Development of Flight Control Software in Ada - Architecture and Design Issues and Approaches. Submitted to: Ada-Europe'99. International Conference on Reliable Software Technologies. June 7-11, 1999. Santander, Spain.

Schwald, A.: UML und Ada95 – Über das Zusammenwirken zweier Stars. (UML and Ada95 – About the Cooperation of two Stars. In German.) Vortrag auf dem Workshop „Objektorientierung und sichere Software mit Ada“ der GI-FG 2.1.5 Ada „Ada-Deutschland“, 21. und 22. April 1999, Karlsruhe, Germany. To appear as a report of the Forschungszentrum Karlsruhe.

Tempelmeier, T.: An Overview of the HOOD Software Design Method. In: Real Time Computing. NATO ASI Series F, Vol. 127. Halang, W.A. and Stoyenko, A.D. (eds.). Proceedings of the NATO Advanced Study Institute on Real Time Systems, Sint Maarten, Dutch Antilles, October 5-17, 1992. Pages 726-734. Springer Verlag, Berlin, Heidelberg, New York 1994.

Tempelmeier, T.: Formal Methods – An Informal Assessment. Technischer Report Dasa MT36 SR-1775-a. 41 pages. Daimler-Benz Aerospace, Ottobrunn August 1997.

Tempelmeier, T.: Hierarchical Object-Oriented Design (HOOD) – Die Software-Entwurfsmethode der europäischen Raumfahrtbehörde ESA. (Hierarchical Object-Oriented Design (HOOD) – The Software Design Method of the European Space Agency ESA. In German.) Kolloquiumsvortrag an der Universität Oldenburg, 29.6.98.

Positionspapier für:
OMER – Objektorientierte Modellierung eingebetteter Realzeitsysteme
Workshop der GI-Fachgruppe 2.1.9, OOSE Objetorientierte Software-Entwicklung,
28./29. Mai 1999, Herrsching am Ammersee

Objektorientierte Konzepte in sicherheitskritischen Echtzeitanwendungen

Berner&Mattner Software Produkte GmbH
Otto Hahn Str. 34
D - 85521 München - Ottobrunn

Autor: Siegfried Hörfarter

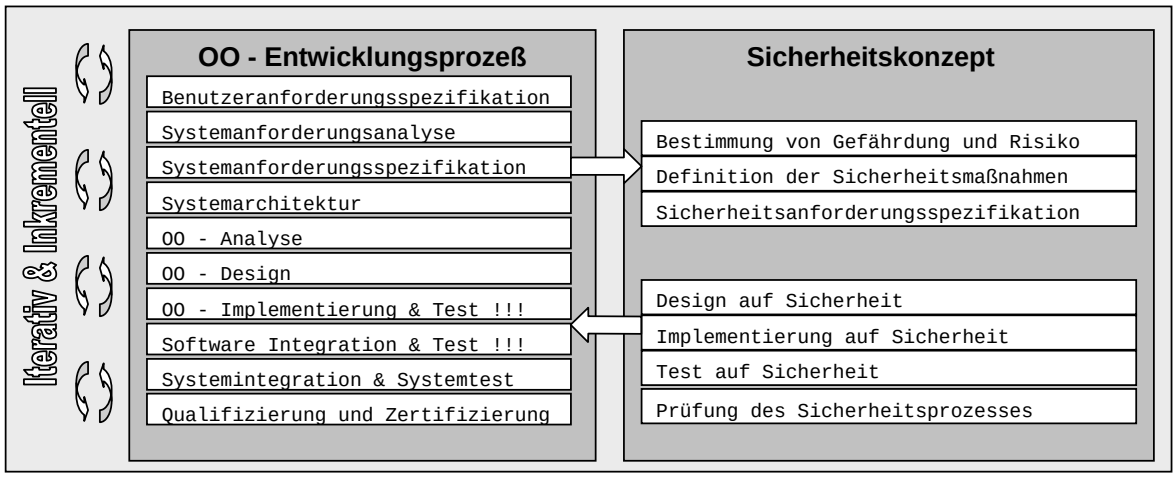
1.) Einleitung: Software hält auch in sicherheitskritischen Systemen unaufhaltsam Einzug. Viele Systeme in der Industrie, in der Medizin- und Verkehrstechnik bis hin zur Luft- und Raumfahrt stellen hohe Anforderungen an die Sicherheit und Zuverlässigkeit der Software. Gleichzeitig werden die Systeme aufgrund steigender Anforderungen immer komplexer. Die Forderung nach Wiederverwendung, Erweiterbarkeit und Wartbarkeit von Software ist vorhanden. Die letztgenannten Forderungen werden nachweislich durch Objektorientierte (OO) Entwicklungsmethoden positiv unterstützt. Inwieweit diese Konzepte auch den Anforderungen an sicherheitskritische Echtzeitsysteme genügen, ist umstritten. In diesem Beitrag wird diskutiert, unter welchen Bedingungen OO - Konzepte für die Entwicklung solcher Systeme verwendet werden können.

2.) Definition: Sicherheitskritische Software ist Software, welche bei einem Fehlverhalten direkt oder indirekt ein System in einen gefährlichen Zustand bringen kann. Echtzeitsoftware ist Software, die neben den funktionalen Spezifikationen auch zeitliche Spezifikationen einhalten muß. Das richtige Ergebnis zum falschen Zeitpunkt ist ein falsches Ergebnis! Wichtig ist die Unterscheidung von Sicherheit (safety) und Zuverlässigkeit (reliability). Ein System kann sehr zuverlässig aber trotzdem für einen Anwender unsicher sein. Sicherheit heißt in diesem Kontext Sicherheit für Mensch und Maschine. Daneben gibt es noch die datenschutzbezogene Sicherheit.
Die Schwierigkeiten bei der Entwicklung von sicherheitskritischer Echtzeit (SKEZ) Software:

- Einhaltung harter Echtzeitanforderungen
- Hohe Anforderungen an Sicherheit und Zuverlässigkeit
- Optimierter Umgang mit Systemresourcen (CPU-Rechenleistung und Speicher)
- hardwarenahe Programmierung
- Einhaltung von strengen Sicherheitsstandards
- Verfügbarkeit von Entwicklungswerkzeugen (z.B. Qualifizierte Compiler und Tools)
- Verfügbarkeit von Betriebssystemen und Zielplattformen - häufig Crossentwicklung

3.) OO Entwicklungsprozeß für SKEZ Software:

Abb. 1: OO - Entwicklungsprozeß mit Sicherheitskonzept



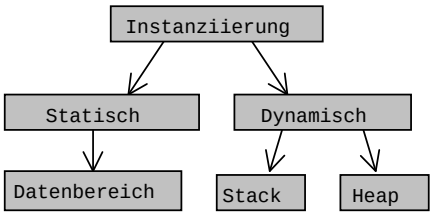
Bei der Entwicklung von SKEZ-Software ist die Einhaltung besonderer Standards (z.B [VDE0801/3],[DO 178B],[MIL-STD-882C]) zwingend vorgeschrieben. Dazu ist ein entsprechend angepaßter Entwicklungsprozeß notwendig. In Abb. 1 ist der prinzipielle Prozeßablauf skizziert. Der Entwicklungsprozeß wird durch das Sicherheitskonzept maßgeblich beeinflußt. Die zur Entwicklung der Software wesentlichen Prozeßschritte können mit OO - Analayse, OO - Design, OO - Implementierung und Test zusammengefaßt werden. Es hat sich dabei gezeigt, daß für die Entwicklung von komplexer Software eine Modellierung unumgänglich ist. Eine gezielte Modellierung in der Analyse- und Designphase reduziert Systemfehler und den Aufwand für die Implementierung. Für die OO - Modellierung von SKEZ Software bietet dabei die Unified Modeling Language (UML) viele Möglichkeiten.
Für SKEZ Software sind die **Verhaltensmodelle** von großer Bedeutung, weil diese eine Modellierung des Zeitverhaltens von Objekten und Objektinteraktionen ermöglichen. **Jedes Objekt** mit Zustandsverhalten kann **jeweils mit genau einem** UML - Zustandsdiagramm ([Harel87]) formal beschrieben werden. Die Interaktion von mehreren Objekten wird in Szenariodiagrammen dargestellt. Beim OO - Entwicklungsprozeß liegt ein entscheidender Vorteil darin, daß im zentralen Bereich der OO - Analyse, des OO - Designs bis hin zur Implementierung **ein konsistentes Darstellungsmodell verwendet** werden kann. Die Modellinhalte unterscheiden sich dabei nur im Detaillierungsgrad. Dies ist wichtig für ein iteratives und inkrementelles Vorgehen. Darüber hinaus ist eine hierarchische Strukturierung des Modells über sog. Packagediagramme möglich.

4.) OO - Konzepte in SKEZ Systemen
Das Klassenkonzept und die Objektinstanziierung bilden die Basis für OO - Software mit Abstraktion, Kapselung, Vererbung und Polymorphismus. Die folgenden Betrachtungen orientieren sich an C++. In anderen OO - Programmiersprachen werden die Konzepte teilweise anders umgesetzt (z.B. keine Konstruktoren und Destruktoren in Ada 95).

4.1) Die **Klassen** bilden die Grundlage zur Erstellung von abstrakten Datentypen. Gute Abstraktion betont die wesentlichen, essentiellen Eigenschaften und unterdrückt die "unwichtigen" Details. Durch diese Vereinfachung können komplexe Systeme wesentlich einfacher strukturiert und beherrscht werden. Klassen stellen Einheiten aus Attributen (Membervariablen) und Operationen (Memberfunktionen) dar. Klassen bilden dadurch Minisysteme mit eigenem Verhalten.

4.2) Eine Klasse stellt die Typdefinition dar. Erst durch die **Objektinstanziierung** wird ein Objekt einer Klasse erzeugt. Bei der Instanziierung wird Speicherplatz für ein Objekt reserviert und die Konstruktoroperation aufgerufen. Das Gegenstück zur Konstruktoroperation, die Destruktoroperation garantiert eine definierte Objektfreigabe. Der Destruktor wird zum Zeitpunkt der Objektfreigabe Compiler-generiert aufgerufen. Es wird zwischen statischer Instanziierung und dynamischer Instanziierung unterschieden.

Abb. 2 Instanziierungsmethoden:




```
CSensor  humiditySensor(...); // statische Instanziierung im Programmdatenbereich

void f1()
```

```
{
   CSensor  pressureSensor(...);    // dynamsiche Instanziierung auf dem Stack
   static CSensor tempSensor(...);  // statische Instanziierung im Programmdatenbereich
   ....
   CSensor* pSensor = new CSensor(...);   // dynamische Instanziierung auf dem Heap
   ....
}  // end of f1()
```

Statisch instanziierte Objekte werden im Programmdatenbereich (Datensegment) angelegt. Bei der dynamischen Instanziierung muß zwischen der Instanziierung auf dem Stack und auf dem Heap unterschieden werden. In jedem Fall ist zu berücksichtigen, daß die dynamische Instanziierung fehlschlagen kann, falls nicht ausreichend Speicher vorhanden ist (out of memory). Daher muß eine ausgefeilte Ausnahmebehandlung vorgesehen werden. **Die dynamische Speicherverwaltung stellt eine der größten Schwachstellen für sicherheitskritische Systeme dar.** Wenn immer möglich sollte nach der Startup- und Initialsierungsphase des Programmes auf dynamische Instanziierung verzichtet werden.
Bei der Speicherverwaltung ist darüberhinaus das Problem der Defragmentierung zu beachten. Speziell bei einer Vielzahl von "kleinen" Objekten und einer hohen Dynamik im Programmablauf kann dadurch der Speicher frühzeitig knapp werden. Eine Möglichkeit zur Abhilfe besteht in der Anwendung des Fixed Size Allocation Patterns. Dabei wird der Speicher in Segmenten einer gewissen Grundgröße (z.B. n * 256 Byte) verwaltet. Generell ist bei zeitkritischen Systemen auch die Programmlaufzeit beim Speichermanagement zu berücksichtigen.
Die statische Instanziierung wird zum Zeitpunkt des Programmstarts (bzw. bei statischen funktionslokalen Objekten beim ersten Eintritt in die Funktion) im Datenbereich des Programmes durchgeführt. Es ist zu beachten, daß auch diese Instanziierung u.U. nicht rein statisch ist, weil die Konstruktoroperation ausgeführt wird. Eine zuverlässige Technik für sicherheitskritische Anwendungen besteht darin, bereits in der **Startupphase** des Programmes alle notwendigen Objekte anzulegen (Static Allocation Pattern). Dieser Vorgang kann sehr gut getestet werden und eine "sichere" Reaktion auf Speicherprobleme ist in dieser frühen Programmablaufphase noch gut möglich. Konstruktoren und Destruktoren haben für sicherheitskritische Systeme zwei Seiten. Sie tragen einerseits wesentlich zur Erhöhung der Sicherheit von Software bei, weil der Programmcode zum Aufruf des Konstruktors und Destruktors automatisch durch den Compiler generiert wird. Ein "Vergessen" einer Init - Funktion kann dadurch sehr einfach ausgeschlossen werden. Andererseits ist aber der Konstruktor- und Destruktoraufruf im Programmablauf nicht direkt ersichtlich. Dies erschwert die Lesbarkeit und Nachvollziehbarkeit von Programmen. Für SKEZ Software gilt daher, daß **Konstruktoren und Destruktoren unbedingt kurz und so einfach wie möglich zu halten** sind.

4.3) Das Konzept der **Kapselung** ist für die Entwicklung von sicherheitskritischer Software von großer Bedeutung. Die Kapselung stellt eine elementare Eigenschaft von Objekten dar und schützt die Attribute von Objekten vor unerlaubtem Zugriff. Nur über öffentliche (public) Operationen darf der interne Zustand von Objekten verändert werden (das vielgenannte Performanceproblem kann durch Inline Operationen praktisch eliminiert werden). Darüber hinaus kann in diesen Operationen jeweils auch eine **Bereichsüberprüfung** vorgesehen werden, welche verhindert, daß Attribute auf Werte außerhalb des Wertebereiches gesetzt werden. Für SKEZ Software ist zu beachten, daß das Kapselungskonzept durch Vererbung aufgeweicht werden kann. Auf "protected" Variablen (in C++) kann auch in abgeleiteten Klassen direkt zugegriffen werden. Dies sollte beim Design berücksichtigt werden.

4.4) Über **Vererbung** können Attribute und Operationen von bereits bestehenden Klassen genutzt werden. Dadurch ist zum einen Wiederverwendung (programming by difference) und zum anderen eine hierarchische Abstraktion einer Problemstellung möglich. In den abgeleiteten Klassen sind alle Attribute aus den Basisklassen enthalten. Beim Design der Klassen ist zu beachten, daß

alle 'protected' Attribute einer Basisklasse auch für die abgeleiteten Klassen direkt zugänglich sind. Für SKEZ Systeme ist zu betonen, daß Vererbungsbeziehungen (ohne Polymorphismus) **zu keinem zusätzlichen Overhead** (weder Speicherbedarf noch Laufzeit) führen. **Vererbung kann daher in SKEZ Systemen gut eingesetzt werden**. Es ist jedoch vernünftig die Vererbungshierarchien einfach zu halten, da komplexe und unübersichtliche Strukturen in sicherheitskritische Anwendungen generell vermieden werden müssen (schwierigere Inspektion und Reviews). Neben der dargestellten Einfachvererbung gibt es z.B. in C++ die Möglichkeit zur **Mehrfachvererbung**. In Abb. 3 wird ein Sensor mit CAN Interface durch Ableitung von einer Klasse Sensor und einer Klasse CANInterface erzeugt.

Abb. 3 Mehrfachvererbung

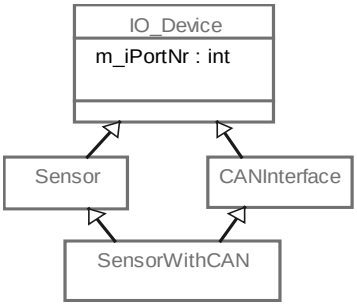
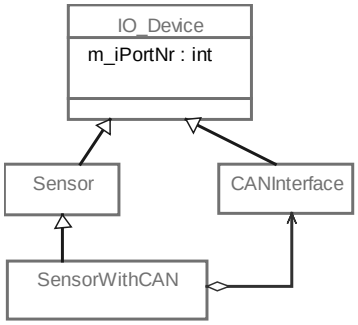


Abb. 4 aufgelöst durch Aggregation



SensorWithCAN erbt alle Attribute von Sensor und CANInterface. Sensor und CANInterface wiederum erben alle Attribute der Klasse IO_Device. D.h. m_iPortNr wird in der SensorWithCAN Klasse zweimal (!) enthalten sein. Dieses Problem läßt sich z.B. in C++ durch 'virtual' Vererbung lösen. In vielen Fällen ist jedoch diese Abhängigkeit nicht so offensichtlich. Mehrfachvererbung kann auch die Ursache eines schlechten Designs sein. Sie kann sehr oft mit einer Aggregationsbeziehung gelöst werden (siehe Abb. 4). Zudem ist Mehrfachvererbung in Ada 95, Java und EC++ nicht möglich. Aufgrund dieser Aspekte und der relativ hohen Komplexität darf **Mehrfachvererbung in SKEZ Systemen nicht eingesetzt** werden.

4.5) Polymorphismus nutzt das Konzept der späten Bindung. Erst zur Laufzeit des Programmes wird anhand des referenzierten Objekttyps entschieden, welche Operation ausgeführt wird. In Abb. 5 ist ein Beispiel dargestellt, in dem ein Roboter ein Positioniersystem verwendet.

Der Roboter verwendet für eine horizontale Positionierung einen Schrittmotorantrieb.

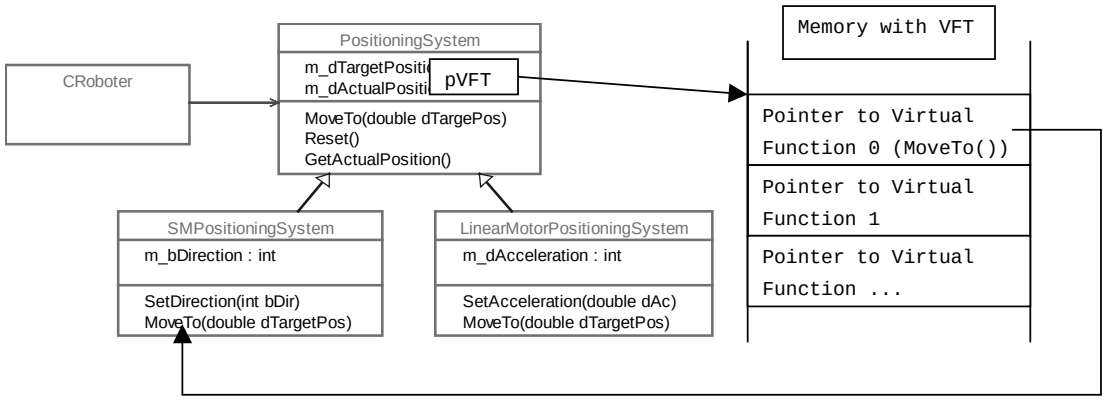
```
// maschinenabhängiger Code
PositioningSystem* pPosSystemHorizontal = new SMPositioningSystem;
// maschinenunabhängiger Code
pPosSystemHorizontal -> Reset();         // Operation der Basisklasse
pPosSystemHorizontal -> MoveTo(171.4);   // Operation der Klasse SMPositioningSystem
```

In einem anderen Roboter wird ein Linearmotor eingesetzt - die Änderung einer Zeile genügt. Der maschinenunabhängige Code bleibt gleich.

```
PositioningSystem* pPosSystemHorizontal = new LinearMotorPositioningSystem;
pPosSystemHorizontal -> Reset();      // Operation der Basisklasse
pPosSystemHorizontal -> MoveTo(171.4); // Operation der Klasse LinearMotorPositioning System
```

Polymorphismus schafft ein hohes Maß an Flexibilität und Abstraktion. In C++ werden alle Aufrufe von virtuellen Funktionen über die Virtual Function Table (VFT) durchgeführt (siehe Abb. 5). *PositioningSystem* enthält als (verstecktes) Attribut einen Pointer (pVFT) auf die VFT. Die Operation MoveTo() ist die erste virtuelle Funktion der Klasse und daher in der VFT auch im erster Eintrag zu finden. Über den Funktionspointer in der VFT wird die Funktion MoveTo() der Klasse *SMPositioningSystem* erreicht. Daraus ist klar ersichtlich, daß für den Aufruf einer virtuellen Funktion eine zusätzliche Pointerdereferenzierung notwendig ist und die Größe eines Objektes um den Pointer pVFT zunimmt. Dies spielt jedoch in vielen Fällen nur eine geringe Rolle. Problematisch für SKEZ Systeme ist die generelle Komplexität, die durch solche Konstrukte erreicht wird. Dies ist vor allem dann der Fall, wenn zusätzlich Funktionsüberladung eingesetzt wird. Ein weiteres Problem kann entstehen, wenn bei der Codierung die Implementierung der Operation MoveTo der Klasse *SMPositioningSystem* versehentlich vergessen wird. Für den Compiler ist dies nicht weiter problematisch, weil in der Klasse *PositioningSystem* eine Defaultimplementierung der Operation vorhanden ist. Zur Laufzeit wird dann einfach diese Defaultfunktion verwendet, die aber nicht die richtige Aktion ausführt. Daher sind virtuelle Funktionen besonders sorgfältig zu testen, weil Probleme u.U. erst zur Laufzeit des Programmes ersichtlich werden.

Abb 5. Polymorphismus und ein Virtual Function Call



Diese Aspekte tragen dazu bei, daß besonders in SKEZ Systemen mit **hohen Sicherheitsanforderungen auf Polymorphismus weitestgehend verzichtet** werden sollte.

5.) Erweiterte OO - Konzepte in SKEZ Systemen sind Klassenbibliotheken, Design Patterns und Frameworks. Es wird hier kurz auf die Design Patterns eingegangen. **Design Patterns** spielen eine wichtige Rolle, weil damit indirekt auf eine jahrelange Erfahrung von Experten zurückgegriffen werden kann. Es wird dabei zwischen den Architectural Design Patterns (für die generelle System- und Softwarearchitektur) und den Mechanistic Design Patterns (für das OO - Design) unterschieden. Einige Patterns sind speziell hinsichtlich Sicherheit und Zuverlässigkeit von Systemen ausgelegt (z.B. Redundancy, Monitor-Actuator, Static Allocation, Smart Pointer, etc.) andere wiederum haben Ihren Schwerpunkt in der Schaffung von wiederverwendbaren und erweiterbaren Systemen (z.B. Abstract Factory, Interface, Container, etc.). Für nebenläufige (concurrent) Software ist das Active-Object Pattern interessant. Aktive Objekte sind Objekte, welche eine Task repräsentieren und damit ihren eigenen Programmablauf (Thread of control) haben. Objekte sind für nebenläufige Systeme gut geeignet, weil der Nachrichtenaustausch über Messages eine grundlegende Eigenschaft von Objekten darstellt. Design Patterns erhöhen in SKEZ Systemen die **Sicherheit durch Verwendung bewährter Konzepte**. Sie müssen aber **nach den erlaubten OO - Konzepten erstellt** werden. D.h. Verzicht auf Mehrfachvererbung und keine dynamische Instanziierung zur Laufzeit!

6.) Die Implementierung von OO - Konzepten: Die OO - Implementierung in einer OO - Programmiersprache ermöglicht die schlüssige Umsetzung der in der Analyse und im Design eingesetzten OO Konzepte. Für die programmtechnische Umsetzung sind folgende Punkte zu berücksichtigen:

Wahl von Programmiersprache, Compiler und Tools
Behandlung von Ausnahmefällen
Verwendung der Programmiersprache und Programmierstil
Erfahrung und Motivation der Softwareentwickler

Der Hauptaspekt bei Programmiersprache und Compiler liegt in der Aufdeckung von möglichst vielen Fehlern bereits zur Compilezeit. Wird Exception-Handling von einer Programmiersprache unterstützt, so sollte die Verwendung ernsthaft in Erwägung gezogen werden. Damit ist eine saubere Trennung von Programmausführung und Fehlerbehandlung möglich. Der Programmablauf wird dadurch leichter verifizierbar und Fehler werden mindestens erkannt. Die Verifzierung des Sourcecodes gegen die Spezifikation durch unabhängige Personen ist eine Grundvoraussetzung für sicherheitskritische Software. Zur Implementierung von SKEZ - Software sollte in keiner Programmiersprache der gesamte Sprachumfang verwendet werden. Es empfiehlt sich immer die Verwendung einer sicheren Untermenge (z.B. Safe Subset von Ada 95 [ADA 95-REF]) oder die eigene Definition einer solchen (beispielsweise in C++: keine void*, keine Copy Konstruktoren und Zuweisungsoperatoren zulassen, etc.). Vielfach wird die Wahl der Programmiersprache von Standards, Verfügbarkeit von Compilern und natürlich von Wirtschaftlichkeitsüberlegungen beeinflußt. OO - Programmiersprachen für Embedded Systeme sind C++, EC++ (Embedded C++), Ada 95 und Java (Problempunkt Garbage Collection!), wobei aus rein sicherheitskritischen Überlegungen ganz klar Ada 95 der Vorrang zu geben ist (hohe Typsicherheit, keine impliziten Typkonvertierungen, keine Pointerproblematik, Arraygrenzenüberprüfung zur Laufzeit, zertifizierte Compiler!)

7.) Zusammenfassung: OO - Konzepte bieten bei richtigem Einsatz für sicherheitskritische Echtzeitanwendungen eine Reihe von sehr guten Möglichkeiten. Besonders durch das durchgängige Modellierungskonzept in der Analyse und Designphase, durch die Kapselung und durch Abstraktion kann die Sicherheit von Software wesentlich erhöht werden. Vor einer zu freizügigen Verwendung der in den Programmiersprachen gebotenen Möglichkeiten muß aber gewarnt werden. Die OO-Konzepte müssen wohlüberlegt angewendet werden. Dann steht einem Einsatz in einem auf Sicherheit ausgelegeten Entwicklungsprozeß nichts entgegen.

8.) Referenzen:

[ADA 95-REF] http://www.adahome.com/rm95/
[BMRSS96] A System of Patterns; Buschmann, Meunier, Rohnert, Sommerlad, Stal
[DO 178B] RTCA/DO 178B Software Considerations in Airborne Systems and Equipment Certification
[DOUG98] Real-Time UML; Bruce Powel Douglass
[GAM96] Entwurfsmuster (Gamma, Helm, Johnson, Vlissides)
[GRI98] Componentware – Konzepte und Techniken eines Software Paradigmas
[Harel87] „Statecharts: a Visual Formalism for Complex Systems“ (David Harel)
[ILOGIX] www.ilogix.com
[MIL-STD-882C] Military Standard 882C
[VDE0801/3] Funktionale Sicherheit – Sicherheitssysteme Teil 3 Anforderungen an Software (IEC 65A/181/CDV)

“Modelware”:
Aspekte der Componentware als Designprinzip für objektorientierte eingebettete Echtzeitsysteme

Daniel Hünig und Herbert Vogt
IDEASOFT AG, Tübingen

Leider hat uns das endgültige Papier nicht bis zur Drucklegung erreicht.

OMER Workshop Organisation

An interdisciplinary approach towards real-time system development

Bernd Gebhard
Bayerische Motoren Werke AG
Forschungs- und Ingenieurzentrum
D-80788 München
Bernd.GA.Gebhard@BMW.de

Abstract

The development of complex, software-intensive systems has set up and established object-oriented development methods. The specifics of the automotive world's reactive real-time systems raise additional requirements to be met. These have to be regarded from a system's point of view. Existing approaches have to be set into this wider spanned context. Utilising an appropriate framework or process model, established and standardised procedures have to be correlated to their compliance with formulated requirements, weaknesses and potentials must be identified and specified.

Automotive Specifics

Automotive product development is characterised by its complexity (many heterogeneous, permanently interacting functions), extreme schedule and cost pressures and demanding safety and reliability requirements. The development of automotive electric/ electronic systems has to meet additional requirements regarding strictly limited HW-resources and short innovation cycles related to longer vehicle life-cycles. The restricted HW-resources of the embedded real-time systems are characterised by RAM, ROM and processing power and have to comply with the correlating requirements regarding environment, thermals, energy, EMC, required space, weight, cost, etc.. The control of enduring product enhancement and upgrade over many years, i.e. the integration of „legacy" and innovative components and subsystems, is a serious issue, that additionally increases complexity and makes great demands on the development process.
This means, that the *system* design must not be considered as a single-shot *software* development that is completed with start of production. Additional management of knowledge is required to control modifications, i.e. to identify, evaluate and integrate the impact of changes, and to respect systems' aspects. This can only be accomplished reasonably by seamless data-flow and support of seamless tool-chains.

Paradigms

Structured analysis/ functional analysis

SA focuses on the demanded functionality of the system (functions, methods). Iterative and incremental refinement of abstract functionality shall lead to single functions, ready to be implemented. Interfaces between functions are defined by

data-flows, data-flows by functions generating and transforming data. Hierarchy concepts support the management of complexity.
Code generated based on a structured analysis will usually require less HW-resources.

Object-Orientation

Object-oriented concepts and languages describe the system's objects („objects of the real world"), its attributes and methods. More sophisticated concepts and mechanisms like encapsulation, inheritance and polymorphism support control of complexity and enable easier and more reliable reuse, maintenance and upgrade; the implementation of these methods into the development process is strongly dependent on the supporting tools. Standardised notations (UML) reduce ambiguity, but do not provide support in terms of process.
Code generated based on OOA/ OOD models will usually require more capable HW-resources.

The Development Process

In order to meet described requirements a process model has to be set up as a framework for discussion. In several discussions within automotive industry's projects there has been established a widely accepted proposal (fig. 1):

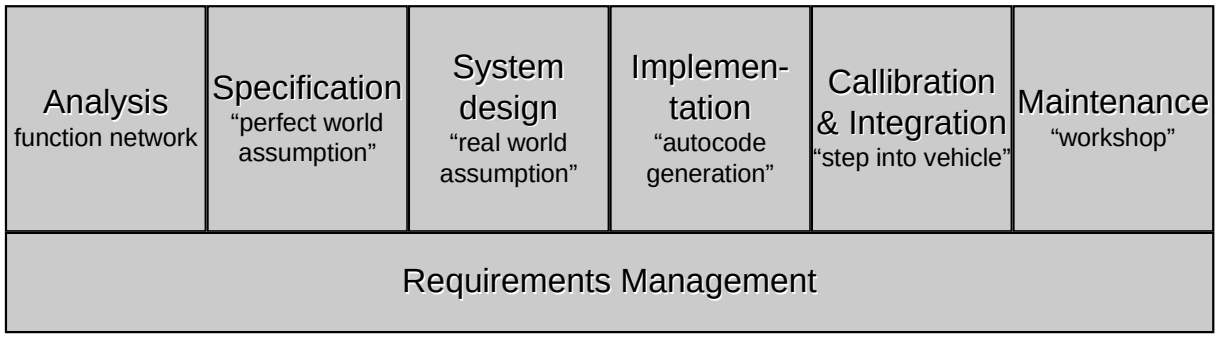


Fig. 1: Development Process

This process model has to be interpreted as a model for increasing maturity and reliability of product information. It does not imply a strict sequential processing of its single steps, on the contrary they *will* be performed in a parallel manner, incrementally and iteratively. Continual requirements management has to ensure internal (design) and external (users' needs) consistency.

Requirements Flowdown

Requirements are organised in different levels, that represent different levels of maturity of information (fig. 2). Basically requirements of all levels have to be complete, unambiguous and non-contradictory. One element of Requirements Management (fig. 1) is the derivation of a system design (architecture) from the initial user requirements, i.e. the controlled flowdown requirements.
User requirements represent the starting point of product development. They describe, *WHAT* the system under consideration has to provide in the customers'

or users' point of view (and language). They can be functional or non-functional (cost, performance, operating conditions, etc.).
Systems requirements describe, *HOW* the system will achieve this („abstract solution of the problem"). They are derived from user requirements and have to comply with them.
The concrete solution is characterised by the design model. It has to meet systems requirements (and thus user requirements). Possibly non-functional requirements will be met by functional implementation (e.g. disabling conditions and mechanisms to meet power and thermal balance).

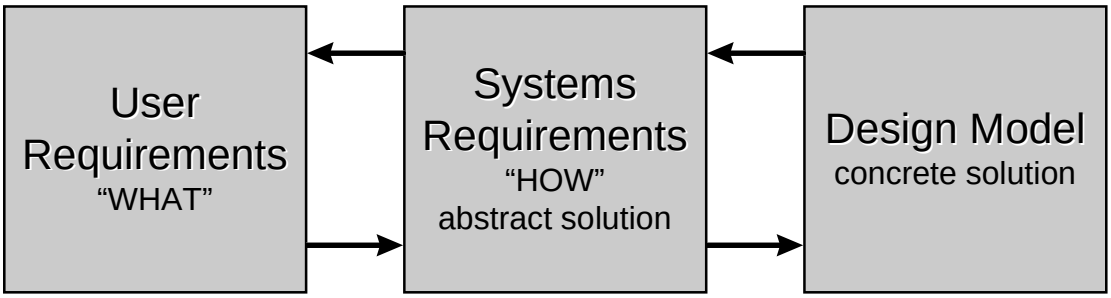


Fig. 2: Requirements Flowdown

Requirements management encompasses capture and maintenance (update) of information and identification and resolution of conflicting goals. I.e. the respective requirements models (users requirements, systems requirements, design) have to be „internally" consistent (resolved conflicts), and the derived requirement models have to be compliant with their ancestors.
It is crucial to separate the different levels of requirements, i.e. to gain logical consistency and non-contradictoriness *within* the items of each level. The impact of new items' integration has to be evaluated within the concerning level as well. The impact of change can be evaluated and calculated if completeness of information and consistency within and between requirements models is accomplished.

Interdisciplinary Co-operation

In object-oriented SW-analysis SW-engineers usually utilise UseCase analysis to define systems behaviour. This mainly focuses on functional requirements („WHATs"?) and preparing them for implementation. Non-functional context will not necessarily be regarded explicitly. In order to avoid late, difficult integration of completely implemented functions and not yet described functions, that are to implement non-functional requirements (power balance, thermal balance, diagnostics, security, etc.), non-functional requirements have to be encompassed as early as possible and integrated to the early requirements model(s). This is an issue, that would typically be carried out by systems engineers co-operating with the specialist disciplines (as shown in fig. 3).

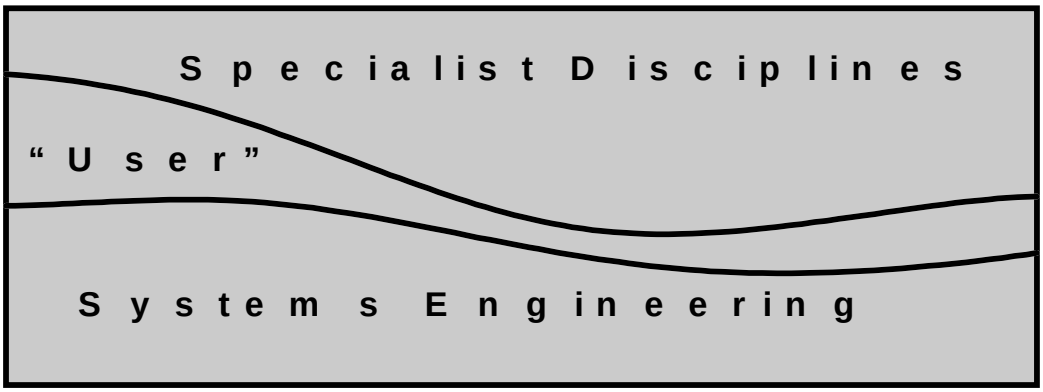


Fig. 3: Interdisciplinary co-operation

Defining an efficient way of interdisciplinary co-operation is an issue, that again has to consider the specific context of the development process.

Open Questions

- How can an analysis process exploit the full capabilities of UML's object-oriented concepts and mechanisms without neglecting non-functional (systems) requirements? Which of UML's mechanisms/ diagrams should be utilised to support the early phases of system analysis in automotive development?
  What could an automotive specific subset of UML look like?
- How can non-functional requirements be refined and implemented with minimum additional effort? Which are the most capable hierarchy concepts?
- Which are the best applicative concepts to refine top-level requirements and enrich the requirements model with non-functional elements? At which level of refinement should they be integrated most sensibly?
- What influence can the context of existing or available tools have on the decision for feasible analysis processes?
- Which are the information, specialist engineers and systems engineers have to communicate in early phases? How have their dedicated processes (s. fig. 1) to be connected?
- How can the results of software and systems engineering be controlled and managed over an entire product life of an automotive development (~10 years)? How are they to be linked (data, tools) most reliably and efficiently?

References

Booch, G.: *Object-oriented analysis and design with applications*, Santa Clara, California 1994

Selic, B., Gullekson, G., Ward, P.: *Real-Time Object-Oriented Modeling*, New York, 1994

Stevens, R., Martin, J.: QSS Library: *What is Requirements Management?*, URL=http://www.qssinc.com/library/reqmanagement.html

Beitrag zur Spezifikation verteilter Systeme mit UML

Ch. Diedrich, R. Simon, M. Riedl
chd@ifak.fhg.de, rsi@ifak.fhg.de, mri@ifak.fhg.de
ifak Magdeburg

MOTIVATION

Dieses Positionspapier ist im Bereich der Instrumentierung von verteilten Automatisierungssystemen angesiedelt. Instrumentierung meint die Umsetzung eines funktionellen Entwurfes von Automatisierungssystemen in reale Geräte bestehend aus Hardware und Software. Software setzt sich aus Betriebssystemen oder allgemeiner Plattformen und speziellen Anwendungslösungen. Die Instrumentierung ist in den Lebenszyklus von Automatisierungsystemen eingebettet und ist Teil des Engineerings, der Inbetriebnahme und der Betriebsphase.

In der heutigen Praxis sind die Lebenszyklusphasen zwar mit Softewarewerkzeugen unterstützt, jedoch fehlt insbesondere für die Instrumentierung eine herstellerübergreifende Konsitenz der Information zwischen den Werkzeugen. Im weitergehenden Sinne fehlen wissemschaftliche Methoden der Instrumentierung. Für die Instrumentierung sind Automatisierungsgeräte Produkte, die durch Eigenschaften, die aus Anwendungsfunktionen abgeleiteten worden sind, für den Auswahlprozeß charakterisiert sind. Dieser Ansatz führt zur Nutzung von Informatikmitteln, insbesondere des 00 Paradigmas für die Instrumentierung.

In diesem Posititionspapier werden mit Hilfe der objekt-orientierten Beschreibungsmethode UML (Unified Modeling Language) Strukturen für den funktionalen Entwurf und Gerätestrukturen gegenübergestellt. Die Beschränkung auf die Klassendiagramme rührt aus der Beschränkung auf statische Datenmodelle. Dynamische Aspekte, d.h. die Nutzung des Modellierungsergebnisses für die Ableitung von abarbeitbaren Kode, sind explizit ausgeschlossen.

Die Verteilung von Automatisierungsfunktionen bringt eine neue Qualität in das Engineering dieser Systeme. Neue Anforderungen werden insbesondere an die Werkzeuge gestellt, die die Produkt- d.h. Geräteentwicklung begleiten, die das Engineering/die Instrumentierung durchführen und die die Laufzeit begleiten. Menschliche Eingriffe in den Lebenszyklus sind auf Grund der hohen Komplexität der benannten Aspekte mit Risiko verbunden und benötigen deshalb methodischer bzw. wissenschaftlich begründeter Verifikation/Validation. Das vorgeschlagene Modell zeigt exemplarisch die Zusammenhänge zwischen funktionellen Anforderungen (z.B. aus Standards, Profilen oder herstellerinternen Festlegungen), der Geräteentwicklung und Produktbeschreibungen. Außerdem wird das Pattern “Proxy” für die Laufzeit aufgenommen, das als Übergangsstadium zu echt verteilten Systemen in Automatisierungssystemen häufig verwendet wird.

Die Plattform der Gesellschaft für Informatik wird in der Hoffnung gewählt, ein Diskussionsforum im Spannungsfeld zwischen ingenieurwissenschaftlichen Forschungen und Informatik zu finden.

BETRACHTETE KOMPONENTEN DES VERTEILTEN SYSTEMS

Verteilte Automatisierungssysteme heutiger Prägung bestehen aus Komponenten, die die verteilten Anwendungsfunktionen in unterschiedliche Plattformen einbetten. Typische Komponenten sind einfache E/As, intelligente Feldgeräte (z.B. Meßumformer, Stellantriebe), Steuerungen und Visualisierungssysteme. Verbunden sind diese Komponenten mit industriellen Kommunikationssystemen z.B. Feldbussen oder Ethernet TCP/IP. Bezogen auf ihre Anpaßbarkeit unterscheiden sich die Komponenten in parametriebare (z.B. Einstellung der Skalisierung von Meßumformern), in konfigurierbare (z.B. Visualisierungskonfigurierung von SCADA-Systemen) und programmierbare (z.B. Speicherprogrammierbare Steuerungen (SPS)). Die Transformation eines funktionalen Entwurfes (siehe Abbildung 1) in die entsprechenden Geräte-Parameter, -Konfiguration oder -Programme ist ein sehr komplexer Vorgang [4]. In geschlossenen, herstellerspezifischen Leitsystemen ist dieser Transformationvorgang meist unsichtbar, solange keine unbekannten Fremdkomponenten eingebunden werden müssen. Für offene Systeme ist der Aufwand jedoch z.Z. ingeneurtechnisch noch zu hoch. Durch geeignete Werkzeuge, die auf einem einheitlichen Modell aufsetzen, kann eine Effektivitätssteigerung erreicht werden. Für diesen Transformationsprozeß soll das hier vorgestellte Modell einen Beitrag leisten. Ausdrücklich wird darauf hingewiesen, daß die Ableitung von Code für die Laufzeit nicht betrachtet wird.

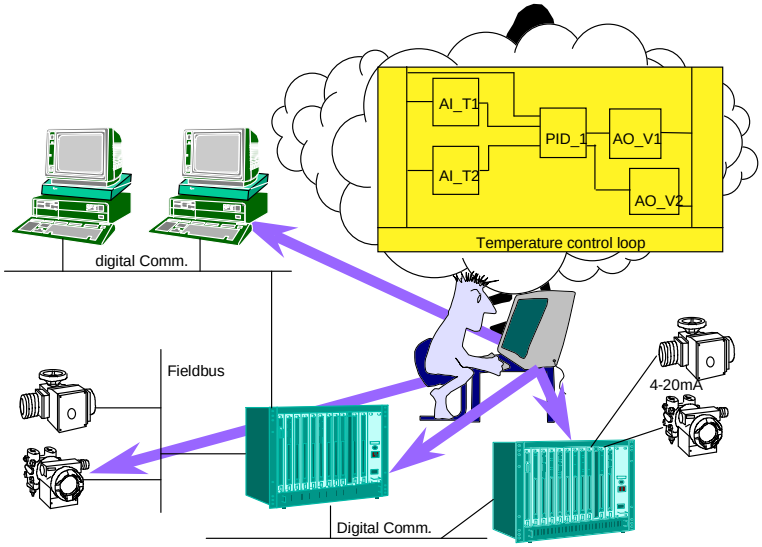


Abbildung 1. Umsetzung des FB-Entwurfes in die Geräte

DAS SYSTEMODELL ALS UML-KLASSENDIAGRAMM

Package-Übersicht

Einen kleinen Ausschnitt aus den benötigten Komponenten für verteilten System stellen die Geräte, deren Beschreibungen und die Repräsentation der Funktionen/Daten der Geräte in übergeordneten Stationen wie z.B. Steuerungen dar. Für jede Komponente sind Teilaspekte des verteilten Systems von Bedeutung, bzw. jede Komponente bringt neue Aspekte hinzu. Dieser Sachverhalt ist exemplarisch in der Packageübersicht Abbildung 2 dargestellt. Eine einheitliche Gerätestruktur wird im Kernel-Package festgelegt, aus der die beteiligten Komponenten abgeleitet werden. Dies ist mit den Dependency – Beziehungen ausgedrückt.

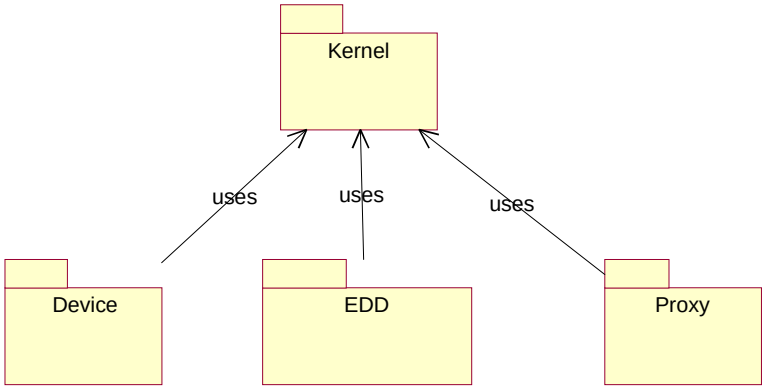


Abbildung 2. Package-Übersicht für verteiltes Automatisierungssystem (Übersicht)

Die Packages Device, EDD (Electronic Device Description) und Proxy sind ausgewählte Repräsentationen von Komponenten der verteilten Systeme. Device sind die Feldgeräte, EDD die Gerätebeschreibung der Feldgeräte für die Inbetriebnahme und Proxy der Stellvertreter der Feldgeräte in der Steuerungsbibliothek, wie in Abbildung 1 dargestellt. Von ihrem Wesen repräsentieren diese Packages sehr unterschiedliche Komponenten. Im Device werden zur Betriebsphase Funktionen ausgeführt. Die EDD ist eine Instanzbeschreibung des deklarativen Anteils der Geräte und kommt in der Regel in Inbetriebnahmewerkzeugen zum Einsatz. Der Proxy ist zur Betriebsphase aktiv und organisiert den Daten-

transfer zwischen Steuerung und Feldgerät so, als ob die Feldgerätefunktion in der Steuerungsbibliothek abgearbeitet würde. Beruhen diese Komponenten auf unterschiedlichen Modellen, so ist erheblicher Aufwand für die Transformation des Entwurfes in die einzelnen Komponenten notwendig. Das folgende Top-Down-Vorgehen soll zeigen, wie ein gesamtheitlicher Strukturentwurf von Komponenten des verteilten Systems durchgeführt werden könnte.

Grundregeln für den Modellentwurf

Folgende Regeln bei der Nutzung der UML-Klassendiagramme wurden bei der Notation des Modells eingehalten:

- Zur Strukturierung des verteilten Systems (verteilt zur Laufzeit, aber auch verteilt zusammenarbeitende Werkzeuge für das Engineering) werden UML-Packages verwendet. Packages werden im Sinne von Softwarekomponenten eines Projektes aber auch zur Abgrenzung verschiedener Betrachtungsräume (z.B. Feldgerät, Inbetriebnahmewerkzeug oder SPS-Feldgeräte-Proxy) verwendet.
- Für Klassen werden nur Operationen angegeben, wenn diese direkt zur automatisierungstechnischen Funktion der Klasse gehören. Operationen wie z.B. der Zugriff auf die Liste der Attribute der Klasse und der Werte der Attribute sind nicht explizit aufgeführt, jedoch im Sinne des Workflows der Systemkomponenten eindeutig definiert. Der Aspekt des Workflows ist in diesem Positionspapier nicht dargestellt.
- Die Klassen, die die Grundstruktur des Systems beschreiben, sind abstrakte Klassen. Dieser Kern (Kernel) setzt eine Grundannahme; die automatisierungstechnischen Funktionen sind in sogenannten Funktionsblöcken (FB) gekapselt, die ihrerseits in Geräten enthalten sind. Diese Annahme ermöglicht die Übertragung der sehr verbreiteten Funktionsblockkonfigurierung der Anwendung auf Gerätetopologien mit frei verteilbaren FBs. (Anmerkung: Es wird weiter angenommen, daß es Geräte mit fester Funktionalität, d.h. vorkonfigurierten FB Instanzen, Geräte mit fest definierten aber frei instantiierbaren FB-Typen und frei definierbaren und instantiierbaren Typen gibt.)
- Spezialisierte Variablen sind Klassen, da deren Attribute nur eine Teilmenge, der in realen Geräten implementierten Eigenschaften widerspiegeln. Es gibt die vielfältigsten Implementierungsmöglichkeiten.
- Manche Spezialisierung erscheint unbegründet (z.B. bei Blöcken), da sich die Klassen nur durch unterschiedliche Klassennamen unterscheiden. Die Ursache liegt in den nicht dargestellten Attributen und Operationen, die sich durch die dahinter liegenden automatisierungstechnischen Algorithmen wesentlich unterscheiden. Diese Methoden sind wiederum Klassenmethoden, da die Implementierung sehr vielfältig ausgeführt werden kann.

Kernel (Abbildung 3**)**

Im Kernel ist bereits der Modellansatz für die Instrumentierung vollständig enthalten. Die anwendungsbezogenen Variablen-, Funktions- und Blockklassen (CVariable, CBlock, CFunction) werden in die Gerätestruktur eingebettet. Die Gerätestruktur wird als blockorientiert postuliert. Dieses Postulat entspricht dem derzeitigen Trend auf dem Gebiet der Automatisierungstechnik [5] sowie kommerziellen Aktivitäten (z.B. PROFIBUS, IEC1131-3, Foundation Fieldbus).

Besonders hervorzuheben ist, daß Elemente der Plattform der Automatisierungsgeräte (Physical Block repräsentiert die Hardware und das Betriebssystem der Geräte und Transducer Block die Ankopplung der Geräte an die physikalische Welt, z.B. an Sensor, Stellantriebe, Bedienterminals) ebenfalls als Blöcke (CTranducerBlock, ...) betrachtet werden.

Automatisierungsfunktionen (CFunction) sind nur durch ihre Namen im Modell charakterisiert, da der Algorithmenentwurf und die Ableitung der automatisierungstechnischen Funktionen aus dem gewünschten Prozeßverhalten kein Betrachtungsgegenstand sind. Außerdem sind die Funktionen meist branchenspezifisches Know-how und werden nicht öffentlich kommuniziert.

Für den Lebenszyklus verteilter Automatisierungssysteme ist die eindeutige Identifikation von Geräten von wesentlicher Bedeutung. Deshalb sind entsprechende Geräteattribute (Device_Vendor, Device_Model, Device_Revison und Device_Ser_No) untrennbar mit dem Gerät (CDevice) verbunden und für alle zwingend.

Für bestimmte Phasen des Lebenszyklusses werden die konkrete Struktur und die instantiierten Objekte des Gerätes detailliert benötigt (z.B. bei der Inbetriebnahme). Diese Informationen müssen maschinenlesbar zur Verfügung stehen. Diese Aufgabe wird von der sogenannten Gerätebeschreibung [2] übernommen, die je nach Einsatzgebiet unterschiedlich sein kann. Sie ist deshalb abhängig vom konkrete Gerät und durch ihre Sprache/Sprachversion gekennzeichnet (CGeneral). Der Autor ist ein oft verwendetes Identifizierungsmittel von Gerätebeschreibungen.

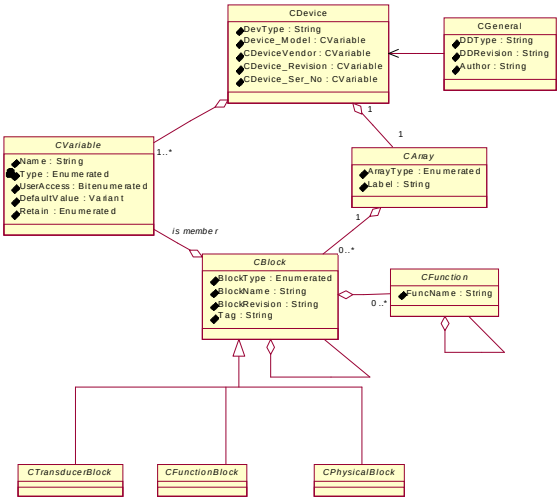


Abbildung 3. Klassendiagramm Kernel

Gerätemodell am Beispiel von PROFIBUS-PA-Geräten – Device Package (Abbildung 4)

PROFIBUS-PA ist eine FB-basierte Spezifikation für intelligente Feldgeräte. In einem entsprechenden Standard (d.h. PNO-Richtlinie (PROFIBUS Nutzerorganisation), kein internationaler Standard) werden Funktionsblöcke, deren Variablen und z.T. die im FB abzuarbeitenden Algorithmen festegelegt [1]. Außerdem wird die Abbildung der Variablen auf den Feldbus PROFIBUS beschrieben. Dabei ist hervorzuheben, daß die Variablen in der Regel in Strukturen gelesen und geschrieben werden, um Konsistenz zwischen bestimmten Variablen und Busentlastung zu erreichen. Die Gerätevariable (Device Variable) ist im Sinne des FBs nur in der PROFIBUS-PA-spezifischen Struktur (Collection) sichtbar. Dieser Sachverhalt und die Adressierung der Collections für den Zugriff sind im Klassendiagramm (siehe Abbildung 4) dargestellt. Aus der Klassenstruktur ist zu ersehen, wie die FB-Spezifikation in die Gerätestruktur überführt werden muß.

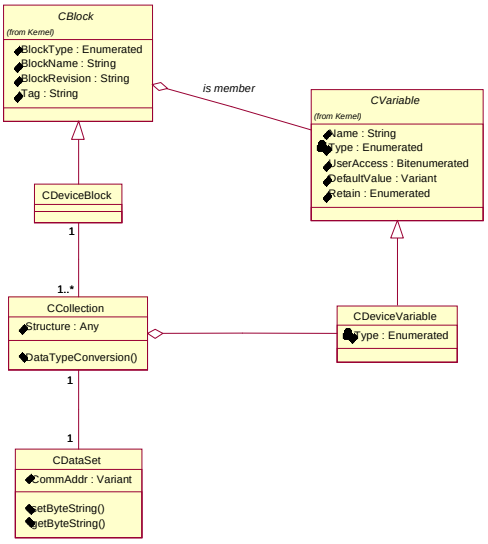


Abbildung 4. Klassendiagramm eines PROFIBUS-PA-Gerätes (Ausschnitt)

Gerätebeschreibung am Beispiel von PROFIBUS EDD (Electronic Device Description) (Abbildung 5**)**

Die Gerätebeschreibung eines Feldgerätes wird benötigt, um z.B. Inbetriebnahmewerkzeuge mit den in einem Feldgerät instantiierten FBs und Variablen bekanntzumachen. Eigens dafür existieren Beschreibungssprachen [3]. PROFIBUS überführt z.Z. die Sprache Device Description Language, entwickelt für das nicht mehr existierende Firmenkonsortium ISP (InterOperable Systems Project), als Electronic Device Description (EDD) zum PNO-Standard [2]. Im Klassenmodell (Abbildung 5) sind einige Sprachelemente und deren Beziehungen zum Kernel dargestellt. Auch hier sind eindeutige Beziehungen zwischen funktionalem Modell des Kernels und den Sprachelementen zu erkennen.

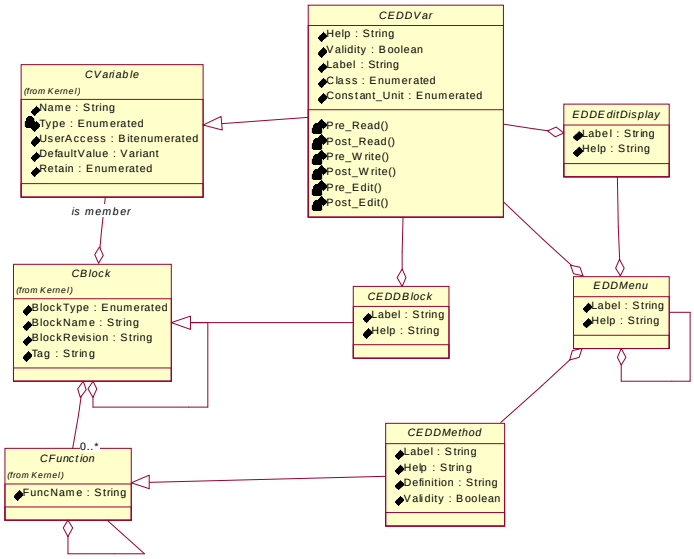


Abbildung 5. Klassendiagramm PROFIBUS-EDD (Ausschnitt)

Die Klassen EDD Var, EDD Block und EDD Method erweitern die Attribute und Operationen ihrer Eltern Klassen. Praktisch bedeutet dies, daß die Sprachdefinition exakt auf dem Gerätemodell des Kernels beruht.

Proxy-Beispiel für SPSen (Abbildung 6**)**

SPS-Sprachen, z.B. der internationale Standard IEC61131-3 haben in der Regel die Möglichkeit, mittels Funktionsblöcke zu progammieren. Die Syntax und z.T. auch die Semantik sind aber jeweils einem Sprachdialekt immanent. Eine Übernahme der Modellstruktur des Kernels ist deshalb nicht in jedem Fall möglich. Seitens der Sprache von IEC 61131-3 –Funktionsblock Diagramm ist dies jedoch für Variablen und Blocks möglich, wenn an die Blocks keine erweiterten Anforderungen gestellt werden (wie z.B. bei IEC 61499). Der Proxy enthält deshalb eine Datentypkonvertierung und eine Zuordnung von Variablen und Funktionsblöcken. Die Attribute der Klassen des Kernels sind so ausgewählt, daß dieser Minimalsatz von allen beteiligten Klassen unterstützt werden kann. Redefinitionen, wie z.B. beim Datentyp der Variablen sind aber teilweise notwendig, was in der Implementierung durch Konvertierungsfunktionen umgesetzt wird.

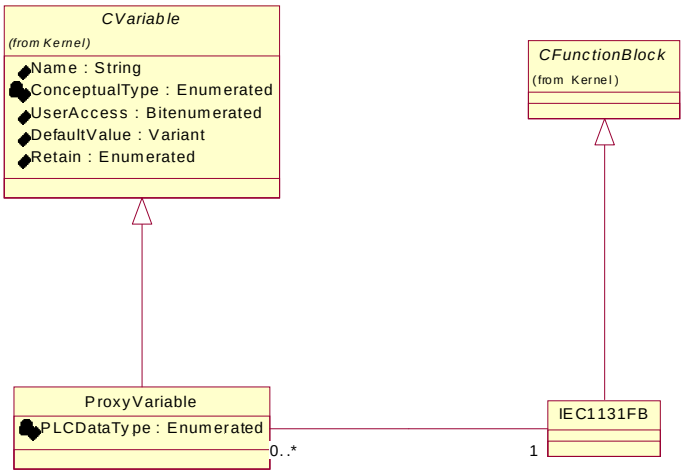


Abbildung 6. Klassendiagramm Proxy

ZUSAMMENFASSUNG

Es wird ein Modell in Form von UML-Klassendiagrammen vorgestellt, das die statische Struktur verteilter Automatisierungssysteme aus unterschiedlichen Sichten enthält. Es wurden beispielhafte Sichten ausgewählt, die einen Beitrag zur Programmierung, Konfigurierung und Parametrierung von Komponenten des Systems leisten. Die Zusammenhänge zwischen den unterschiedlichen Sichten können an den Klassendiagrammen eindeutig gezeigt werden. Daraus sind Anforderungen und z.T. Unterstützungen für die Toolentwicklungen ableitbar.

Am Klassenmodell des Packages Kernel wurde das funktionsblockbasierte Gerätemodell und an weiteren Klassenmodellen die Zusammenhänge zwischen den beteiligten Komponenten dargestellt. Das Top-Down-Vorgehen soll zeigen, wie ein gesamtheitlicher Strukturentwurf von den Komponenten des verteilten Systems durchgeführt werden könnte. Dieses Vorgehen ist in der Praxis kaum realisierbar, da in der Regel auf Strukturen existierender Komponenten aufgesetzt werden muß. Ein Button-up Vorgehen, d.h. eine Darstellung der existierenden Komponenten als UML-Klassenmodell zeigt Modellbruchstellen auf. Diese Bruchstellen sind in den Softwarewerkzeugen des Engineering und in der Programmbibliothek durch zusätzliche Funktionen zu kompensieren.

LITERATUR

[1] PROFIBUS-PA: Profile for field devices Version 3.0, PROFIBUS User Organisation, Karlsruhe 1999.

[2] EDD: Draft Electronic Device Description Specification V.09, PROFIBUS User Organisation, Karlsruhe 1999.

[3] Diedrich, Ch.; Simon, R.; Riedl, M.: Device Description Technology. Interkama Conference 1999, accepted paper.

[4] Diedrich, Ch.; Neumann, P.: Field device integration in DCS engineering using a device IECON'98, IEEE conference Aachen September 1998.

[5] Meyer, D.; Epple, U.: Realisierung leittechnischer Applikationen auf derBasis standardisierter objektorientierter Modelle. Regelungstechnisches Kolloquium, Boppard 1999.

[6] Quartani, T.: Visual Modeling with Rational Rose and UML. Addison-Wesley Longman, Inc. 1998.

Panel: OO-Development of RT Systems

Our Needs and Future Standards

Andy Schürr
Institute for Software Technology
University of the German Federal Armed Forces, Munich
85577 Neubiberg, Germany

The OMER workshops shows that many companies, which are involved in embedded realtime system development, are planning to jump onto the OO-bandwagon. Some of them have already made their first experiences with OO modeling languages such as UML or ROOM and their accompanying dialects of class/component diagrams, Statecharts, and Message Sequence Charts. From the experience reports presented here one can infer that the OO approach is used in rather different ways: In some cases only "flat" class diagrams are employed for modeling the static structure of a system; inheritance is either not used at all or plays a modest role for modeling compile time variants of developed subsystems. In other cases first experiences with the usage of behavior modeling diagrams such as Message Sequence Charts and Statecharts are reported.

It is rather obvious that many topics still need to be addressed until OO modeling languages and tools are widely accepted for mainstream development of embedded realtime systems. First of all the OMG standard UML must be extended for the development of distributed-object-based realtime, embedded and fault-tolerant applications. This is the mission of the OMG "RT Analysis and Design Task Force", which was founded in March 1998. Second, most OO-CASE tools do not yet offer adequate support for in-depth analysis (verification), simulation, and highly efficient code generation, as expected in this application domain. Third, a company has to face many technical and nontechnical problems, when it plans to switch its whole RT system development process from a traditional structure-oriented approach to the new object-oriented approach.

It is the purpose of this panel to address the problems mentioned above with a main focus on (1) the development of **future** OO modeling standards in this field and their relationships such as beween the planned RT extension of UML and the ITU-T standards MSC and SDL and (2) the problems companies have to face **today**, when they are planning to replace the traditional RT system development approach by an OO-driven approach on a wide scale.

These topics will be discussed from the following five different points of view: (1) definition of needed formalisms, (2) design of modeling languages, (3) implementation of CASE tools, (4) technology transfer, and (5) application development.

Panelists:

- Manfred Broy, TU Munich
- David Harel, Weizmann Institute of Science
- Rudolf Resch, Berner & Mattner GmbH
- Bran Selic, ObjecTime Ltd.
- N. N.

Using UML to Model Complex Real-Time Architectures

Extended Abstract

Bran Selic
ObjecTime Limited
Kanata, Ontario, CANADA
bran@objectime.com

1 Introduction

The term "software architecture" is widely used but infrequently specified. We define it as: *the organization of significant software components interacting through interfaces, those components being composed of successively smaller components and interfaces*[1].

Note that, when dealing with architectures, we are only interested in *high-level* features. Architecture necessarily involves abstraction — the elimination of irrelevant detail so that we can better cope with complexity.

The term "organization" pertains to the high-level structure of a system: its decomposition into parts and their mutual relationships (interaction channels, hierarchical containment, etc.). However, architecture is more than just structure. It includes rules on how system functionality is achieved across the structure. This includes high-level end-to-end behavioral sequences by which a system realizes its use cases.

Structure and behavior come together in interfaces, which, appropriately, also play a fundamental role in the specification of a software architecture. The last part of the definition above is simply a recursion of the first part. This tells us that architecture is a relative concept. It does not just occur at the "top" level of decomposition. A system may be decomposed into a set of subsystems, each of which, in turn, can be viewed as a system in its own right etc.

Software architectures play a pivotal role in two types of situations. During initial development of a software system, an architecture is used to separate responsibilities and distribute work across multiple development teams. If the architecture is well defined, it should be straightforward to put the individually developed parts together and complete the system. Unfortunately, properly specified architectures are rare, which explains why there is a large incidence of so-called "integration" problems. (Almost invariably, an integration problem is the result either of an inadequate architecture or an inadequately specified architecture.)

However, even more importantly, software architectures play a crucial role in system evolution. It can be safely said that the architecture is the fundamental determinant of a system's capacity to undergo evolutionary change. A flexible architecture with loosely coupled components is much more likely to accommodate new feature requirements than one that has been highly optimized for just its initial set of requirements.

2 Requirements for Architectural Modeling

Complex systems often have complex architectures. An architectural modeling language must have the range to define all the necessary elements of that in a clear and unambiguous manner.

Most complex software systems are structured in some layered fashion. Layering is one of the most common ways of dealing with complexity. Yet, no standard programming language used in industry provides the concept of a "layer" as a first class concept. Instead, layering is simulated in a variety of ways, such as the use of compilation module boundaries, none of which are adequate to precisely capture the subtle semantics of layering. For instance, although layered architectures are often depicted as simple

[1] I owe this definition to Grady Booch.

"onion-peel" structures, the layering in most real systems is far more complicated. Consider, for example, the well-known seven-layer architecture of ISO's Open System Interconnection standard. This architecture deals exclusively with communication aspects of a system. It is typically implemented as an application on top of an operating system, which represents an orthogonal layering hierarchy. In other words, complex systems require multiple dimensions of layering — two are never enough.

Another issue associated with architectures is the potential for reuse. Many modern systems are built around the "product line" concept. A product line is a set of different products that are built around a common abstract architecture. This architecture is then refined in different ways to realize individual products. This leads us to the requirement for subclassing at the architectural level. Needless to say, with very poor support for high-level concepts such as layering, no standard programming language in use today supports such a capability.

It is clear that, we need a new breed of specification languages that is capable of addressing these requirements.

3 An Architectural Modeling Language

We define an architectural modeling language building on the ideas of the ROOM modeling language [6]. A part of ROOM was specifically designed for modeling architectures of complex real-time systems. However, we will expand on those ideas and, furthermore, express them in the industry-standard Unified Modeling Language (UML) [1][2][3][4]. This allows us to take advantage of semantics and notation that are widely recognized by software practitioners. Specifically, we use the extensibility[2] mechanisms of UML: stereotypes, tagged values, and constraints. In other words, we define our specific architectural modeling concepts as specializations of generic UML concepts. These specializations, usually expressed as stereotypes, conform to the generic semantics of the corresponding UML concepts, but provide additional semantics specified by constraints.

3.1 Capsules

The basic concept is that of an architectural object called a *capsule*. A capsule is a stereotype of the UML class concept with some specific features:

- it is always an active object, which means that it has behavior that executes concurrently with other behaviors
- it has an encapsulation shell such that it not only hides the implementation from outside observers, but also prevents the implementation from directly (and arbitrarily) accessing the external environment; in other words, it is an encapsulation shell that works in both directions[3]
- it may be a truly distributed object — it may even span multiple physical nodes; this makes it suitable for modeling physically distributed conceptual entities

Capsules are used to capture major architectural components of real-time systems. For instance, a capsule may be used to capture an entire subsystem, or even a complete system. As we shall see later on, a capsule may encapsulate any number of internal capsules.

One of the interesting features of capsules is that they can have multiple interfaces, called *ports*. Each interface presents a distinct aspect of a capsule. Different collaborators can access different interfaces, possibly in parallel. We shall describe ports in more detail in the following section.

A capsule uses its ports for *all* interactions with its environment. The communication is done using signals, which can be used to carry both synchronous and asynchronous interactions. The advantage of signals as

---

[2] The term "extensibility mechanisms" is somewhat misleading since they are really used not to introduce new concepts, but to allow the definition of specialized versions of existing ones.

[3] In classical OO languages, this is not the case: while the implementation is hidden from external entities, the implementation has unhindered access to the environment. This creates problems since a "component" in some class library may be very tightly coupled with other entities. Unfortunately, this coupling can only be determined by careful inspection of the implementation.

the underlying communication vehicle is that, in contrast to communications based on procedure calls, they are more amenable to distribution.

Because capsules are a kind of class, they can be subclassed. This gives us the capability to produce variations and refinements of architectural components and even entire architectures

3.2 Ports

In contrast to interfaces one finds in traditional object-oriented programming languages, ports are distinct objects (stereotypes of the UML class concept). They convey signals between the environment and the capsule. The types of signals and the order in which they may appear is defined by the *protocol* associated with the port. Protocols are discussed in the next section.

Ports are used not only for receiving incoming signals, but also for sending all outgoing signals. This means that a capsule is fully encapsulated: internal components never reference an external entity, but only communicate through ports. Consequently, a capsule class is fully self contained — which makes it a truly reusable component.

3.3 Protocols

A *protocol* specifies a set of valid behaviors (signal exchanges) between two or more collaborating capsules. However, to make such a dynamic pattern reusable, protocols are de-coupled from a particular context of collaborating capsules and are defined instead in terms of abstract entities called *protocol roles*. A protocol role represents the activities of one participant a protocol. Formally, it is defined by a set of incoming signals, a set of outgoing signals, and an optional behavior specification. The behavior specification represents that subset of the behavior specified for the overall protocol that directly involves this role.

A particularly common type of protocols are *binary* protocols, which involve only two roles. For binary protocols it is sufficient to define one protocol role to define the entire protocol.

The relationship between ports and protocols is crucial: each port plays exactly one role in some protocol. The protocol role of a port represents the type of the port.

Protocols are modeled in UML as a stereotype of the collaboration concept. Since collaborations are generalizable elements, they too can be refined using inheritance-like mechanisms to produce variations and refinements.

Protocol roles are stereotypes of the classifier role concept in UML.

3.4 Connectors

To allow capsules to be combined into aggregates, we define the concept of a *connector*. A connector is an abstraction of a message-passing channel that connects two or more ports. Each connector is typed by a protocol that defines the possible interactions that can take place across that connector.

Connectors are stereotypes of the association class concept in UML with the added constraints that they can only pass signals and that their behavior is governed by a protocol.

3.5 Composite Capsules

Using the simple concepts of capsules, ports, and connectors, we can easily assemble complex aggregates of diverse capsules that achieve complex functionality. The design paradigm behind this is very similar to the design of hardware. Complex systems emerge by interconnecting simpler specialized parts drawn from a component catalog.

It is often the case that we may need to reuse a particular object composition pattern in a variety of different situations. In other words, we would like to make these assemblies into components in their own right. For this purpose, the object paradigm gives us the class concept. A particularly convenient way of realizing such a class is to define it as a capsule. This makes the capsule concept recursive: a capsule can be

decomposed into lesser capsules, and so on. There are no theoretical limits to this hierarchy, capsules can be nested to whatever extent is necessary to realize our desired system. Note that such composite capsules can have their own ports, like any other capsules. These ports may be used to delegate functionality, using an internal connector, to some internal component. Such ports are called *relay* ports since their function is simply to act as a funnel for signal traffic. Alternatively, a port may be connected directly to a state machine. This type of port maintains a queue of incoming signals that have been received but not yet processed by the state machine. These ports are called *end* ports.

A composite capsule, therefore, represents a network of collaborating capsules. A particularly important feature of composite capsules is their *assertion semantics*. What this means is that when a composite capsule is instantiated, all of its internal nested parts are *automatically* instantiated along with it. Furthermore, since the internal capsules may themselves be composites, an entire system can be created with just a single "create" action. The consequence of assertion semantics is that the designer is liberated from the often tedious and highly error prone task of having to write the explicit code that generates the complex internal structure piece by piece. This not only saves effort, but also provides a tremendous boost in reliability, because in many dynamic systems, the code used to establish a structure can represent a major percentage of the overall software.

4 Summary

Architecture plays a key role in the design of complex real-time systems. In order to specify the types of architectures that are common in this domain, we need a suitable set of modeling concepts. We have presented here a very simple set of modeling concepts consisting of capsules, ports, protocols, and connectors, that has already been proven effective in industrial practice. To make these modeling capabilities available to the broadest set of users possible and to take advantage of commercial tools, we have expressed them using the industry-standard Unified Modeling Language. For this purpose, we have used the extensibility mechanisms of UML. These allowed us to very effectively capture the full semantics of the concepts in a manner consistent with the general semantics of UML.

Acknowledgement

The author recognizes the major contribution of Jim Rumbaugh of Rational Software Corporation who collaborated very closely with the author on the work reported here.

References

[1] Booch, G., J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley Publishing Co., 1998.

[2] OMG, *UML Semantics (Version 1.1)*, The Object Management Group, Doc. ad/97-08-04. Framingham MA, 1997.

[3] OMG, *UML Notation Guide. Version 1.1*, The Object Management Group, Doc. ad/97-08-05. Framingham MA, 1997.

[4] Rumbaugh, J., I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley Publishing Co., 1998.

[5] Selic, B. and J. Rumbaugh, "Using UML for Modeling Complex Real-Time Systems," ObjecTime Limited/Rational Software Corp. white paper, March 1998.

[6] Selic, B., G. Gullekson, and P. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons, New York, NY, 1994.

LSC'S: BREATHING LIFE INTO MESSAGE SEQUENCE CHARTS

Werner Damm[1] and David Harel[2]

extended version presented at FMOODS 99:

W.Damm and D. Harel
Breathing Life into MSCs
Proceedings of the IFIP TC6, Third International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS)
Editors P. Ciancarini, A. Fantechi, R. Gorrieri
pages 293-312, Kluwer Academic Publishers, 1999

Summary

Message sequence charts (MSCs) are a popular visual medium for the description of scenarios that capture the typical interworking of processes or objects. They are particularly useful in the early stages of system development. There is also a standard for the MSC language, which has appeared as a recommendation of the CCITT [Z120]; it defines the allowed syntactic constructs rigorously, and is also accompanied by a formal semantics [Z120S] that provides unambiguous meaning to basic MSCs in a process algebraic style. Other efforts at defining a rigorous syntax and semantics for MSCs have been made [GGR93, LL95], and some tools supporting their analysis are available [AHP96a, AHP96b, BL97a].

Surprisingly, despite the widespread use of the charts themselves and the more rigorous foundational efforts cited above, several fundamental issues have been left unaddressed. One of the most basic of these is, quoting [BL97b]: "What does an MSC specification mean: does it describe all behaviors of a system, or does it describe a set of sample behaviors of a system?". While typically MSCs are used to capture sample scenarios corresponding to use-cases [Jac92, BJR96] , as the system model becomes refined and conditions characterizing use-cases evolve, the intended interpretation often undergoes a metamorphosis from an existential to a universal view: earlier one

[1]OFFIS, Oldenburg, Germany. E-mail: damm@offis.uni-oldenburg.de
[2]The Weizmann Institute of Science, Rehovot, Israel. E-mail: harel@wisdom.weizmann.ac.il

wants to say that a condition can become true and that when true the scenario can happen, but later on one wants to say that if the condition characterizing the use-case indeed becomes true the system must adhere to the scenario described in the chart. Thus, we want to be able to specify liveness in our scenarios, that is, mandatory behavior, and not only provisional behavior.

In fact, the confusion between necessity and possibility arises even within a basic MSC itself: should edges of an MSC prescribe only (partial) ordering constraints, or should they entail causality? While the standard [Z120S] views the semantics of MSCs as merely imposing restrictions on the ordering of events, designers are often interested in shifting the intended meaning depending on the current design level. And this, again, means preferring initially an existential interpretation, but transforming these into necessity interpretations as design details are added, thus enforcing messages to be sent and received, progress to be made, etc. We feel that the lack of variety in the semantic support of conditions in the CCITT standard may well have contributed to its inability to distinguish between possibility and necessity.

There are other inadequaces in the CCITT standard, and for that matter in most other proposals for a rigorous MSC language too. They involve the subtle relationship between timing constraints and the synchronization of events in different object instances, and the lack of an adequate hierarchical structuring mechanism and conventional control constructs such as branching and iteration.

For all of these, we feel the dire need for a highly expressive MSC language with a clear and usable syntax and a fully worked out formal semantics. Such a language is needed in order to construct semantically meaningful computerized tools for describing and analyzing use-cases and scenarios. It is also a prerequisite to a thorough investigation of what we consider to be one of the central problems in the behavioral specification of systems: relating inter-object specification to intra-object specification. The former is what engineers will typically do in the early stages of behavioral modeling; namely, they come up with use-cases and the scenarios that capture them, specifying the interrelationships between the processes and object instances in a linear or quasi-linear fashion in terms of temporal progress. That is, they come up with the description of the scenarios, or "stories" that the system will support, each one involving all the relevant instances. An MSC language is best used for this. The latter, on the other hand, is what we would like the final stages of behavioral modeling to end up with; namely, a full behavioral specification of each of the processes or object instances. That is, we want a complete description of the behavior of each of the instances under all possible conditions and in all possible "stories". For this, a state-machine language (such as statecharts [HAR87, HG97]) appears to be most useful. The reason the state-machine intra-object model is what we want as an output from the design stage is for implementation purposes: ultimately, the final software will consist of code for each process or object. These pieces of code, one for each process or object instance, must - together - support the scenarios as specified in the MSCs. Thus the "all relevant parts

of stories for one object" descriptions must implement the "one story for all relevant objects" descriptions. Investigating the two-way relationship between these dual views of behavioral description is an ultimate goal of our work.

How to address this grand dichotomy of reactive behavior, as we like to call it, is a major problem. For example, how can we synthesize a good first approximation of the statecharts from the MSCs? Finding good ways to do this would constitute a significant advance in the automation and reliability of system development. However, this problem cannot even be contemplated properly without a powerful MSC language.

In this paper we propose a language for scenarios, termed live sequence charts, or LSCs for short. LSCs constitute a smooth extension of the CCITT standard for MSCs along several fronts. We allow the user to selectively designate parts of a chart, or even the whole chart itself, as live, or necessary, thus forcing messages to be sent, conditions to become true, etc. By taking the existential interpretation as a default, the designer may incrementally add liveness annotations as knowledge about the system evolves. Hand in hand with this extension comes the need to support conditions as first-class citizens: we assume availability of interface definitions for instances, containing events that can be sent and received, and also variables that may be referred to when defining (first-order) conditions. By associating activation conditions with an LSC, a live interpretation of the chart becomes more significant; it now means, informally, that whenever the system satisfies the chart's activation condition its behavior must conform to that prescribed by the chart. As we shall see, live elements (we call them hot) also make it possible to define forbidden scenarios, i.e., ones that are not allowed to happen - a very important need for the engineer at the early stages of behavioral modeling.

Another use of LSCs, indeed one of our motivations for the present work, comes from the UML standard [UML97], which recommends statecharts as well as sequence-charts for modeling behavior, but says little about the precise relationships between the two. The Rhapsody tool from i-Logix is based on the language set for executable object modeling (XOM) defined in [HG97]. This set has become the executable kernel of the 1.1 version of UML, and as thus can be regarded as UML's definitive rigorous core. It consists of the constructive languages of object-model diagrams and state-charts, and allows a variant of MSCs as a descriptive language. The work presented in this paper provides the semantical basis for rigorous and complete consistency checks between these descriptive and the constructive views of the system. Such checks could eventually be made using formal verification techniques like model-checking [BCM+92,BCMD90]. (Some of the ideas of this paper were indeed inspired by the symbolic timing diagrams of [Sch98, SD93,FJ97,SJW98], used to specify and verify safety-critical requirements for systems modeled using Statemate; see [DEB+97, DJHP98, BW98, BW98b, HK94].)

Combining Statecharts and Collaboration Diagrams for the Development of Production Control Systems

Position Paper

Ulrich Nickel, Jörg Niere, Wilhelm Schäfer, Albert Zündorf
AG-Softwaretechnik
University of Paderborn, Germany
[duke|nierej|wilhelm|zuendorf]@uni-paderborn.de
D-33095 Paderborn

1 Motivation

Todays manufacturing industry faces big challenges. The market demands a very flexible production process which adjusts quickly to e.g. changing characteristics of the produced goods or different lot sizes and which tries to minimize production costs as much possible. In order to meet these challenges todays production control systems become more and more decentralized [GBFG96]. The building blocks of such a production control system are different, selfacting and computer controlled resources like e.g. switches, shuttles, machines or robots.

One major problem in building those decentralized systems is to develop a detailed and precise plan how those different components interact and in particular how their control software has to behave. There is no specification approach which allows to define, analyze and simulate such a production control system upfront before it is actually built, physically. With regard to the frequent changes, this is of course an important cost factor which should be avoided.

Such a production control system may be regarded as a system of active autonomous objects. These objects communicate with each other to synchronize their concurrently executed activities. We use state-charts [Har84] to specify the behaviour of the objects and SDL [ITU96] to specify the communication aspects. Both are very popular in mechanical and electronic engineering science because they fit the demands for the specification of systems e.g. in the above picture. Many specification tools generate a C-dialect code out of specifications, and the correctness of the specification must be tested by a physical simulation.

We decided to use the FUJABA environment [FNT98, FNTZ99] to specify production control systems. FUJABA is an integrated development environment for UML class diagrams, collaboration diagrams, and Story-Diagrams. The environment allows to generate executable Java code out of specifications and to simulate the execution. Furthermore, the environment is currently extended to state-charts and SDL.

2 Integrating State-Charts and UML Collaboration Diagrams

Figure 1 shows a scenario of a sample factory. The factory is modeled as a flat building without levels and pillars in it. The floor is layered with rectangle shaped fields allowing to address certain positions in the building. The factory contains certain kinds of production places. A production place is e.g. an assembly line, where goods arrive and are loaded on shuttles, or a storage, where goods can be stored.

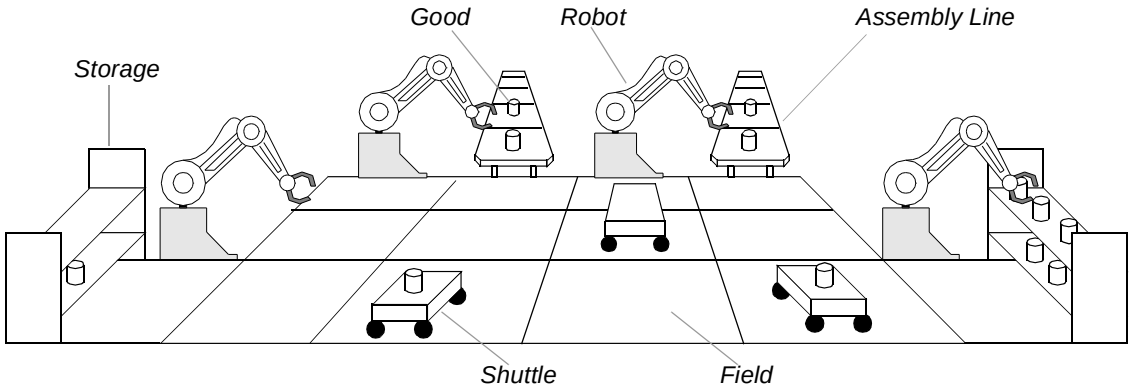


Figure 1 Simple factory example

State-charts provide sophisticated means for the specification of (concurrent) control flows for reactive objects (shuttle, assembly line, storage). However, by purpose state-charts abstract from the complex application specific data structures that build up the concrete states of a system. State-charts do not explicitly deal with values of attributes or with links to other objects nor with the evolution and changes of these object-structures caused by the execution of operations or action methods.

The specification of application specific object-structures is a well known application area for graph grammars, cf. [Roz97]. Basically, a graph grammar rule allows the specification of changes to complex-object-structures by a pair of before and after snapshots. The before snapshot specifies which part of the object-structure should be changed and the after snapshot specifies how it should look like afterwards, without taking care of regarding how these changes are achieved. While graph grammars are appropriate for the specification of object-structure modifications, they lack appropriate means for the specification of control flows.

To overcome these problems, we propose to combine state-charts and graph-rewriting rules. We use state-charts (and activity diagrams) to specify complex control flows and graph rewriting rules to specify the entry, exit, do, and transition actions that deal with complex object-structures. In order to facilitate the use of graph rewriting rules for object-oriented designers and programmers, we propose to adopt UML collaboration diagrams as a notation for object-structure rewriting rules.

Figure 2 shows an example for the combination of state-charts (activity diagrams) and object-structure rewrite rules within an UML collaboration diagram. The left half of Figure 2 shows a cut-out of the state-chart of class shuttle, namely its fetch state. The initial substate of state fetch is startFetch. In our example, the do-action of state startFetch is specified via an object-structure rewriting rule that shows three objects, this, f, and al. The this object is attached to the Field object f via an at link and the

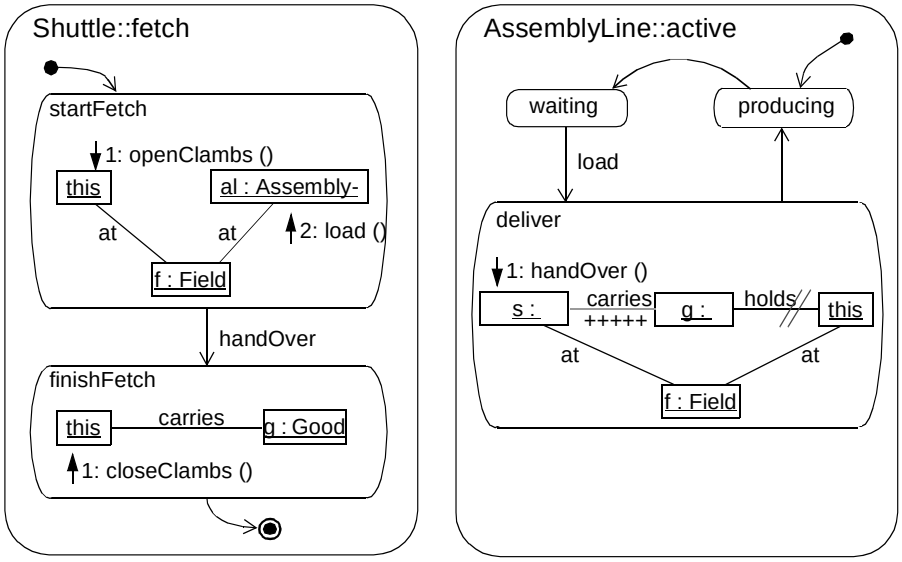


Figure 2 Refinement of state fetch

same holds for the AssemblyLine object al. We interpret this, al and f as variables and the shown links as logical constraints. Such a diagram is executed by binding the specified variables to concrete object instances such that all specified constraints are fulfilled.

In our example variable f is bound to the field object that is connected to the current shuttle object (bound to variable this) via an at link. In turn, al is bound to the AssemblyLine object that is attached to that field object via an at link. Once all variables are bound to appropriate object instances the specified change effects and method invocations are executed. State startFetch shows no object-structure changes but two method invocations (the do-action of state startFetch is a pure collaboration diagram specified in UML notation). Thus, in step 1 startFetch calls method openClambs on the object this, i.e. the current shuttle. In step 2, startFetch calls method load at the AssemblyLine al. After that the do-action terminates and state startFetch is ready to accept the next event. Let us assume, that the AssemblyLine object bound to variable al is in the state waiting, cf. the right-hand side of Figure 2.

Thus, the load method invocation on al generates a load event that causes al to switch to activity deliver. In this case we deliver is named an activity, because there are only triggerless outgoing transitions and to underline the parallelism between state-charts and activity diagrams. Activity deliver is again specified by an object-structure rewriting rule. In activity deliver variable this represents the current assembly line. Deliver determines its field f, the shuttle s attached to field f and a good g it holds. The activity shows two object-structure changes. First, the holds link connecting variable this and g is canceled by two small lines. This is executed by deleting the corresponding link. Second, the + symbols attached to the carries link connecting variables s and g indicates that such a link is created. Finally, activity deliver calls method handOver at shuttle s and terminates. On termination of activity deliver, the outgoing triggerless transition fires and the corresponding assembly line object switches to activity producing. Once our shuttle object receives the handOver event the corresponding transition fires and activity finishFetch is triggered. finishFetch just checks whether the shuttle actually carries a good and then it closes its clambs and terminates. Thereby, the terminal state is reached and the whole fetch activity terminates.

The FUJABA environment is able to generate Java code for such behavioural specifications[1]. For the static aspects of a system UML class diagrams are used (details, see [FNT98, FNTZ99]). The advan-

1. To be honest, currently only activity diagrams and collaboration diagrams are supported. State-charts are scheduled for August 1999.

tage of FUJABA in comparision to other tools is the opportunity to validate a specification with a graphical simulation environment before it is physically tested.

3 Simulation of the system

Figure 3 shows the graphical debuging tool of FUJABA, called Dobs (dynamic object browsing system). On the right-hand side, the screenshot shows a cut-out of the current object structure of the running program. There are four field objects (f1-f4). Each field is linked to its neighbours. The assembly line (a6) holds an instance of the class Good (g7). Both, the assembly line and the shuttle s5 are linked to the same field. Now, the shuttle can take the good to carry it to the storage. Therefore, the method load of the assembly line a5 has to be executed. In our example, the user has selected the assembly line and the tool displays the public methods of class AssemblyLine on the left part of the screenshot.

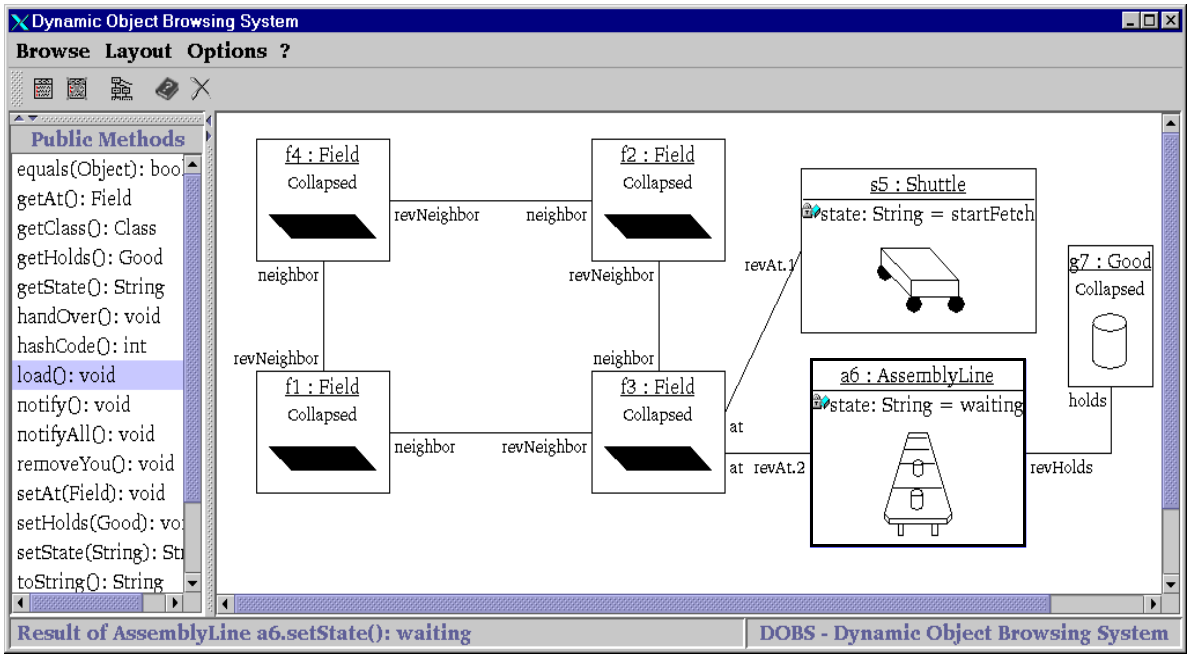


Figure 3 Dynamic Object Browser

Now, the user can invoke the method AssemblyLine::load() and the dynamic objects browser displays the changed object structure after the execution has terminated. Then, the good will have a carries link to the shuttle and the holds link to the assembly line will be deleted.

We plan to employ FUJABA for the specification, validation/simulation, and control of complex production control systems. Thus, it is indispensable to provide a simulation environment which runs automatically and allows for real time aspects. Additionally, a more sophisticated graphical representation should be implemented. The user must be able to check the states and event queues of all objects and the tool has to show the event/message flow between these objects. Last but not least, we have to animate the specified system to validate, whether it works as intended.

4 Conclusions and Future Work

FUJABA has been developed since November 1997. The first 'release' version has supported an editor for UML class diagrams and UML activity diagrams and object structure rewriting rules. It comprises a code generator and a basic consistency analyser.

Dobs, the Dynamic Object Browsing System, has become part of FUJABA in the beginning of 1998. The assistance of Design Pattern [GHJV95] recognition and instantiation is also a part of the environment (not mentioned in this paper).

As current work FUJABA is enhanced by state-charts and SDL. For both an editor, a consistency analyser, and a code generator are scheduled for autumn 1999 and spring 2000, respectively.

The described extentions of Dobs up to a graphical simulation environment are also current work and are scheduled for next year. Since the physical parts of our production control system e.g. shuttles, have to be programmed in Neuron-C and not in Java, the code generator must be accomodated for those dialects. We assume, next generation shuttles, switches, assembly-lines, toasters, televisions, etc. have an integrated Java-Chip so that the specifications can be executed directly.

References

[FNT98] T.Fischer, J. Niere, L. Torunski. *Design and Implementation of an integrated Development Environment for UML, Java, and Story Driven Modeling*, Master Thesises, Paderborn 1998 (in German)

[FNTZ99] T.Fischer, J. Niere, L. Torunski, A. Zündorf. *Story Diagrams: A New Graph Grammer Language based on the Unified Modelling Language and Java*. to appear in Proceedings of TAGT '98 (Theory and Application of Graph Transformations), LNCS, Springer 1999

[GBFG96] J. Gausemeier, H.-J. Buxbaum, S. Förste, G. Gehnen. Decentral Control Architecture for Modular Flow Systems. *Proceedings CAD/CAM Robotics and Factories of the Future*, London, England, 14. - 16. August 1996.

[GHJV95] E.Gamma, R.Helm, R. Johnson, J.Vlissides. *Design Patterns*. Addison-Weslay, 1995.

[Har84] D. Harel. *Statecharts: A visual approach to complex systems.* CS84-05, Department of Applied Mathematics, The Weizmann Institute of Science, 1984

[ITU96] ITU-T Recommendation Z.100. *Specification and Description Language (SDL).* International Telecommunication Union (ITU), Geneva, 1994 + Addendum 1996

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. *Object-Oriented Modelling and Design*. Prentice Hall, Englewood Cliffs, N. J. 07632, 1991.

[Roz97] G.Rozenberg (ed). *Handbook of Graph Grammars and Computing by Graph Transformation*. World Science, 1997.

Experiences With UML and the Rhapsody CASE Tool

Kayser-Threde GmbH
Perchtinger Str. 3
D-81379 München

Alfred Hönle

Introduction

Kayser-Threde has specialized in the development and manufacture of complex systems for aerospace, scientific and industrial applications including space systems for the International Space Station.

Currently three first software projects are developed utilizing object-oriented technology. These are the

- Master Control Processor of the SOFIA (Stratospheric Observatory For Infrared Astronomy) project, an airborne 3 m class infrared telescope in a Boeing 747.
- Material Science Laboratory - a payload of the International Space Station providing capabilities to investigate crystallization and solidification processes under µ gravity conditions.
- Vehicle Analysis Data Recording System of the NASA/ESA Lifting Body X-38 technology demonstrator for the CRV (crew return vehicle for the International Space Station)

The object-oriented approach directly supports decomposition, abstraction and hierarchy to master complexity. We have selected UML for modeling our embedded real-time software systems (with the cross-developed VxWorks as the real-time operating system for mainly VMEbus based industrial and radiation-hard processor hardware) as UML seems to become the industry-standard modeling language with a prospering future. This is in the meantime also expressed in an ESA paper about object-oriented methods.
After an evaluation phase the Kayser-Threde software department has chosen i-logix's Rhapsody CASE tool as the UML software engineering tool which is utilized together with several VxWorks cross-compilers for the different target architectures as well as the DOORS requirements management software and the PVCS version management software.

Experiences

To keep the risk for utilizing the new software technology as low as possible the first project where Rhapsody is used at KT for the *entire* software life cycle consisting of requirements model, requirements analysis, design, implementation, simulation and test phases is the relatively small (> ½ man year) X-38 Data Handling System DHS which is the core of the earlier mentioned X-38 Vehicle Analysis Data Recording System. A very important aspect is also that the available DHS hardware resources are much beyond the estimated resource requirements for the task to be performed.

Currently it is very hard to compare the new technology's resource requirements to conventional "C" programs for the same problem as we have heard anything from "needs five times the space and CPU

performance" to "OO based software outperforms the old way" from real projects of other companies. Therefore this DHS project serves also for the purpose of evaluating the possibility of using the object-oriented way for µ-controllers and radiation-hard hardware that is usually very resource-constraint.

Very surprising for us was the high detail wealthiness of the OOA model that can be found with object identification strategies during the object-oriented requirements analysis performed on the software requirements. In two of the three projects it was possible to detect some inconsistencies in the software requirements already with the OOA.
During the OOA we have not only employed object/class diagrams but also statecharts to model the dynamic behavior of parts of the system. This first-step dynamic model serves several purposes:

- Extension to the textual description in Interface Control Documents describing the behavior of the system when interfacing to other systems
- Basis for discussions with the user
- Basis for the software design
- Early simulation (supported by Rhapsody)

Interesting for us was also the fact that we seem to be one of the first companies in Germany utilizing UML/C++ basing on Rhapsody for low level hardware access as coupling of Interrupt Service Routines and Rhapsody generated State Machines is not yet implemented in the current Rhapsody version so we had to find our own way for doing this important part of real-time systems.
A short introduction:
Rhapsody State Chart transitions are controlled by the reception of "events" where these events are objects derived from a common OMEvent base class in Rhapsody. The memory for the event object is allocated by the sender and is returned to the system memory pool when the receiver thread deletes the event after it is being consumed. The problem behind coupling of ISRs and State Charts now is that it is not possible in an ISR to allocate memory with the standard "new" operation resulting in an operating system call so event objects can not be generated this way.
The next Rhapsody version V2.1 will address this according to i-logix by providing special "new" and "delete" operators for events originating from ISRs.

Particularly for the problem described above and in general for a better understanding what is going on it is very helpful that Rhapsody’s execution framework is supplied together with the tool in C++ source code. But even without this the code generation and other features are largely modifiable by the huge amount of properties that can be changed on company, project or class level.

With Rhapsody debugging occurs at two levels, the model level and the implementation level where the model level is everything one designs with Rhapsody (except Use Cases) and the implementation level is the hand-written C++ code of the single class operations where Rhapsody provides the framework for. Unfortunately this source-level debugging requires switching between Rhapsody and the VxWorks development environment Tornado with its associated CrossWind debugger.
Debugging at the model level with tracer output or graphical animation (of statecharts and MSCs) can be performed by generating instrumented code and this code actually running on the target system. Hardware independent code could first be tested on the host development system, however when it comes to implementation (C++ code) debugging one has to become familiar with a second source

level debugger besides the one of the development environment for the target system. Therefore the approach so far was to use the target hardware whenever available.

During the design we try to develop reusable software components by strictly separating hardware dependent from hardware independent classes and by holding project specific code in different classes than code related to common tasks that can potentially be reused.
Reuse of legacy code (in our case existing low level hardware code, e.g. a DMA chip library) written in C has also been performed by using Rhapsody's class importer tool and first performing a relatively easy conversion of a C module into a C++ class. However care must again be taken with Interrupt Service Routines even if they do not trigger statecharts.
We also try to use design patterns however this is not easy especially at the beginning when the system and the CASE tool itself is not fully understood. Several design patterns come already with Rhapsody so they can directly be integrated in the application design.

The personal profile of the developer is also an issue as he or she should be familiar with embedded real-time systems on one hand and OO methodology, the Unified Modeling Language, C++ and also the CASE tool on the other hand as it is not enough to just draw pictures if the technology and the tool is to be used efficiently.

Conclusion

Kayser-Threde sees identifiable benefits in the object-oriented way. So far we are also very happy about Rhapsody as it makes software development clearer and faster. Although it is sometimes said that the very first OO project needs 2-3 times the costs of the same software system being built conventionally with structured analysis, so far our first project basing fully on object-orientation and a CASE tool seems to stay within budget.
When the first project is successfully finished this technology will be widely used in future software tasks.

Real-Time UML: Eine Anregung zur Diskussion

Roland Petrasch
Fachhochschule für Technik und Wirtschaft Berlin
Fachbereich 4, Treskowallee 8, 10313 Berlin
Email: petrasch@fhtw-berlin.de

1. Einleitung und Motivation

Bei *Real-Time-Systemen* steht begrifflich der Zeitaspekt im Vordergrund, während *Embedded-Systeme* die Verbindung zwischen Soft- und Hardware hervorheben. Aus funktionaler Sicht ergibt sich weiterhin der Begriff *„reaktives System“*. „Reactive systems denote a class of interactive processes fully responsible for the sychronisation with their environment“ [Maf96]. Real-Time- und Embedded-Systeme lassen sich als non-disjunkte Klassen reaktiver Systeme bezeichnen.

Die Anwendungsbereiche, der Umfang der Nutzung und die Komplexität von reaktiven Systemen steigt stetig [Lig98], so daß sich ein Bedarf nach entsprechenden Methoden und Notationsformen ergibt. Berichte über Fehlverhalten solcher Systeme unter Software-Beteiligung motivieren zusätzlich zum systematischen Entwurf stabiler und robuster Architekturen [Pet98]. Hierzu gibt es eine Reihe theoretischer und praktischer Arbeiten, z.B. der ESPRESS-Ansatz [Egg96], die u.a. formale Spezifikationstechniken verwenden.

Aufgrund der Verbreitung objektorientierter Entwicklungsmethoden und der *Unified Modeling Language* (*UML* [Boo99]) als Notationsform für diesen Bereich, ist nun zu untersuchen, ob und wie weit sich diese für die Entwicklung von Echtzeitsystemen eignet.

Die Frage scheint durch die Literatur und Praxisberichte zunächst uneingeschränkt zugunsten der UML beantwortet worden zu sein. Dennoch ist eine kritische Diskussion angebracht, welche dieser Beitrag fördern möchte. Eine Klärung der Frage, in welchem Umfang eine Eignung der UML für die Real-Time-Systementwicklung vorliegt, kann hier allerdings nicht geliefert werden.

2. Real-Time UML: Problembereiche

Die UML stellt keine Entwicklungsmethode für Software oder gar Systeme dar, sondern gibt eine Semantik und Notationsform für Diagramme zum Zweck der Modellierung von (Software-)Systemen vor, wobei ein mehrschichtiges Metamodell *(Semantic Core Concept)* und eine Formalisierung mit graphischen Symbolen, der *OCL (Object Constraint Language)* und einer *„precise Natural Language“* erfolgt. Wichtige Diagrammtypen sind *Klassen-* und *Collaborations-Diagramme*, um Struktur und Verhalten in der Analyse- und Designphase modellieren zu können. Eine explizite Berücksichtigung der Entwicklung von Real-Time-Systemen sieht die UML nicht vor, da sie den Anspruch der Unabhängigkeit von Klassen von Anwendungsfällen und Architekturen erhebt.

Bevor weiter auf den Echtzeitaspekt eingegangen wird, seien einige allgemeine Kritikpunkte bzw. Problempotentiale der UML genannt (die Vorteile dieser einheitlichen Modellierungssprache sind hier nicht erwähnt). Die Begriffe und Diagramme sind teilweise unpräzise und inkonsistent (ein Mangel, den andere Notationsformen durchaus auch besitzen), z.B. „widerspricht die Verwendung des Begriffes ‚Action‘ im Aktivitätsdiagramm der Definition desselben Begriffes im Statechart-Diagramm ...“ [Rob98]. Weiterhin führt die Vielfalt der Diagramme dazu, daß zunächst ein Standardisierungsprozeß notwendig wird, um eine einheitliche UML-Untermenge im Projekt zu definieren. Auch die Darstellungsvielfalt innerhalb der UML-Diagrammtypen kann zu sehr unterschiedlichen Darstellungen eines Sachverhaltes führen, z.B. Klassen mit/ohne *Stereotypes*, *Tagged Values*, Methoden, Attribute bzw. *Icons*. Weiterhin existiert keine zwingende Beziehungen zwischen einigen Diagrammtypen, z.B. *State-Machine* und Klasse bzw. Methode, so daß eine Konsistenzprüfung erschwert wird [Rob98].

Um Real-Time-Systeme modellieren zu können, kann der Erweiterungsmechanismus verwendet werden: „Der Standard-UML-

Tailoring-Mechanismus basiert auf Stereotypen, *Tagged Values* sowie auf *Constraints* ...“ [Sel98]. Diese Erweiterungen lassen dann jedoch eine Konsistenzprüfung mit der UML nicht zu, da deren Semantik der UML nicht bekannt ist.

Prinzipiell ist es durchaus möglich, spezielle Anforderungen an die Modellierung von Echtzeitsystemen durch die UML zu erfüllen, z.B. durch *Timing Constraints* im *Sequence Diagram* [Eri98]. Auch ist es möglich ist, vorhandene Konzepte zur Modellierung von Echtzeitsystemen auf die UML zu übertragen, z.B. *ROOM* [Sel98].

Kritisch ist bei der Verwendung der UML für den Echtzeitbereich die fehlende Unterstützung in der Praxis eingeführter Ausdrucksmittel, z.B. das *Timing Diagram* [Dou98], welches zwar zusammen mit UML-Diagrammen verwendet werden kann, ein Abgleich dann jedoch manuell erfolgen muß. Auch die Verwendung von formalen Techniken, z.B. formale Spezifikation mit *Z* oder *Object-Z*, ist nicht Bestandteil der UML, obwohl die „chronische Krise“ [Gib94] eine Einbettung „mathematischer Techniken in den konventionellen Systementwurf“ [Hei95] besonders bei sicherheitskritischen Systemen notwendig erscheinen läßt.

3. Ausblick

Die o.g. Defizite bzw. Problempotentiale bzgl. Real-Time-UML werden Gegenstand zukünftiger Untersuchungen sein. Insbesondere eine stärkere Unterstützung beim UML-Tailoring in Hinblick auf Erweiterungen für die Entwicklung reaktiver Systeme ist wünschenswert. Abzuwarten bleibt die Entwicklung des *Unified Software Development Process* [Jac99] und dessen Interpretation.

Bei der Widerverwendung sind Design Patterns hilfreich, die sich häufig in der Form von UML-Diagrammen und Code-Beispielen (z.B. C++, Java, Ada) finden [Gam98]. Allerdings bleiben auch hier Fragen in Hinblick auf die Integration in den Entwicklungsprozeß offen.

Obwohl an dieser Stelle keine abschließende Beurteilung möglich ist, sollte das Thema Real-Time-UML vor einem Praxiseinsatz bzgl. Subset, Tailoring, Entwicklungsprozeß und UML-fremde Ergänzungen genau gegen die eigenen Projektanforderungen geprüft werden.

Literatur

[Boo99] G. Booch, J.Rumbaugh, I.Jacobson: UML Users Guide, Addison Wesley Longman, 1999

[Dou98] B.P. Douglas: Real-Time UML, Addison Wesley Longman, 1998, S. 186

[Egg96] G. Egger: ESPRESS: Ingenieurmäßige Entwicklung sicherheitsrelevanter eingebetteter Systeme, BMBF-Statusseminar Softwaretechnologie, 1996 (http://www.first.gmd.de/~espress/)

[Eri98] H.-E. Erikson, M. Penker: UML Toolkit, John Wiley & Sons New York 1998, S, 161-196

[Gam98] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns CD, Addison Wesley Longman 1995

[Gib94] W. Gibbs:Software’s chronic crisis. Scientific American (Int. Ed.) 271, Sept. 1994, S. 72-81

[Hei95] M. Heisel, S. Jähnichen, M. Simons, M. Weber: Einbettung mathematischer Techniken in den Systementwurf. GI-Softwaretechnik-Trends, Mitteilungen der Gesellschaft für Informatik, Fachgruppen „Software-Engineering“ und „Requirements-Engineering“, Band 15, Heft 3, Okt. 1995, S. 99

[Jac99] I. Jacobson, G. Booch, J.Rumbaugh: The Unified Software Development Process, Addison Wesley Longman, 1999

[Lig98] P. Liggesmeyer, M. Rothfelder, M. Rettelbach, T. Ackermann: Qualitätssicherung Software-basierter technischer Systeme – Problembereiche und Lösungsansätze, In: Informatik Spektrum, Organ der GI, Heft 5, Okt. 1998

[Maf96] O. Maffeïs, A. Poigné: Synchronous Automata for Reactive, Real-Time or Embedded Systems, Arbeitspapier der GMD no.967, GMD SET-EES, Jan. 1996, S. 3ff.

[Petr98] R. Petrasch: Einführung in das Software-Qualitätsmanagement. Logos Verlag Berlin, 1998, S. 26-30

[Rob98] J. Robra: Qualitätsrisiken durch UML. In: CONQUEST ’98, Tagungsband zur „Conference on Quality Engineering in Software Technology“ des Arbeitskreises Software-Qualität Franken e.V. (ASQF), 28./29. Sept. 1998, S. 114-151

[Sel98] B. Selic, J. Rumbaugh: Die Verwendung der UML für die Modellierung komplexer Echtzeitsysteme, In: OBJECTspektrum Nr. 4, Juli/Aug.1998, S. 24-36

Refining analysis patterns into correct control software

Günter Graw
Universität Dortmund
graw@ls4.cs.uni-dortmund.de

Introduction

Currently object-oriented software (e.g. embedded Java, Real-Time Java) has reached the state to be applied in technical applications of real life like controllers or hand held telephones. There are even extensive efforts to use object-oriented software in applications having hard real-time requirements [N99].
Furthermore there are some new approaches for the usage of object-oriented modelling techniques based on the UML in the analysis and design of real-time software. These are the Real-Time UML[D98] and the UML-RT [ObjecTime].
The analysis, design and implementation of object-oriented software has strongly been influenced by the idea of patterns in the last years. On the one hand there are some patterns for the design of real-time software like those given in [D98].
On the other hand analysis patterns have to be defined for every domain on their own applying techniques for domain engineering. Analysis patterns have been applied by Fowler [F99] who has found and applied them in several industrial projects. In [RZ96] a taxonomy of patterns is presented. Riehle uses the term conceptual pattern to define a similar concept to analysis patterns:
A conceptual pattern is a pattern whose form is described of the terms and concepts from an application domain. Therefore conceptual patterns help to understand an application domain and guide the perception of an application domain. Riehle demands that conceptual patterns should have interrelations within a restricted application domain.
In this paper we concentrate on the domain of control software for chemical plants which has real-time requirements. A taxonomy for this domain has been given by Epple in [E92] and [P94]. Moreover we want to continue this work searching typical analysis patterns for this domain which should be collected in a pattern catalog. Thus we modelled these analysis patterns in UML by the aid of the stated taxonomy.
We currently apply these analysis patterns for the modelling as well as the design of a control software for a chemical plant which is used for demonstration purposes at the University of Dortmund. In the following step we want to address the problem of refining analysis patterns into correct designs concerning structural as well as behavioural aspects. Furthermore it is remarkable to think about how real-time requirements can be specified and refined using patterns.

Analysis Patterns for control software

Using the taxonomy from [EKS92], [E93] and [P94] we have found several analysis patterns for the control software in plants. Examples of these are:

- the discrete control device/group control pattern

- the group control/plant section control pattern
- the plant section/warning/image pattern
- the group control/locking pattern
- the functional unit/state pattern

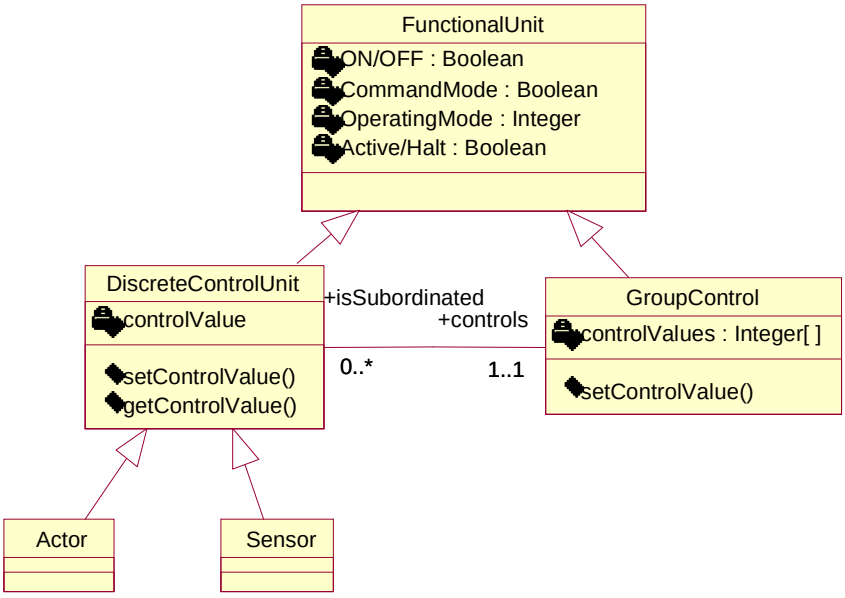


Figure 1:Class diagram of discrete control device/group control pattern

We have collected these patterns in a pattern catalog [G99].
Here only the discrete control device/group control pattern is presented in detail to explain the principle. The class diagram in Figure 1 and the collaboration diagram in Figure 2 depict the structural and dynamical descriptions of this pattern.
The pattern is applied in the context of plant control. It represents a solution to the problem of modelling the structure of control software on the group level. The abstract discrete control device class as well as the group control class are subclasses of the abstract Functional Unit class. This class models the several states of a unit of control software. The attribute *ON/OFF* models whether the unit is active or inactive. Furthermore the attribute *CommandMode* specifies who is responsible for setting the control values of the unit. This may be a plant operator (manual) or a higher level technical unit (automatic mode). The attribute *OperatingMode* describes the different behaviours a functional unit on the group level may have. It is possible that a function unit can carry out distinct (selectable) tasks. Finally the *active/halt* attribute models finer steps of a behavioural strategy. Because a group control has to communicate with its control devices and vice versa there is an association between the group control and the discrete control device. It is possible to set a control value in control device via the *setControl* Operation. The sense of this variable is to provide a time-varying setpoint of a controlled process property. Furthermore a group control is also equipped with control values. The *setControlValue* operation is responsible for setting control values in control devices. Finally the classes sensor control and actor control are concrete subclasses of the discrete control device class.

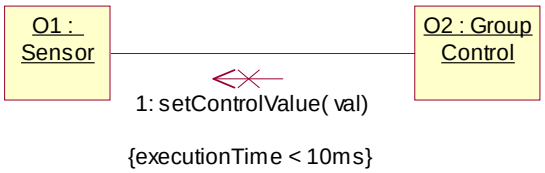


Figure 2:Refined collaboration for the example pattern

In Figure 2 the collaboration diagram for the pattern is shown. The synchronous message between the group control O2 and the sensor O1 to set a control value should require less than 10ms which is expressed by a timing constraint.

Refinement of analysis patterns

Analysis patterns provide small models of software on a very abstract level. To obtain a design it is necessary to refine them. The following possibilities exist to refine analysis patterns:

- The combination of classes from analysis patterns with design patterns.
- The replacement of classes from analysis patterns by real-time patterns.
- The extension of classes from analysis patterns by additional behaviour (attributes, operations).
- Of course combinations of the former three variants are permitted.

For the combination of classes from analysis patterns several design patterns exist. Well-known pattern catalogs include behavioural patterns (e.g. state, strategy..) and structural ones (e.g. facade, flyweight...). Furthermore in [D98] patterns which are especially interesting in the context of real-time computing are presented. These also deal with fault-tolerance and distribution(transaction, synchronisation) aspects of software systems. E. g. the watchdog pattern controls a system on a periodic or sequence-keyed basis. If a service occurs to late some action to correct this delay are taken.
Depending on the target system classes of active objects of analysis patterns have to be replaced by so called task objects. For example Newmonics [N99] offers with Real-Time Java a virtual machine which is able to fulfil real-time requirements. In the according API every active object has to be an object of a task class. Several tasks may be combined in a real-time activity. Thus every active object is represented by a task object. There are three basic types of task classes:

- The periodic task. These tasks are invoked at a regular time within a fixed period of execution. E.g. group controls become periodic tasks in this execution model. Furthermore some kinds of sensors are periodic tasks.
- A sporadic task is typically triggered in response to particular external events. Actuators normally are sporadic tasks.
- The spontaneous task. These are similar to sporadic tasks in that they are triggered at unpredictable times, in response to events or conditions detected in the external environment. Spontaneous tasks may be rejected by the real-time executive depending on the workload.

Finally it is possible to extend classes of analysis patterns by adding new attributes or operations which are important for the design of control software.
Of course a correct design may consist of different kinds of combinations of all three variants. In the refined collaboration diagram of Figure 3 O5 is an instance of a watchdog pattern that requests the group control in certain intervals (every 50ms). Furthermore the different operating modes of a group control are compressed within a strategy pattern which is represented by O4. A proxy is also added as O3 which is the reason that we have two *setControlValue* messages the first one should take less than 2ms while the second one should last less than 8ms. The sensor object O1 becomes a sporadic task while the group control object O2 is represented by a periodic task.

Correct refinement

To support the design of correct software we provide for two models on different levels of abstraction the abstract software model (ASM) and the concrete software model (CSM).

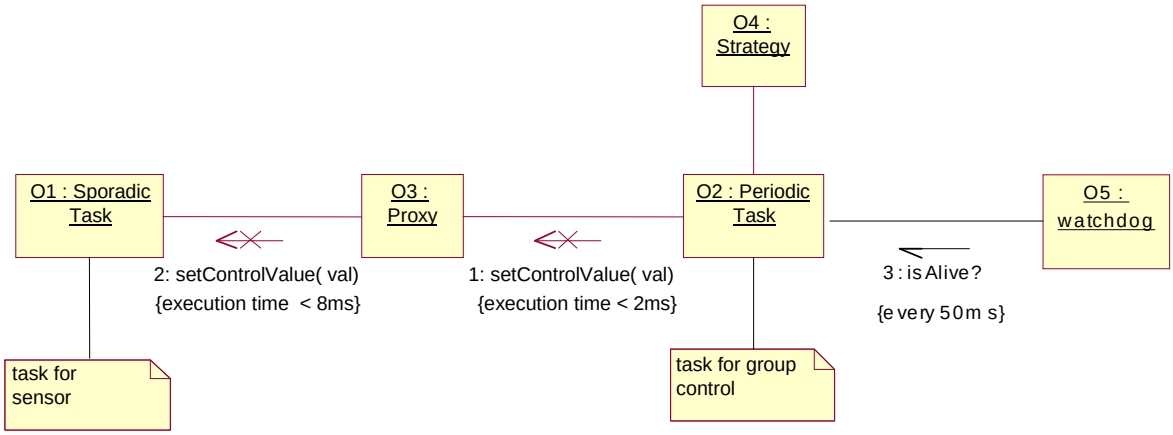


Figure 3: Collaboration diagram for the pattern

The ASM serves as interface between application engineering and software development and defines a starting point. It models the structuring of the software parts of the system into logical components. Abstract real-time and functional requirements are captured in the requirements engineering and are represented in the ASM.
The CSM is a refinement of the ASM and the destination of the development process. It explicitly refers to distribution and network communication, to fault-tolerance mechanisms and performance optimisation as well as to the allocation and management of resources. So, it structures the software into implementation-oriented components (e.g., application processes and communication stubs, back-up and watchdog processes).
It is important to note that patterns are used to build the ASM and CSM. The ASM is composed of analysis patterns while the CSM is composed of design patterns, respectively patterns that deal with implementation oriented aspects like distribution or fault-tolerance. Correctness is established as a refinement relation between ASM and CSM patterns.
To perform correctness proofs CSM and ASM patterns have to be formalised. We use the specification language cTLA supporting the constrained-based formal modelling of UML diagrams [GHK99]. Moreover, cTLA supports formal correctness proofs of refinements (CSM) with respect to more abstract specifications (ASM). Both, the abstract specification and the description of a refinement are expressed by cTLA-formulas. The refinement is correct, if the implication from the CSM to the ASM can be deduced in the TLA calculus (linear time temporal logic).

Conclusion and open questions

The following results should be contributed to the workshop:

- Analysis patterns are an appropriate means to capture functional and non-functional requirements.
- Analysis patterns for control software represent small reusable units for the OOA which can be refined into software designs. Furthermore they represent good starting points for iterative refinement processes into designs for control software.
- The three possible ways of refinement are extension, combination and replacement as well as their combination.

Refinements are graphically denoted using the <<*refines*>> stereotype. In the UML refinements are performed between collaborations which realise use cases. This means that the according use cases are refined. A collaboration on a more abstract level may be refined by some more detailed collaborations. There are still open questions:

- How are the more detailed collaborations which belong to different use cases composed ? E.g. Catalysis [DW98] offers the possibility to compose collaborations.
- Which ways do exist to show that refinements between collaborations on different levels of abstraction are correct ? There are of course informal ways that are sometimes reasonable in practice. Which benefits could formal methods have ? Currently several refinement calculi exist which allow to prove the correctness of a refinement, but it is still open how they can be applied to achieve this goal.
- And finally which ways do exist to show that the refinement of timing constraints is correct ? Of course it is possible to simply add up the results of time expressions for all the operations in an interaction(there may be couples of threads), but this seems not to be suitable for the design of big software systems.

References

[BJR99] Grady Booch, Ivar Jacobson, James Rumbaugh, The Unified Modeling Language User Guide (The Addison-Wesley Object Technology Series), 1998.

[D98] Bruce Powel Douglass, Real-Time Uml : Developing Efficient Objects for Embedded Systems (Addison-Wesley Object Technology Series), 1998.

[DW98] Desmond Francis D' Souza, Alan Cameron Wills: Objects, Components, and Frameworks with Uml:The Catalysis Approach, *Addison-Wesley Object Technology Series*, 1998.

[E93] Ulrich Epple, Die Bausteintechnik - Grundlage einer objektorientierten Netzstruktur für die Prozeßleittechnik, GMA-Fachbericht 4, (1993). Page 91-100.

[EKS92] Ulrich Epple, Hellmut Kopec, Rüdiger Schmidt, Strukturierung von Prozeßführungsaufgaben und Leitsystemsoftware, Automatisierunstechnische Praxis, 1992.

[G99] Günter Graw, A pattern catalog for the analysis and design of software for control systems, Forschungsbericht Nr. 706/1999, Fachbereich Informatik, Universität Dortmund

[GHK99] G. Graw, P. Herrmann, H. Krumm: Constraint-Oriented Formal Modelling of OO-Systems. To appear in: Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS 99), Helsinki, June/July 1999. Kluwer Academic Publisher.

[N99] http://www.newmonics.com

[F99] Martin Fowler, Analysemuster, Addison-Wesley, 1999.

[ObjecTime] http://www.objectime.com/otl/technical/index.html

[P94] Martin Polke, Prozeßleitechnik, Oldenbourg, 1994.

[RZ96] Dirk Riehle, Heinz Züllighoven, Understanding and Using Patterns in Software Development, Practice of Object Systems 2, 1 (1996). Page 3-13.

OO-Methoden für harte Echtzeitsysteme

Laila Kabous Wolfgang Nebel
Univeristät Oldenburg
Postfach 2503
26111 Oldenburg
email: Laila.Kabous@informatik.uni-oldenburg.de
http://babbage.informatik.uni-oldenburg.de

1 Einleitung

Als Echtzeitsysteme werden i.A. Elektroniksysteme bezeichnet, die in größere Umgebungen integriert sind. Beispiele solcher Systeme sind Controller, Telekommunikationssysteme, sowie Systeme aus den Bereich der Automobilindustrie und der Robotik. Es werden zwei Hauptklassen von Echtzeitsystemen unterschieden: harte und weiche Echtzeitsysteme. Harte Echtzeitsysteme unterscheiden sich von weichen Echtzeitsystemen dadurch, daß bei ersteren Systemen die spezifizierten Deadlines eingehalten werden müssen. Das Nicht-Einhalten einer Deadline würde bei einem harten Echtzeitsystem zu einem fehlerhaften Verhalten des Systems führen.
Dies könnte in einigen Fällen sogar die Gefährdung von Menschenleben zur Folge haben. Die Anforderungen an ein Echtzeitsystem sind häufig komplexer als bei konventionellen Systemen und müssen dementsprechend präziser beschrieben werden. Sie beinhalten zusätzlich zu den funktionalen Anforderungen sogenannte nicht-funktionale Anforderungen wie Zeitanforderungen, die Spezifikation von Unterbrechungen oder die Vergabe von Prioritäten. Die wichtigste Phase in der Entwicklung eines Echtzeitsystems ist die Generierung eines konsistenten Designs, das alle Spezifikationsanforderungen erfüllt.
Die in Forschung und Praxis relevanten Analysemethoden verwenden zwei unterschiedliche Paradigmen. Diese lassen sich in zwei Methodengruppen klassifizieren: die Gruppe der strukturierten Analysemethoden und die Gruppe der objektorientierten Analysemethoden.
In der Praxis sind die strukturierten Analysemethoden [6] etabliert und weit verbreitet, während die objektorientierten Analysenmethoden von re-

lativ wenigen Softwareentwicklern eingesetzt werden.
In der Forschung ist die Entwicklung strukturierter Methoden weitgehend abgeschlossen. Aufgrund der zunehmenden Verbreitung objektorientierter Programmiersprachen gilt das Interesse zunehmend den objektorientierten Analysemethoden. Heute sind sie in den meisten Anwendungsbereichen bereits nutzbringend einsetzbar. In einigen speziellen Anwendungsbereichen können sie durch geeignete Erweiterungen noch wesentlich effizienter werden.

2 Objektorientierte Analysemethoden

Mittlerweile sind über 40 Analysemethoden in der OO-Welt bekannt (OOA [5], OMT [10], HOOD [8], HRT-HOOD [1], UML [11], ROOM [14] etc.). Mehrere Methoden wurden im Bezug auf deren Anwendbarkeit und deren Eignung zur Modellierung harter Echtzeitsysteme systematisch und detailliert verglichen [9]. Dieser Vergleich kommt zu folgenden Schlußfolgerungen:

1. das dynamische Verhalten der Systemmodelle wird nicht hinreichend spezifiziert.
2. in den Modellen fehlen integrierte aussagekräftige Beschreibungen des zeitlichen Verhaltens der einzelnen Objekte.
3. die meisten Methoden unterstützen keine Simulation auf Basis der Modelle.
4. bei den objektorientierten Systemmodellen resultieren Probleme aus der Kombination von Vererbung und Synchronisation, die von allen bekannten Methoden ignoriert werden.

3 Aktuelle Forschung

Echtzeitsysteme sind zu vielfältig, um mit einer einzigen universellen Designmethode modelliert zu werden. Ausgehend von den vershiedenen objekt orientierten Methoden ist es Ziel die Entwicklung einer objekt orientierten Methode, die insbesondere die Anforderung von harten Echtzeitsystemen unterstützt. Als Entwicklungsmethode für solchen Systemen sollte sie eine Notation und eine Entwurfsvorgehensweise beinhalten.Zusätzlich soll die Methode die folgenden Punkte mitberücksichtigen:

- Spezifikation von Zeitanforderungen *Timing Constraints.*
- ein Kommunikationsmodell, das nebenläufige Abarbeitung ermöglicht.
- Prioritätenvergabe: jedem Prozeß muß eine Priorität zugeordnet werden können
- Unterbrechung von laufenden Prozessen, um Prozesse mit einer höheren Priorität ausführen zu können *Interrupt Specification.*
- Simulation der Systemmodelle: das in der Methode spezifizierte Systemmodell sollte alle notwendigen Informationen enthalten, um das dynamische Verhalten des zu entwickelnden Systems zu simulieren.
- Implementierungunabhängigkeit hinsichtlich der Programmiersprache.

Literatur

[1] Burns, A., Wellings, A.: HRT-HOOD A Structured Design Method for Hard Real-Time Systems. Elsevier (1995)

[2] Awad, M., Kuusela, J., Ziegler, J.: Object-Oriented Technology for Real-Time Systems: A Practical Approach using OMT and Fusion. Prentice Hall (1996)

[3] Booch, G.: Object oriented design with applications. Benjamin/Cummings Publishing Company (1991)

[4] Calvez, J.P.: Embedded real-time systems A Specification and Design Methodology. Wiley (1993)

[5] Coad P, Yourdon E.: Object-Oriented Analysis. Prentice Hall Englewood Cliffs New Jersey (1991)

[6] DeMarco T.: Structured Analysis and System Specification. Yourdon Press, Prentice Hall (1979)

[7] Douglass, B.P.: Real-Time UML: Developing Efficient Objects for Embedded Systems. Addision Wesley (1997)

[8] HOOD Technical Group: HOOD Reference Manual Release 4.0. (1999)

[9] Kabous, L., Nebel, W.: Design Methodology for Hard Real-Time Systems. Available on the WWW from URL http://babbage.informatik.uni-oldenburg.de/ laila/paper/hrtdes.ps.gz

[10] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorenson, W.: Object-Oriented Modelling and Design. Prentice Hall (1991)

[11] Rational Software: Unified Modeling Language UML. Available on the WWW from URL http://www.rational.com/uml/index.shtml

[12] Selic, B., Gullekson, G., Ward, P.: Real-Time Object Oriented Modeling. Wiley (1994)

[13] Selic, B., Rumbaugh, J.: UML for Real-Time. Available on the WWW from URL http://www.objectime.com/uml/

[14] Selic, B., Gullekson, G., Ward, P.: Real-Time Object Oriented Modeling. Wiley (1994)

Making UML Work for Real Time Systems

What's Code Generation got to do with it

Georg Beier, FH Mannheim (g.beier@fh-mannheim.de)
Dirk Dinger, AOT GmbH Mannheim (ddinger@acm.org)

Motivation

The ultimate goal of building UML models for software systems is producing efficient, well documented, error free code. To do this automatically, a well defined, consistent and reproducible transformation of UML model elements to code structures is required. For some elements, like class symbols to class frames with attributes, accessor functions and operation stubs, this mapping is straight forward, at least for object oriented target languages. For many other concepts of the UML meta model, the corresponding code constructs are much less obvious. By building a code generating utility that is directly based on the UML meta model, we pursued different goals:

a) exploit UML models as far as possible for the generation of various aspects of code and other textual artifacts;

b) test UML meta model constructs for completeness and applicability and extend them using standard extensibility mechanisms, if needed.

Basic code generator technology

UML models are generated using a commercial CASE tool that supports UML notation/diagrams and offers good external access to its data dictionary. Using this access we populate a semantic meta model that itself is modeled using UML and implemented as a C++ library, using the code generator itself.

The meta model is a simplified version of the OMG meta model. It imposes several restrictions and few extensions to UML models that were found practicable in recent projects. As the meta model itself is modeled in UML and resulting code is generated to approximately 90%, changes and extensions reflecting our actual knowledge are easily incorporated. Our current focus of work is on type mapping, automatically transforming application data types (called "Fachwert" by Züllighoven et al.[1]) to context sensitive implementation types, and on defining interfaces to packages modeling different abstraction layers, and weaving them into produced code.

The actual code generator is template based. It enables a software designer to develop a system architecture by writing code templates that query the meta model. Using a WYSIWYG metaphor for the template language, code templates look as much as possible like the target language, e. g. C, C++, Ada, IDL, SQL or others. UML features that are especially important for real time systems like state machines and event communication can easily be mapped to appropriate target language constructs, operating system services and libraries. The great flexibility in code generation simplifies to insert additional code to support high level runtime tracing and verification of programs.

Transforming UML models to real time architectures

Real time systems often execute in a very specific environment. Often, restrictions like the Ravenscar Profile [2] apply. This demands high flexibility in code generation to map to the given architecture.

A typical problem is object generation and initialization. Real time control systems often run with a unchanging set of objects that are constructed during system startup and have to be set to consistent starting states. The order of object construction and initializing actions can, at least partially, be inferred from implicit UML model information like aggregation associations, event protocols and dependencies which often cannot be mapped to implementation languages. By generating this initialization code, a significant quantity of manual work can be automated and several kinds of errors can be excluded by construction.

Experiences

In a recent project, we applied the template based code generation process and graphic run time tracing to the development of control software for Siemens Simatc S7 automation systems. It proved to be relatively simple to generate SCL code from UML models. SCL on a S7 features a procedural, Pascal-like control language with numerous coding restrictions and a restricted address space. Code was generated from UML class models and state diagrams. The complete, quite intricate behavior of a control program for rain water reservoir cleaning pumps (so called whirl jets) could be generated from the models. Program behavior was verified and debugged using generated sequence diagrams. We found that “semantic debugging” with sequence diagrams was a key feature to identify and correct malfunctions. We also used transition coverage to build an exhaustive suite of test cases. Transition lists generated from the UML models could be easily matched to transition execution information from the program executing test case suites.

As a conclusion, we would like to stress the following statements:

- UML with its behavioral models is well suited do develop real time systems with high control complexity. The models should be translated to code automatically, including state models, to support development on a high abstraction level and gain high productivity.
- The system architectures of real time systems are very specific, demanding a high level of flexibility in code generation. Even environments like Siemens S7 with SCL can be supported with little effort writing code generators when using a template based code generator technology.
- Test, debugging and verification should be done on the same level of abstraction as the development. If UML models are used as primary development tools, the debugger should also use UML diagrams. Dynamically generated sequence diagrams are a first step in this direction.

[1] H. Züllighoven: Das objektorientierte Konstruktionshandbuch, Hamburg: dpunkt Verlag, 1998

[2] Proceedings of the 8th International Real-Time Ada Workshop: Tasking Profiles, ACM Ada letters, September 1997

Message Queue Concept – An Implementation Pattern for Concurrent Objects

Pierre Metz (p.metz@fbi.fh-darmstadt.de), Fachhochschule Darmstadt
John O´Brien (jobrien@cit.ie), Cork Institute of Technology
Wolfgang Weber (w.weber@fbi.fh-darmstadt.de) , Fachhochschule Darmstadt

1 Abstract

In the field of reactive and realtime systems, concurrency is an essential part of software design. The aim of adding concurrency is to improve the overall performace of the system, to build event-centered architectures and priority-controlled interaction of parallel components. In particular, concurrent software design meets future distribution requirements. Moreover, concurrent components are able to exchange synchronous or asynchronous messages explicitly.

This paper addresses concurrency aspects and synchronization requirements of object-oriented software in the field of reactive and realtime systems. It shows different ways how concurrency may be introduced and what consequences these issues have on the internal consistency of objects regarding instance and class properties as well as state semantics. It is pointed out how synchronization techniques can be applied to avoid these problems.
In this context, the Active Object Pattern by R. Greg Lavender and Douglas C. Schmidt [Vlissides 96] is a meaningful approach to model and implement concurrent software components. This paper shows how the Active Object Pattern can be further refined and improved by introducing an implementation pattern extension. These refinements focuse on full transparency concerning creation and communication of concurrent objects for the paramount benefit of reusability and extensibility. In particular, it includes queueing of received but deferred events if concurrent object are controlled by a state machine.

2 Introducing Concurrency to Object-Oriented Systems

Concurrency can be introduced to OO software in different ways. In particular, these are *threads for each scenario behind a use case*, *threads per asynchronous operation invocation* and *Object Threads*. Depending on the particular concept, special synchronization policies must be defined.

2.1 Thread for Each Scenario

The *Thread for Each Scenario* approach can be applied for systems supporting external interfaces (e.g., for graphical user interfaces, external software and hardware elements etc.).

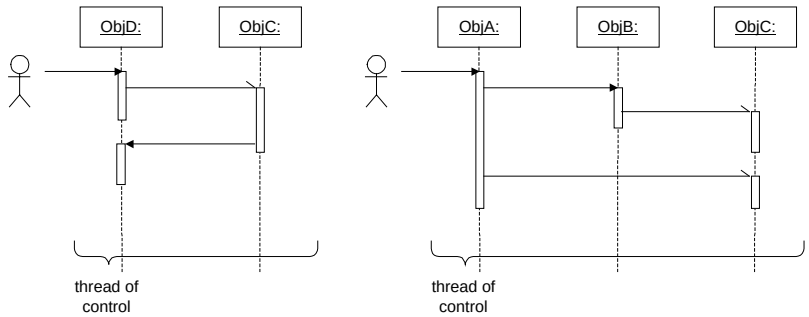


Figure 1: Thread for Each Use Case Scenario

That means an application has several threads, each of these threads is responsible for the processing of one use case (see Figure 2).

If any actor instance invokes a use case, the corresponding thread controls the scenario behind it.

2.2 Thread per Asynchronous operation Invocation

If necessary, concurrency can be increased by having more than one thread within a scenario. We can do so by creating an additional thread for asynchronous. The main thread of the scenario branches into two concurrent flows of control: Each time an object calls an asynchronous operation of another object, an additional thread is created performing the activity defined by the operation implementation of the receiver (see Figure 2). It terminates when the operation is completed.

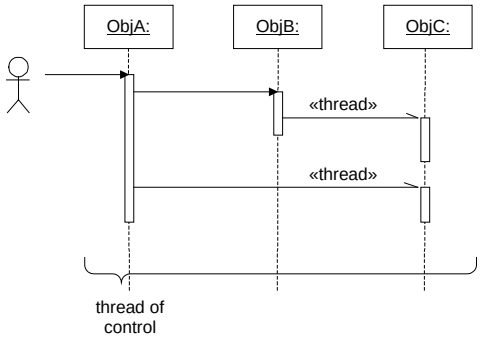


Figure 2: Thread for Asynchronous Operation Invocation

2.3 Object Threads

Another way of applying concurrency is to establish a thread for a single object or a group of objects as a logical package (Object Thread). Following this approach, we let objects exist in parallel. There is exactly one thread working on an object at any point. An object receives operation call requests and serves them (see Figure 3). Its thread is responsible to invoke operations on the object as a result of an incoming request.

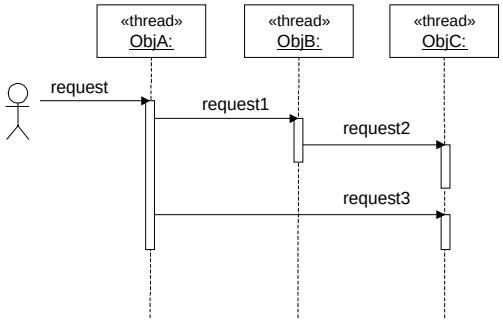


Figure 3: Object Threads

In the following section, an implementation pattern based on the Active Object Pattern by R. Greg Lavender and Douglas C. Schmidt is introduced. The Active Object pattern focuses on Object Threads as they are mostly used in concurrent systems.

3 Message Queue Concept

This paper introduces the Message Queue Concept which is considered to be an implementation pattern for the Active Object Pattern by R. Greg Lavender and Douglas Schmidt. The intention of the Active Object Pattern is to decouple the invocation of an operation from its execution in different threads of control. It is intended to be used for message dispatching instances within a concurrent

system that performs nonblocking operations (e.g., gateways). Lavender and Schmidt propose to have a message queue for objects living in their own thread. These Object Threads interchange information and communicate with each other by placing messages into each other´s message queue (see Figure 4).

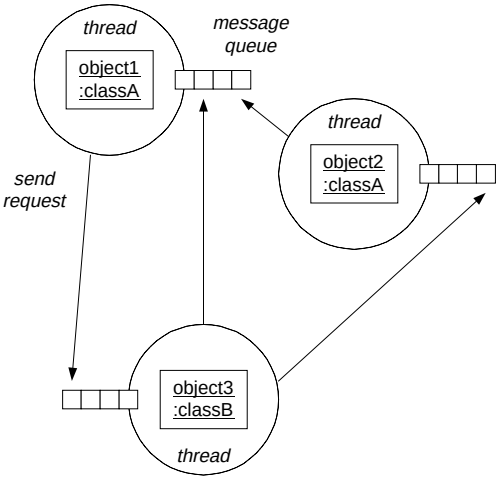


Figure 4: message queueing of concurrent objects

In order to introduce the Message Queue Concept, we consider the example of a person in Figure 5. It shows a design class *Person*. Two person instances may talk to each other. The stereotype «thread» indicates that instances of this class are considered to be concurrent.

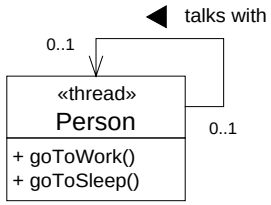


Figure 5: Example

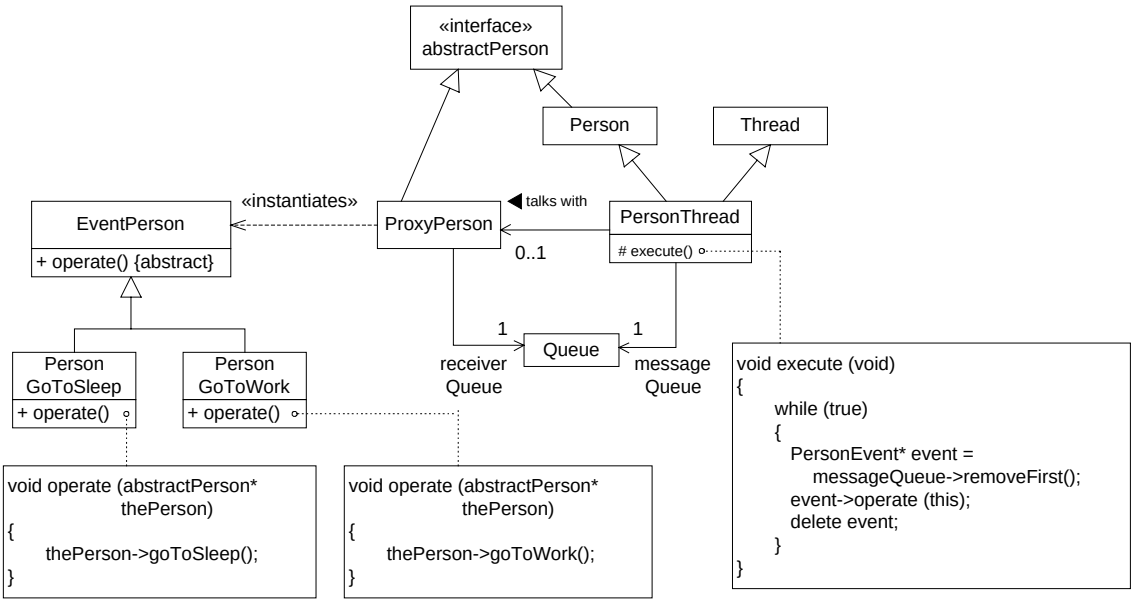


Figure 6: Class Model for the Message Queue Concept for Object Threads

A client communicates with an Object Thread by addressing its proxy. An Object Thread´s proxy transforms the operation calls into event class instances and places them into the message queue of the Object Thread.
The event classes (see Figure 6) represent visitors following the design pattern ´Visitor´ [Gamma 96]. There is an event class for each operation of class *Person´*s interface. In an endless loop, the Object Thread takes these event instances out of its queue sequentially and processes the requests (see Figure 7). A message queue acts as a shared resource following a producer/consumer synchronization policy. If empty, the Object Thread is suspended.
Following Figure 5 and Figure 6, two concurrent person instances are communicating with each other. Thus, in our example, the Object Threads are acting as a client as well as a server.

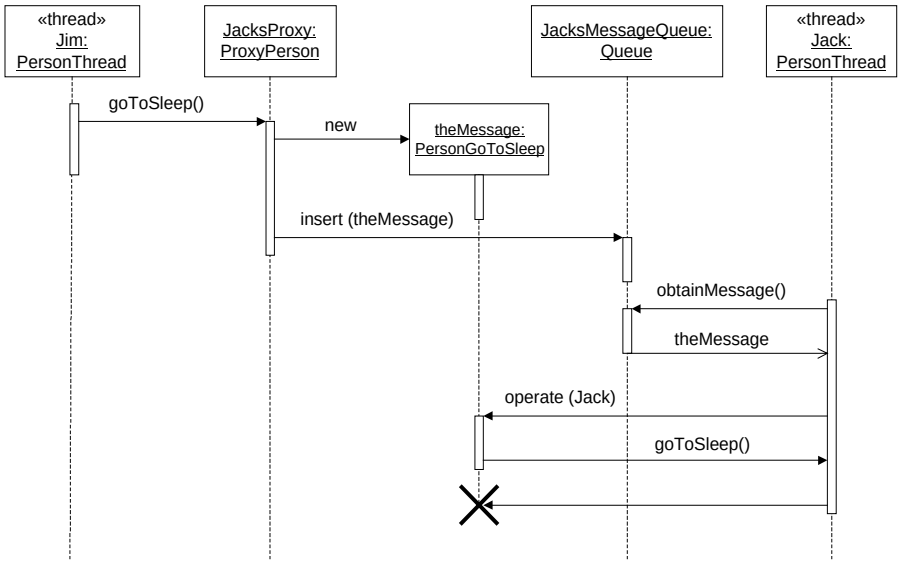


Figure 7: Sample Scenario for the Message Queue Concept

4 Implications of the Message Queue Concept

Like the Active Object Pattern, the Message Queue Concept is based on concurrent objects. It is not designed to be applied to systems with a sequential flow of control. However, it is also possible to implement message queueing in a similar way for objects in non-concurrent systems. As objects and components in event-centered architectures such as reactive and realtime systems are mostly concurrent and even distributed, a message queueing mechanism for procedural flow of control is not considered.

Full Transparency

Regarding transparency, the Message Queue Concept, rather, reveals three essential achievements which are a very important contribution to reusability and portability of code:

1. By introducing a suitable type hierarchy, the Message Queue Concept creates two versions of a domain class designated as a thread: A “raw“ class implementation for non-concurrent environments and another one including thread properties.
2. From the client´s point of view, the Message Queue Concept is completeley transparent by the use of proxies for Object Threads that encapsulate the message queue handling and thread synchronization. Message sending looks as if there is sequential operation invocation. Thus

the client code is not concerned with any low-level functionality (e.g., API calls) at all. Also, the Message Queue Concept is transparent from the perspective of the server that receives messages from the client. The server does not need to implement any message identifier decoding logic. The Message Queue Concept directly uses the design class interface of the server.

3. It is possible to transparently create concurrent objects. The creation and connecting of Object Thread, message queue instance and proxy is hidden from the client. Moreover, a client can decide dynamically whether he wants to have two objects of the same type to be concurrent or not.

Architectural Considerations

The Message Queue Concepts is a powerful and useful technique to establish transparent communication between subsystems (e.g., the Presentation-Abstraction-Control Architecture Pattern [Buschmann 96] or the Recursive Control Pattern of B.Selic [Martin 98]) and horizontal architecture layers (Multi-Tier architectures) if the requirements demand concurrency. In general, the Message Queue Concept can serve as a "communication interface" between concurrent software elements. The Message Queue Concept is independent of any high-level event-creation and -notification strategy.

Communication Semantics

The use of proxies for Object Threads makes synchronous and asynchronous communication explicit. A sender blocks if he sends a synchronous message. Further, a proxy may register with a timer if synchronous communication with timeout semantics is necessary. Furthermore, a transparent interaction logging mechanism for debugging purposes can be applied easily.

Synchronization Policies

The Message Queue Concept is attached to each object representing a thread. Because there is only one thread of control working on an object anytime, there are no concurrency conflicts. However, the Message Queue Concept does not avoid additional synchronization primitives if used not only for single objects but for logical groups of objects (composites, subsystems, architecture layers). In the case of the existence of sub-threads followed by other implementation techniques such as do-activities of statechart diagrams (do-activities are interruptable, see [UML 97]) or real asynchronous operation invocations within an thread, protection against multi-thread impact must be established additionally.

State Semantics of Objects

The UML v1.1 offers deferred event handling for statechart diagrams [UML 97]. This means to hold a particular event for later management if it cannot be served in the current state. The Message Queue Concept is flexible enough to be expanded for event queueing requirements.
The Message Queue Concept, including event queueuing, is applicable to any implementation pattern of state machines. The design of the Message Queue Concept can easily be combined with the State pattern [Gamma 96], State Table pattern [Douglass 98], the implementation strategy by Shlaer/Mellor [SM 92] or any other approach.

Inter-Process Communication

Applying the Message Queue Concept requires objects located in the same address space. The message queue strategy does not include communication between processes. Anyhow, the Message Queue Concept can be expanded for transparent inter-process communication. For instance, this can be done by the use of the Forwarder/Receiver pattern or Broker pattern [Buschmann 96].

Code Generation and Tool Support

All components necessary for the Message Queue Concept can be generated from CASE-tool repository data automatically. In this case, the Message Queue Concept appears in the programming domain, the design model is left unconcerned. Of course, code generators need to know what classes are designated as an Object Thread. The UML offers suitable notation and stereotypes (synchronous, asynchronous and timeout message semantics for Interaction Diagrams, built-in stereotypes «thread», «process» and «signal») which are to be considered by the software engineer. This requires in-depth knowledge of the UML and an agreement of how to use these modelling elements in the context of an OO method. These possibilites do not influence model checking and simulation of design models as supported by commercially available CASE-tools (e.g., *Rhapsody*)

Modelling Constraints

In the presence of Object Threads, it is important to follow the "Single Point of Reference" principle, which is an important concept of methods and techniques for building software architecture layers and decoupling software elements. In the case of concurrency, the system must be designed carefully to maintain internal object consistency regarding access to object properties and state transitions. Otherwise, inconsistencies and side effects may occur.

5 References

[Buschmann 96] F.Buschmann, R.Meunier, H.Rohnert, P.Sommerlad, M.Stal
"Pattern Oriented Software Architecture - A System of Patterns"
1996 John Wiley & Sons, Ltd.

[Douglass 98] B.P.Douglass
"Real-Time UML"
1998 Addsion-Wesley

[Gamma 96] E.Gamma, R.Helm, R.Johnson, J.Vlissides,
"Design Patterns - Elements Of Reusable Object-Oriented Software"
1996 Addison Wesley

[Martin 98] R.Martin, D.Riehle, F.Buschmann
"Pattern Languages Of Program Design 3"
1998 Addison-Wesley

[SM 92] S.Shlaer, S.J.Mellor
"Object Lifecycles - Modeling The World In States"
1992 Prentice Hall

[UML 97] G.Booch, J.Rumbaugh, I.Jacobson
"Unified Modeling Language"
UML Semantics, Version 1.1
1997 Rational Software Corporation

[Vlissides 96] J.Vlissides, J.Coplien, N.L.Kerth
"Pattern Languages Of Program Design 2"
1996 Addsion-Wesley

Eine vergleichende Fallstudie mit CASE-Werkzeugen für objektorientierte und funktionale Modellierungstechniken

Erik Kamsties[1], Antje von Knethen[2], Jan Philipps[3], Bernhard Schätz[3]

[1]Fraunhofer Einrichtung für Experimentelles Software Engineering,
Sauerwiesen 6, D-67661 Kaiserslautern
[2]Fachbereich Informatik, Universität Kaiserslautern,
D-67653 Kaiserslautern
[3]Fakultät für Informatik, Technische Universität München,
Arcisstraße 21, D-80290 München

- Positionspapier -

1 Einleitung

Software wird zunehmend in Produkten des Alltags wie Videorecordern und Waschmaschinen eingesetzt, aber auch im Automobilbereich, in der Luft- und Raumfahrt, in der Telekommunikation und in Medizintechnik. Mehr und mehr Funktionalität solcher Systeme wird nicht mehr durch Hardware, sondern durch Software realisiert.
Objektorientierte und funktionale Modellierungstechniken ermöglichen es, Systeme präzise zu beschreiben und helfen somit Mehrdeutigkeiten der natürlichen Sprache einzuschränken. Objektorientierte Modellierungstechniken wie ROOM oder UML/RT lösen dabei langsam einige funktionale Modellierungstechniken wie SA/RT ab. Für den industriellen Einsatz von Modellierungstechniken sind CASE-Werkzeuge unerläßlich. Das herausragende Merkmal von CASE-Werkzeugen ist die Unterstützung von Aktivitäten, die ohne Werkzeug nicht in dem Umfang möglich wären, z.B. Konsistenzüberprüfung, oder Simulation. Außerdem bieten diese Werkzeuge oft partielle oder vollständige Kodegenerierung, was Entwicklungszyklen verkürzt, und unterstützen die Verwaltung von Spezifikationen.
Ein breites Spektrum von Modellierungstechniken steht dem Entwickler zur Verfügung. Dieses Positionspapier beschreibt eine Fallstudie mit acht Modellierungstechniken und den dazugehörigen CASE-Werkzeugen. Konkret wurden als objektorientierte Modellierungstechniken OCTOPUS, ROOM, SDL und UML, sowie als funktionale Modellierungstechniken Focus [BDD+92], Petrinetze, SCR und Statecharts gewählt. Es wurden die folgenden CASE-Werkzeuge benutzt: Software through Pictures/OMT (Aonix), ObjecTime Developer (ObjecTime Systems), ObjectGEODE (Verilog), Rhapsody (I-Logix), AutoFocus [HMR+98], PEP [Gra97], SCR* [HBGL95], und Statemate MAGNUM (I-Logix).
Ziel der Fallstudie war es, den Grad der Unterstützung von Aktivitäten der Anforderungsspezifikationsphase vergleichend[1] zu bewerten. Die beiden betrachten Aktivitäten waren die Erstellung einer Spezifikation und deren Überprüfung; die Anforderungen selbst waren bereits vorgegeben. Außerdem wurde auch noch der Aufwand für die Einarbeitung in die Modellierungstechnik und das entsprechende Werkzeug bewertet.
Die Fallstudie wurde im Rahmen eines Seminars durchgeführt, daß zeitgleich an der Universität Kaiserslautern und der Technischen Universität München stattfand. Insgesamt nahmen 22 Studenten an der Fallstudie teil, die über 13 Wochen lief. Als Spezifikationsaufgabe wurden textu-

1. Vergleichend heißt, daß keine absolute quantitative Bewertung (z.B. “sehr gut” bis “mangelhaft”), sondern eine Bewertung der Modellierungstechniken und Werkzeuge relativ zueinander in qualitativer Weise erfolgen sollte.

elle Anforderungen an das Tamagotchi vorgegeben (das Tamagotchi ist ein eingebettetes System, daß ein virtuelles Lebewesen simuliert).

2 Aufbau der Fallstudie

In der Fallstudie wurden die folgenden Fragestellungen konkret untersucht:

- Einarbeitung
  - Wie leicht kann sich ein Entwickler in die Modellierungstechnik und das Werkzeug einarbeiten? Hierzu wurde der Aufwand und die Qualität der Dokumentation bewertet.
- Erstellung einer Spezifikation
  - Wie gut wird Teamarbeit durch Modellierungstechnik und Werkzeug unterstützt?
  - Wieviele und welche Arten von Mängeln in informellen Anforderungen werden durch die Erstellung einer Spezifikation aufgedeckt?
  - Wie groß ist die entstandene Spezifikation?
  - Wie hoch ist der Aufwand und was sind Probleme bei der Erstellung einer Spezifikation?
  - Wie ist die Qualität der entstandenen Spezifikation?
- Überprüfung einer Spezifikation
  - Wie umfassend ist die automatische Überprüfung einer Spezifikation?
  - Wie ausführlich ist die Simulation bzw. wie umfangreich ist die Kodegenerierung?
  - Wie umfassend ist die Generierung von Dokumentation für Reviews?

Weitere interessante Fragestellungen würden sich im Zusammenhang mit der weiteren Verwendung der Spezifikationen für Entwurf, Implementierung und als Basis für Test, Verifikation und Wartung ergeben. Diese Aktivitäten lagen aber außerhalb des zeitlichen Rahmens der Fallstudie. Der Aspekt der Benutzerfreundlichkeit (Useability) der Werkzeuge wurde bewußt ausgelassen, da dies schwer ist auch nur annähernd objektiv zu bewerten. Diese Studie soll existierende Vergleiche von Werkzeug auf Basis von Fragenkatalogen zu Leistungsmerkmalen (“welche Betriebssysteme werden unterstützt?”) durch empirische Daten komplettieren.
Die Fallstudie lief in drei Schritten ab: Einarbeitung, Spezifikation, Review und Überarbeitung.

1. Einarbeitung in Modellierungstechnik und Werkzeug (4 Wochen)
   Die Studenten erhielten eine kurze Anforderungsbeschreibung einer Notabschaltung eines Reaktors als Trainingsbeispiel und Literatur zur Technik und Handbücher zum Werkzeug. Die Aufgabe bestand darin, die Literatur durchzuarbeiten und eine Spezifikation des Trainingsbeispiels mit Hilfe des Werkzeugs zu erstellen.
2. Spezifikation und Simulation des Tamagotchis (5 Wochen)
   Die Studenten erhielten das Anforderungsdokument für das Tamagotchi. Zunächst wurden die Anforderungen unter den Teammitgliedern aufgeteilt und ein Arbeitsplan für die Erstellung der Spezifikation aufgestellt. Beides wurde dann mit dem Betreuer abgestimmt. Anschließend wurde die Spezifikation geschrieben. Schritthaltend mit dem Spezifizieren wurde auch simuliert, sofern das Werkzeug dies zulies, um frühzeitig Fehler zu eliminieren.
3. Review und Rework (3 Wochen)
   Die Teams tauschten nach der vorläufigen Fertigstellung der Spezifikation diese paarweise untereinander aus. Die Spezifikation wurden zunächst inspiziert und anschließend in einem Meeting diskutiert. Außerdem wurden im Meeting die Spezifikationen auf Basis von Fragen stichprobenartig simuliert, die sich in der Inspektion ergeben haben.

3 Resultate

Im folgenden sind die Ergebnisse der Fallstudie auszugsweise dargestellt.

Einarbeitung. Der Aufwand für die Einarbeitungsphase betrug zwischen 20 und 70 Stunden. Tabelle 1 zeigt die Aufwandsverteilung über die verschiedenen Modellierungstechniken. Im

Aufwand enthalten ist das Lesen der Literatur und Manuals, sowie die Erstellung der Trainingsspezifikation. Der Aufwand ist über die Teammitglieder summiert.

Focus AutoFocus	Petrinetze PEP	Octopus StP/omt	ROOM ObjecTime	SCR SCR*	SDL ObjectGEODE	Statecharts Statemate	UML Rhapsody
20	70	60	50	40	70	30	40

Tabelle 1: Aufwand für Einarbeitungsphase (Stunden)

Die Aufwandsdifferenzen entstanden im wesentlichen durch den Umfang der Literatur bzw. Handbücher, die bei den kommerziellen Werkzeugen natürlich umfangreicher ausfallen als bei Forschungsprototypen. Der Einarbeitungsaufwand des Statemate- und des AutoFocus-Teams ist deswegen so gering, weil zur Modellierung der Reaktornotabschaltung die Kenntnis von nur wenigen Konstrukten (z.B. State-charts) ausreichend war. Die komplette Statemate-Vorgehensweise mit Activity- und Module-charts wurde erst in der Spezifikationsphase gelesen und verstanden; ähnlich verhielt es sich bei AutoFocus. Der niedrige Aufwand für Rhapsody erklärt sich durch Vorkenntnisse eines Teammitglieds (hatte an Rhapsody-Schulung bei Berner&Mattner teilgenommen). Die Qualität der Dokumentation ist erwartungsgemäß bei den kommerziellen Werkzeugen sehr gut. Die Dokumentation zum Statemate-Ansatz beispielsweise ist zwar umfangreich (ca. 300 Seiten), ließ aber keine Fragen offen.

Größe der Spezifikation. Die Seitenanzahl der ausgedruckten Spezifikation gibt nur eine grobe Annäherung, da durch das automatisches Layout der Informationsgehalt von Seite zu Seite stark schwankt. Die Tabelle 2 zeigt die Größe der einzelnen Spezifikationen. ObjecTime und Rhapsody erfordern die Beschreibung bestimmter Aspekte direkt in C++ Kode, daher ist hier die Seitenanzahl insgesamt und die des graphischen Teils (in Klammern gesetzt) angegeben.

Focus AutoFocus	Petrinetze PEP	Octopus StP/omt	ROOM ObjecTime	SCR SCR*	SDL ObjectGEODE	Statecharts Statemate	UML Rhapsody
30	20	20	84 (33)	36	136	40	? (20)

Tabelle 2: Größe der Spezifikationen (Seitenanzahl)

Die Größe der Spezifikationen ist von der Beherrschung der Modellierungstechnik und ihrer Darstellungseffizienz abhängig. Die Größe (und damit die Darstellungseffizienz) der Spezifikationen in Focus, Petrinetzen, Octopus, SCR, und Statecharts ist relativ ähnlich. Die ROOM und UML-Spezifikationen sind etwas größer, da hier ein Teil graphisch (Seitenanzahl in Klammern gesetzt), und der andere Teil textuell als C++ Kode beschrieben wird. Die Größe der SDL Spezifikation ist relativ hoch, was zum Teil darauf zurückzuführen ist, daß Fallunterscheidungen, in SDL ausgedrückt, viel Platz benötigen und das automatische Seitenlayout nicht optimal ist.

Teamarbeit. Das SCR* Werkzeug sieht Teamarbeit nicht vor, es erlaubt weder die Aufteilung der Spezifikation in unabhängig bearbeitbare Teile noch die Zusammenführung getrennt entwickelter Spezifikationen. Der SCR Notation fehlt ein Konzept zur Kapselung von Spezifikationsteilen, was bei der Teamarbeit hilfreich wäre. Statemate unterstützt Teamarbeit vollständig, soweit dies im Rahmen der Fallstudie bewertbar ist; die Spezifikation konnte arbeitsteilig entwickelt und simuliert werden. AutoFocus unterstützt durch die Verwendung eines Versionsverwaltungssystems mit Mehrbenutzerprinzip auf der einen Seite sowie der modularen Modellierungstechniken auf der anderen Seite die Entwicklung im Team im wesentlichen sehr gut. Vier der acht Teams konnten aus technischen Gründen (Lizenzen, o.ä.) nicht arbeitsteilig

arbeiten (ObjecTime, ObjectGEODE, Rhapsody und StP/OMT).

Aufwand. Der Aufwand für die Spezikationsphase betrug zwischen 100 und 230 Stunden. Tabelle 3 zeigt die Aufwandsverteilung über die verschiedenen Modellierungstechniken. Im Aufwand enthalten ist nur die Erstellung der Tamagotchi-Spezifikation mit dem Werkzeug. Der Aufwand ist über die Teammitglieder summiert.

Focus AutoFocus	Petrinetze PEP	Octopus StP/omt	ROOM ObjecTime	SCR SCR*	SDL ObjectGEODE	Statecharts Statemate	UML Rhapsody
140	110	100*	110*	100	210 *	100	230 *

Tabelle 3: Aufwand für Spezifikationsphase (Stunden, *=keine Arbeitsteilung)

Die Aufwände für Focus, Petrinetzen, SCR, und Statecharts können nur indirekt mit den Aufwänden von OCTOPUS, ROOM, SDL und UML verglichen werden. Bei den Teams, die die Spezifikation aufgeteilt haben, ist zusätzlicher Koordinierungsaufwand entstanden. Bei den Teams, die die Spezifikation gemeinsam erstellt haben, hätte im Grunde ein Student die Spezifikation auch alleine erstellen können, d.h. der Aufwand würde dann nur ein Drittel bzw. die Hälfte (je nach Teamgröße) betragen. Teilt man beispielsweise den Aufwand für UML durch die Teamgröße ergibt sich ein Aufwand von 80 Stunden; zieht man bei Statemate einen gewissen Koordinierungsaufwand ab, kommt man auf einen sehr ähnlichen Wert. Fazit ist, daß keine signifikanten Unterschiede im Aufwand zwischen den verschiedenen Modellierungstechniken und Werkzeugen zu verzeichnen sind, mit Ausnahme von ROOM. Die ROOM-Spezifikation hätte arbeitsteilig einen Aufwand von ca. 35 Stunden + Koordinierungsaufwand benötigt, was daran liegt, daß große Teile der Funktionalität des Tamagotchi durch C++ - Kodefragemente in den Transitionen realisiert sind; im Umgang mit CASE-Werkzeugen unerfahrenen Studenten fällt dies vermutlich leichter als alle Möglichkeiten der eigentlichen Modellierungssprachen auszuschöpfen.

4 Zusammenfassung

Wir haben acht Modellierungstechniken und zugehörige CASE-Werkzeuge im Rahmen einer Fallstudie miteinander hinsichtlich Einarbeitungsaufwand und Unterstützung bei der Erstellung und Überprüfung einer Spezifikation verglichen. Dabei haben wir Erfahrungen auf mehreren Ebenen gemacht:

- methodisches Vorgehen bei der Anwendung der einzelnen Modellierungstechniken und Werkzeuge,
- Effektivität und Effizienz der Modellierungstechniken im Vergleich
- Durchführung von Fallstudien zur Bewertung von Modellierungstechniken

Die Erfahrungen mit dem methodischen Vorgehen, die wir mit den einzelnen Modellierungstechniken gesammelt haben, ließen sich natürlich nicht im Rahmen dieses Positionspapiers darstellen. Für die Studenten bestand das Problem, daß die verwendeten Modellierungstechniken in der Regel keine oder nur schwache methodischen Hinweise lieferten. Für den Experten ist dies in kein Problem, da sich Erfahrungen, die mit anderen Modellierungstechniken gewonnen wurden, oft übertragen lassen. Für uns als Betreuer war daher die Schwierigkeit am Anfang der Fallstudie, daß *implizite* (d.h. verinnerlichte) Wissen an die Studenten weiterzugeben, da vieles scheinbar so selbstverständlich ist, daß man es nicht mehr erwähnt [DD86]. Durch die Beobachtung der Probleme, die die Studenten mit der Modellierungstechnik und dem Werkzeug hatten, und der Bewertung der entstandenen Spezifikation, haben wir *explizites* methodisches Wissen gewonnen, von dem jetzt weitere Studenten profitieren können.
Zwischen den Modellierungstechniken und CASE-Werkzeugen konnten wir nur leichte Unter-

schiede feststellen. Petrinetze und SCR wurden subjektiv als leicht erlernbar empfunden, aber der Einarbeitungsaufwand für die Werkzeuge hat diesen leichten Vorteil wieder relativiert.
Unterschiede konnten auch in der Unterstützung von Teamarbeit auf Seiten der Modellierungstechnik wie auch der CASE-Werkzeuge festgestellt werden. Die kommerziellen Werkzeuge - wie z.B. Statemate- schnitten in diesem Punkt natürlich deutlich besser ab. Focus/AutoFocus und Statemate unterstützten Teamarbeit gut, bei SCR/SCR* und Petrinetzen/PEP fehlten Sprachkonstrukte, sowie eine adäquate Werkzeugunterstützung dieses Aspekts.
Die deutlichsten Unterschiede zeigten sich in der Länge der entstandenen Spezifikationen, die zwischen 20 und 100 Seiten variierte - bei vergleichbarer Präzision und Vollständigkeit. Im Erstellungsaufwand schnitt ROOM etwas besser als die restlichen Techniken ab, was darauf zurückzuführen ist, daß größere Teile der Spezifikation direkt in C++ beschrieben wurden.
Die Qualität der entstanden Spezifikation ist was Konsistenz und interne Vollständigkeit angeht hoch, was durch die Konsistenzprüfungen der Werkzeuge ermöglicht wurde. Die Simulatoren haben außerdem geholfen, die Semantik der Modellierungstechnik besser zu verstehen und somit Fehler zu vermeiden. Die Simulationsfähigkeit einer Spezifikation ist darüber eine wichtige vertrauensbildende Eigenschaft, wenn der Leser, d.h. in der Realität der Kunde/Auftraggeber, mit der Modellierungstechnik nicht völlig vertraut ist. Durchgängiges Problem bei allen Spezifikationen ist, daß sie zu kompliziert sind; der gleiche Informationsgehalt könnte mit der gleichen Technik einfacher modelliert werden. Der Erzeugung leicht reviewbarer Dokumentation sollte bei der Weiterentwicklung von Werkzeugen mehr Aufmerksamkeit geschenkt werden.
Diese Ergebnisse beinhalten keine Wertung. Ob es z.B. ein Nachteil ist, daß Spezifikationen mit einer Modellierungstechnik möglicherweise größer ausfallen, als mit einer anderen, muß in Bezug zum jeweiligen Anwendungskontext der Modellierungstechnik interpretiert werden.
Durch den Vergleich mit der ersten Fallstudie zu Modellierungstechniken, die mit dem Tamagotchi jedoch ohne CASE-Werkzeuge durchgeführt wurde [KKR+98], können wir Aussagen zum Effekt der Einführung von CASE-Werkzeugen ableiten. In dieser Vorgängerstudie zeigten sich größere Differenzen im Aufwand zwischen den Modellierungstechniken, bedingt durch unterschiedlich reichhaltige Notationen (z.B. SCR vs. UML). Diese Differenzen wurden in dieser Fallstudie durch den Einsatz von CASE-Werkzeugen nivelliert. Der absolute Aufwand stieg durch die Einführung von CASE-Werkzeuge um ca. 10-50%, wobei sich aber auch die Qualität der Spezifikationen stark verbesserte. Differenzen zwischen den Modellierungstechniken hinsichtlich der Anzahl der Mängel, die durch die Formalisierung aufdeckt wurden, sind durch den Einsatz von CASE-Werkzeugen ebenfalls nivelliert worden.
Das wesentliche Resultat der Studie betrifft nicht die Modellierungstechniken, sondern das Aufdecken von Mängeln in informellen Anforderungen bei der Modellierung im allgemeinen. Die Tabelle 4 zeigt die Fortpflanzung von Mängeln aus den Anforderungen in die Spezifikation nach den unterschiedlichen Arten von Mängeln und Häufigkeit prozentual aufgeschlüsselt.

Mangel	Unvollständigkeit	Mehrdeutigkeit	Widerspruch
# nicht gemeldete Mängel	0	3	1
% außerhalb des Models	3	4	0
% Meldungen	34	23	40
% übernommene Mängel	21	0	20
% richtig gelöste Mängel	37	46	35
% falsch gelöste Mängel	5	27	5

Tabelle 4: Propagierung von Mängeln nach Arten

Mehrdeutigkeiten wurden öfter völlig übersehen (d.h. von keinem Team gemeldet), als Widersprüche oder Unvollständigkeiten. Mehrdeutigkeiten wurden während des Modellierens auch prozentual gesehen am wenigsten gemeldet, während Widersprüche am auffälligsten waren. Unvollständigkeiten und Widersprüche wurden direkt in die Spezifikation übernommen, wenn sie übersehen wurden, Mehrdeutigkeiten hingegen gar nicht. Die Mehrdeutigkeiten wurden in der vorliegenden Fallstudie in allen Fällen zu "Eindeutigkeiten", d.h. ihre ursprüngliche Mehrdeutigkeit kann nicht mehr als solche auffallen. Die Mehrdeutigkeiten wurden zwar zu 46% bewußt bzw. unbewußt korrekt gelöst, aber auch zu 27% falsch. Im Vergleich wurden Unvollständigkeiten zu 37%/5% korrekt bzw. falsch gelöst und Widersprüche zu 35%/5%. Fazit ist, daß es einen signifikanten Unterschied zwischen den Arten von Mängeln in informellen Anforderungen, die durch Anwendung von Modellierungstechniken identifiziert werden können, gibt. Auf Unvollständigkeiten und Widersprüche wird der Spezifizierer bei der Modellierung gestoßen, auf Mehrdeutigkeiten nicht. Dies ist ein Problem, da Mehrdeutigkeiten durch die Modellierung eindeutig gemacht werden, diese unbewußte Lösung jedoch öfter zu Fehlern führt, als vergleichsweise bei Unvollständigkeiten, oder Widersprüchen. Zwischen den Modellierungstechniken konnten keine signifikanten Unterschiede bezüglich der Aufdeckung von Mängeln festgestellt werden.
Die Resultate wirken im Vergleich zum Personenaufwand vielleicht etwas unspektakulär, es muß aber folgendes bedacht werden: das Tamagotchi ist nicht nur von seiner Komplexität, sondern auch von der Art der Anforderungen her einfach strukturiert. Würde eine Modellierungstechnik bereits hier Probleme bereiten, dann wäre sie für den industriellen Einsatz gänzlich ungeeignet. Für die Durchführung weiterer Studien stellen sich mehrere Fragen: welche Arten von Spezifikationsbeispielen sollten verwendet werden? Welche Größen sind aus der Sicht der Industrie interessant? Wie können die interessierenden Größen zuverlässig gemessen werden ? Das Design der Fallstudie an sich und die Kooperation zwischen den Hochschulen hat sich bewährt. Die Modellierungstechniken konnten besser betreut werden, und die Ergebnisse sind durch die vergleichsweise große Anzahl von Teilnehmern relativ zuverlässig. Wir streben daher eine Wiederholung der Studie, gerne auch unter Beteiligung weiterer Arbeitsgruppen, an.

5 Literatur

[BDD+92] M. Broy, C. Dendorfer, F. Dederichs, M. Fuchs, T. Gritzner, and R. Weber. The design of distributed systems - An introduction to Focus. TUM-I 0192, Technical University of Munich, 1992.

[DD86] H. L. Dreyfus and S. E. Dreyfus. *Mind over Machines*. The Free Press, New York, 1986.

[Gra97] Bernd Grahlmann. The PEP Tool. In *Tool Presentations of ATPN'97 (Application and Theory of Petri Nets)*, June 1997.

[HBGL95] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proceedings of the 10th Annual Conf. on Computer Assurance*, pages 109–122, Gaithersburg, MD, USA, June 1995.

[HMR+98] Franz Huber, Sascha Molterer, Andreas Rausch, Bernhard Schätz, Marc Sihling, and Oscar Slotosch. Tool supported specification and simulation of distributed systems. In Bernd Krämer, Naoshi Uchihira, Peter Croll, and Stefano Russo, editors, *Proceedings International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 155–164, 1998.

[KKR+98] Antje von Knethen, Erik Kamsties, Ralf Reussner, Christian Bunse, and Bin Shen. A comparative case study with industrial requirements engineering methods. In *Proceedings of the 11th International Conference on Software Engineering an its Applications*, Paris, France, December 8–10 1998.

sE – synchronousEifel
A Design and Programming Environment*

Reinhard Budde, Axel Poigné, Karl-Heinz Sylla

Abstract

We summarise the main design ideas of the object-oriented language synchronousEifel and present the tools available in the programming environment. The language focuses on the design and implementation of embedded software which has to satisfy hard and soft real-time constraints, and which requires a high level of dependability. Classes may define a reactive behaviour in addition to routines and attributes. The reactive behaviour is specified either by a hierarchical state machine or in an imperative style. The language and the tool environment are designed to support developers in gaining and increasing confidence into a construction under development. The compiler generates highly efficient and compact code which even maps to small micro-controllers.

1 Motivation

Embedded reactive systems are crucial components for plants, robots, or cars, up to household goods. In order to decrease the production costs and to gain flexibility such devices are built using micro-controllers with an increasing share of software. Such software should be dependable since malfunction may be dangerous or may involve financial risk.

With synchronousEifel, we provide a design environment which supports the following design and programming paradigms:

- *Synchronous modelling*, for constructing reactive real-time components, and for enabling system validation by model-checking.
- *Object-oriented modelling* in a strongly typed language, which is well suited for a robust and flexible design of complex systems.

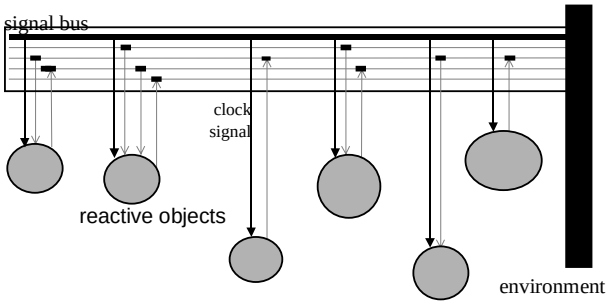
We support multiple formalisms for specifying reactive behaviour: imperative, state machine-like, data flow, logic. We provide a smooth and well defined integration of the object-oriented and of the synchronous execution model, and an easy way of defining reliable and reusable components.

2 The Synchronous Execution Model

An embedded reactive system interacts with its environment via signals. Input signals may carry sensor-data and output signals may trigger actuators, for instance. A systems reaction depends on its internal state and on the value of input-signals. It generates output-signals and changes the internal state.

The synchronous execution model reflects the basic idea of digital hardware design and of many engineering formalisms: Processing proceeds in steps controlled by a trigger signal, a “clock”. When the trigger signal is present, or “when the clock ticks”, a system starts to react and the reaction is finished in time before the next trigger signal is present. Such a processing step is called an instant.

In synchronousEifel the hard real-time part of an application is based on this execution model.



Signals are broadcast via a signal bus to which all so called “reactive” objects connect, linking input and output signals to the respective ”wires” of the signal bus. At the tick of the clock, all signals provided by the environment are made available to all objects, signals may be emitted and the internal state may change.

* The work is partially funded by the ESPRIT LTR “SYRF” (Project No. 22703) and ESPRIT IiM “CRISYS” (Project No. 25514)

3 Reactive Classes: An Example

Reactive Classes consist of two parts:

- A description of the synchronous behaviour. As notations we use well known formalisms like Statecharts, as well as other styles of synchronous programming like Esterel and Lustre.
- Descriptions of routines and attributes. In case of reactive classes all features must be declared private.

The public interface of an reactive object consists of a specification of the input and the output signals only, its features may only be internally accessed from its reactive part.

Classes which are not reactive define objects as usual with public features being made available to clients. Below we present a simple reactive class to give a rough idea of its two parts.

```
reactive class Timer
creation
  set ( d : Integer) is do
    latch := d
  end;
-- the signal interface
input START; CLOCK;
output ELAPSED;
-- the reactive behaviour
reactive is do
  loop
    await START; reset();
    next; -- await the next tick
    terminate
      loop
        await CLOCK;
        decrement();
        next;
      end;
    when is_elapsed() then
      emit ELAPSED;
    end;
  end;
end;
-- routines and attributes
feature [None]
  counter : Integer;
  latch is const : Integer;
  reset is do
    counter := latch;
  end;
  decrement is do
    if counter > 0 then
      counter := counter – 1;
    end;
  end;
  is_elapsed : Boolean is do
    result := (counter = 0);
  end;
-- scheduling
sequence
  s: decrement < is_elapsed < reset
end class Timer
```

The reactive part should be rather self-explaining: A Timer-object initially waits for the START signal. If this signal is present the counter is set to a start value by calling the feature `reset()`. At each CLOCK signal the counter decrements until the condition `is_elapsed()` holds. It is important to notice that "time passes" only when waiting. Reactions are considered instantaneous: If the CLOCK signal is present the counter decrements at the same instant and the termination condition `is_elapsed` is evaluated anyway.

The concept of an instantaneous reaction makes possible time races explicit. In order to avoid nondeterminism we have to specify the desired sequence of feature executions (here: `decrement` before `is_elapsed` before `reset`). Possible time races will be detected by the compiler based on the analysis of the data flow and the state transitions. The programmer is required to specify the scheduling of feature calls through sequence clauses, if necessary.

The avoidance of time races has considerably influenced the language design. Features in reactive objects must be private to restrict time races to objects. Objects are interfaced via the signal bus only by which data can be exchanged. Additionally, time races may be present in the inter-object communication. These are read/write conflicts on the signal bus and can be detected by a uniform causality analysis. Programs with "causality cycles" are rejected. Any synchronousEifel program which passes the compiler is deterministic.

3 The Real-Time Kernel of an Application

The object configuration of an application is defined by a root class, e.g.

```
root class TimerApp
timing 1 ms;
creation
  run is do
    t1.set (800);
    t2.set (1000);
    reactive loop;
  end;
-- signal interface and signal bus
input
  START = t1.START = t2.START; --(*)
  CLOCK = t1.CLOCK = t2.CLOCK;
output
  T1_ELAPSED = t1.ELAPSED;
  T2_ELAPSED = t2.ELAPSED;
-- reactive objects
reactive t1, t2 : Timer;
end class TimerApp
```

The external interface and the connection of reactive objects to the signal bus are defined simultaneously, e.g. the declaration (*) defines the input signal START and connects it to the respective START signals of the timers

`t1` and `t2`. An application is started by instantiation of a singleton object of the root class and by execution of the creation feature `run`. The statement `reactive loop` called by the creation routine starts the reactive engine. The set of reactive objects is defined statically. The composition of their reactive parts determines the reactive behaviour of the application. This "kernel" can be checked to satisfy hard real-time constraints as described now:

The timing clause of a root class specifies the minimal delay between two reactions. Each reaction is atomic and its execution must terminate within this minimal delay time. Whether this "synchrony hypothesis" holds can be checked by a tool. This tool evaluates the worst case execution time of all features for a given combination of target processor and C-Compiler. The worst case execution time of an instant is determined by the maximal sum of feature execution times with regard to all possible reactions.

The reactive kernel may be considered as a real-time operating system which provides a precompiled static scheduling for the controlled activities.

4 Concerning Soft Real-Time

The kernel may trigger features to be executed in separate threads using the statement:

`schedule` *feature-call* `;`

Threads can be interrupted, They will at least be interrupted by the atomic reactions of the kernel which have highest priority to guarantee the hard real-time constraints. A thread may read the current state of an output signal. It may set an input signal to be present at the next reaction using the statement

`emit next instant` *signal-id* `;`

Up to now we provide only these facilities, but we do not yet provide scheduling strategies or support for the analysis and the checking of soft real-time constraints.

5 The Development Environment

synchronousEifel is a strong typed object-oriented language. We have designed it in accordance with the language Eiffel. Design by contract, multiple inheritance and generic types are supported. Compared to Eiffel subclassing is restricted to introducing subtypes.

The environment supports compilation, configuration, simulation, test, and verification of synchronous object-oriented programs. Behavioural descriptions may be edited in graphical or in textual form. Code generators for efficient and compact code in C and diverse hardware formats, e.g. Verilog, are available.

For model-checking, code is generated which is accepted either by the VIS or SMV model checkers. Specification of properties either in temporal branching time logic (CTL) and Past Time Logic (PTL) are supported.

The compilation and also the optimisation of reactive behaviour is compositional and may be performed separately for each class. Thus the size of the model of an application can be reduced considerably. This enlarges the size of applications which can be validated by model-checking.

For validation also, so called, "synchronous observers" may be defined. An observer is a reactive program constructed to detect defective conditions. The non-occurrence of a defect may be model-checked for the application combined with the observer, or the observer program is executed in parallel to the application, thus serving as a watchdog.

The Team, the Product and Partners

The synchronousEifel Environment and Tools are developed in the team VerS (Dependable Real-Time Systems): Reinhard Budde, Monika Müllerburg, Marie-Luise Christ-Neumann, Axel Poigné and Karl-Heinz Sylla.

Industrial partners in ESPRIT projects are Aerospatiale (F), Electricite de France (F), Elf (F), Prover Technoloy (S), SAAB MA (S), Schneider Electric (F), Siemens Electrocom (D), Verilog (F). Additional cooperations exist with Bosch and Daimler-Benz Research.

synchronousEifel is delivered with a public domain license. The products site is `http://ais.gmd.de/~budde/se_home.html`
Contact the team at `Reinhard.Budde@gmd.de`

Building Real-Time Applications with Rose RealTime

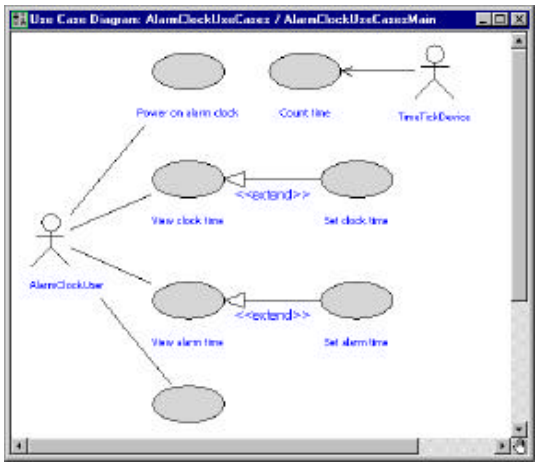
Rainer Knödlseder
Rational Software
D-82041 Oberhaching
Keltenring 15
rknoedls@rational.com

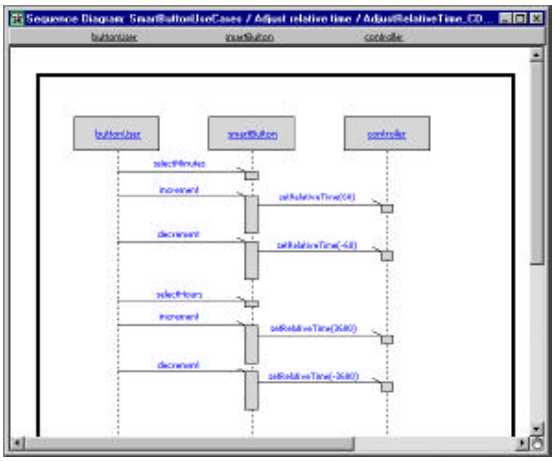
Today, in real-time business, we have to face a common challenge: To produce more products, in less time and with fewer peoples. Yet the complexity of real-time development projects steadily increases, caused e.g. by the need for distributed or concurrent systems with massive dynamic behavior. But nevertheless, most applications are build using C programming language or even assembler.
And there is also another big challenge: Requirements change very fast, customization of existing application is often necessary an time-to-market is a strong need.

To face these problems, many companies think about using object oriented modeling techniques and to use the well accepted **U**nified **M**odeling **L**anguage (UML) to model their applications. In the past, this was often done using traditional Case Tools like Rational Rose. But with this standard tools, there is a gap: Visual modeling is supported but coding of the application code must be done by hand. To overcome this gap, some people started in the late 1980s to build a visual development tool for real time development. This toolset is called **ObjecTime Developer** and was used in many commercial projects to create complex real-time applications in-time with improved quality and functionality.
Today, all the world uses the UML instead of the ROOM (**R**eal-time **O**bject **O**riented) **M**odeling language [1], used in ObjecTime Developer. So Rational Software and ObjecTime Limited started in 1997 to join the benefits of Rational Rose with the field proved code generation of ObjecTime Developer.
Rational Rose RealTime is available since April, 1 and it's goals are to

- support the software developer in visual development of his application
- use UML **as the source** for real-time applications
- generate high quality and highly reusable C++ application code
- allow integration of existing C or C++ code
- help software developers to test their application on the development host and on the target

To develop an application with Rose RealTime, an iterative and incremental process like the **Rational Unified Process** should be used. Following this process, the first step during software development is to describe some requirements. These requirements are described in Rose RealTime using Use Case diagrams.

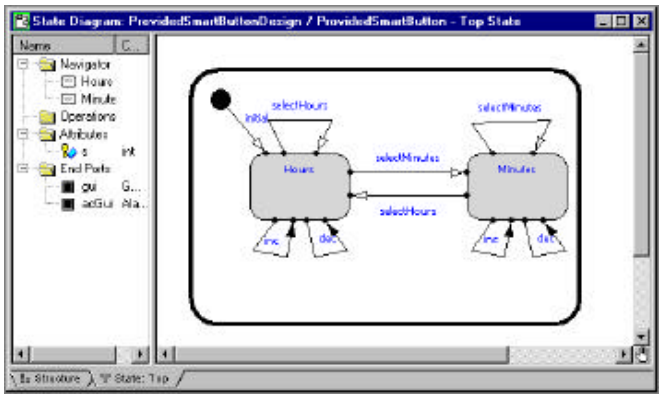




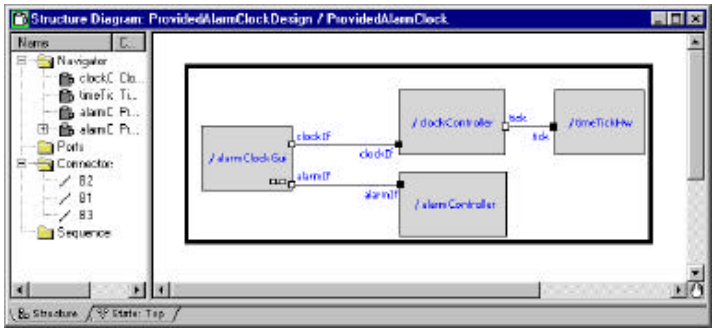
The Use Cases may be described in text. Another way, often used in event driven real-time applications, is the use of sequence diagrams, also provided by Rational Rose RealTime. Another way to manage requirements is to use the integration with requirement management tools like **Rational RequisitePro**.

When some requirements are fixed, the software developer start the design of a part of the application. Typical real-time applications consist of a large number of active components, representing the Use Cases defined before. In Rational Rose RealTime, these active components are represented by **capsules**.
Capsules are a special stereotype of UML classes [2] [3] and have some special features:

- they communicate by each other using messages
- they may be distributed on different tasks or even on different processors
- they have a behavior, represented by an UML state chart.
- state transitions are triggered by messages, sent to capsules via **ports**.
- in a state transition, new messages may be sent. Also passive data classes or existing code may be used by standard function calls
- they support inheritance, allowing another capsule to specialize the behavior of a capsule.
- they handle concurrency problems using a run-to-competition mechanism, preventing the software developer from the need to deal with semaphores etc.
- they use C or C++ detail action code which is executed when a state transition is performed. By this code, functions may be called, internal attributes may be updated or new messages can be sent



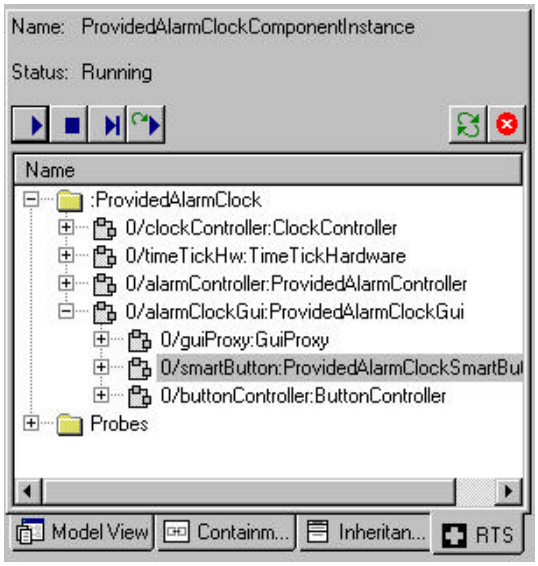
To create high reusable code, the encapsulation shell of a capsule is not only one-way (as usual in OO languages) but in both directions. This means, if an capsule wants to interact with another capsule, this must be done using a port. All messages, which may be sent through a port are defined by a **protocol**. A protocol is a special stereotype of a UML class and defines a set of input and output messages. Each message may be sent in a synchronous or asynchronous way and consist of a signal and optional data and priority. As before, inheritance is also supported by protocols.
In general, real-time applications are formed by several capsules. These capsules are grouped in UML collaboration diagrams. In these diagrams, the ports of the capsules are connected using **connectors**.

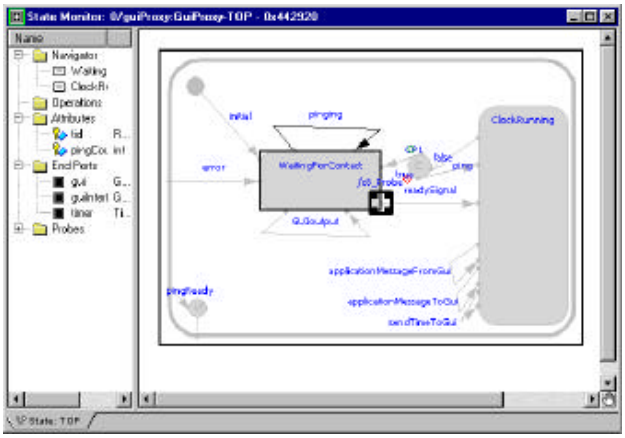


When the software developer has designed some capsules, it's only a few clicks to create the application from the UML source. Depending on the given configuration, the generated application may be executed on the host (Windows NT, Solaris, HPUX) or on the target. In this case, many popular operating systems like VxWorks, pSOS, VRTX or OSE are supported.

When some capsules are designed, the software developer should start to test his application. This should be started on the host, using the target observability feature of Rose RealTime. With target observability, a tester may:

- run /stop the application or execute in single step mode
- view the dynamic behavior of each capsule's state machine
- trace messages, sent through ports
- generate new messages to test specific conditions
- stop the application when a special message triggers a capsule
- generate sequence diagrams of the application's dynamic behavior
- do source code debugging of the transition code using source code debugger like Microsoft Visual Studio





Another way to improve testing is to use additional test tools like Rational Purify or Rational Quantify.
When tests on the host passed successfully, it's only a change in configuration and the application can run on target. If there is a TCP/IP connection available between host and target, target observability as described before is also available on target.

The model can be stored in a standard file or using asourcecode management system like Rational**ClearCase**. This can be done in one single file or each component (package, capsule, protocol, data class) can also be stored in an individual file to support application development in larger teams.

At this point, an iteration in the development process is over and the next iteration may start. The software developer may choose some additional requirements to be realized in the next development step.

References

[1] Selic, B., G. Gullekson, and P. Ward, *Real-Time Object-Oriented Modeling,* John Wiley & Sons, New York, NY, 1994.

[2] Selic, B., "Using UML to Model Complex Real-Time Architectures", OMER-Workshop., May 1999.

[3] Selic, B. and J. Rumbaugh, "Using UML for Modeling Complex Real-Time Systems," ObjecTime Limited/Rational Software Corp. white paper, March 1998.

Instantaneous System Generation and Validation

Rainer Gerlich, BSSE System and Software Engineering

Auf dem Ruhbuehl 181, D-88090 Immenstaad
Phone: +49/7545/91.12.58 Mobile: +49/171/80.20.659 Fax: +49/7545/91.12.40
e-mail: gerlich@t-online.de internet: http://home.t-online.de/home/gerlich/

1. INTRODUCTION

"Instantaneous System and Software Generation" (ISG) introduces an abstraction level into the development process which allows an engineer to implement a system by entities of his world not being forced to deal with software engineering details, but allowing him to get the real implementation of his ideas. In addition, ISG provides means for behavioural, functional and performance validation and stress testing.

ISG is based on object-oriented ideas like encapsulation, interfaces and abstraction, but it goes beyond the capabilities of object-oriented approaches like inheritance by use of code generators because contruction rules can cover a wider range in view of reuse.

ISG allows to express system properties like data exchange, topology and performance by parameters from which the system specific source code and data are automatically generated as needed for execution. This is achieved by encapsulation of system properties by a specific organsiation of a system component, separation of system functionality and topology, standardisation of interfaces, and abstract parametrisation of communication, topology and performance. The automatically generated code is complemented by libraries and other user-defined functions.

Due to the formal definition of system behaviour tests can be derived automatically, and faults can be injected for stress testing. By introduction of a "communication" layer and "logical" communication channels transparent distribution, fault tolerance (redundancy management), and system (re-)configuration are inherently supported.

Usual development methods intend to simplify the problem by introduction of "views" like "Object Models, Dynamic Models, Functional Models" for OMT, "state charts, activity charts, module charts" as suggested by Harel. This makes it difficult to keep the full view on a system's properties due to separation of information. In principle, this is in conflict with encapsulation.

ISG encapsulates the three different views into one piece of information which covers the data flow, the data processing and the performance: the definition is based on Finite State Machines enriched by communication and performance information. The (needed) reduction of complexity is achieved by hiding of implemtation details: the detailed implementation is peformed according to "construction rules" which take the system parameters and convert them into source code. This even works well in case of hard-real-time processing. On this high abstraction level an engineer can express scheduling strategies, monitor deadlines, implement timeouts for responses, setup cyclic or one-shot tasks

The ISG approach allows for easy change of system architecture and maintenance, incremental development, immediate feedback from the real system right from the beginning, validation by simulation and on the target system. Typically, having provided the formal description of system properties, after 15 minutes a feedback is available from the automatically generated and executed system.

2. THE REALISATION

2.1 Overview

The principal idea (Fig. 2-1) is to provide a facility which allows

- to generate a system (program) from a minimum of engineering information on high abstraction level,
- to automate the generation of an executable system and to provide the environments for simulation and the target system within minutes, and

- to provide an immediate feedback from the executing system by graphical figures and reports.

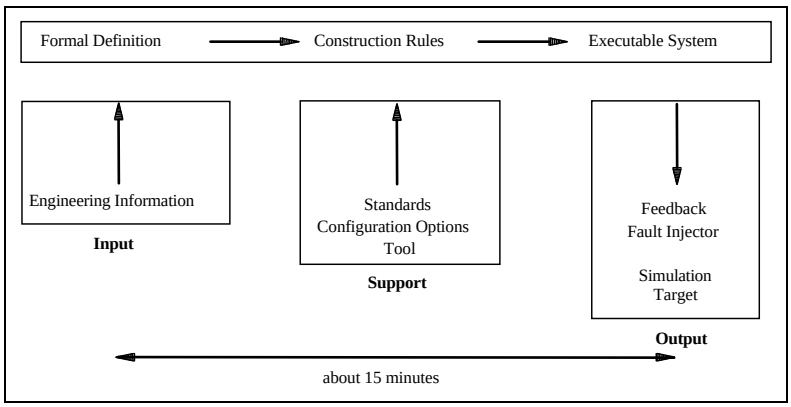


Fig. 2-1: Automated and Instantaneous Construction of a System from Engineering Information

ISG takes the information about the architecture and topology, the behaviour as expressed by "input-processing-output", the performance figures and constraints and converts them directly into an executable program which can be configured either for simulation or for execution on the target system. ISG allows a system engineer to immediately obtain a feedback on his (first) ideas and to perform iterations until a satisfying solution is achieved. The functional refinement of the system can be done in an incremental manner by plugging in the corresponding functions into the behavioural skeleton established by ISG.

2.2 The Organisation

ISG takes a formal specification of behaviour, performance, topology and communication based on Finite State Machines which is delivered by the engineer by a table (Fig. 2-2). However, a user is not forced to define "big" state machines, if desired it is sufficient to define just one state. Additionally, generation and configuration options have to be provided by two more files. The ISG toolset takes this information and builts automatically a program.

For automated (stress) testing the fault injection feature is available which will insert additional code or modify the code in order to inject faults into the system during execution.

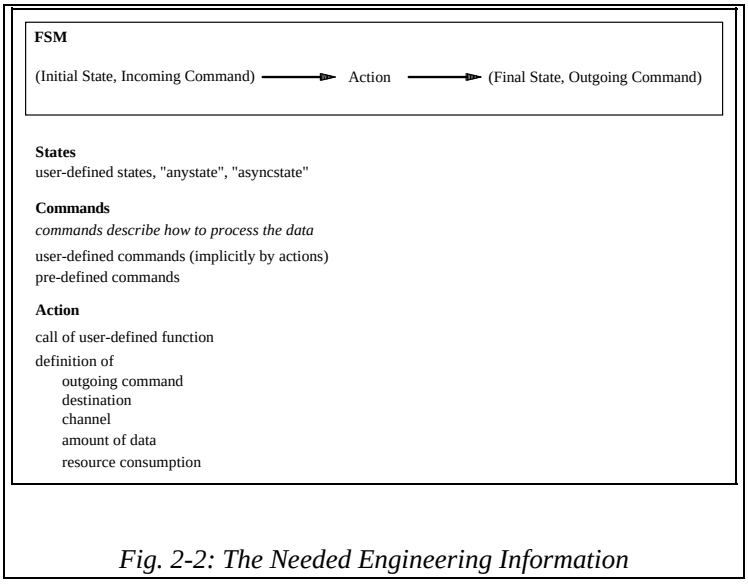


Fig. 2-2: The Needed Engineering Information

As far as detailed functional algorithms are missing ISG provides decision criteria for execution of the program logics in order to allow for automated testing. Actions as defined by the engineer are either selected randomly or sequentially one after the other. In latter case a repetition factor can be given so that a group of actions may be repeated before the next group is executed.

2.3 Feedback and Visualisation

As feedback the program generates reports, data flow and timing diagrams.

Reports inform about coverage of executed actions, states and state transitions, execution time, consumption of resources, injected faults and observed exceptions like a time-out or a cycle overrun. The delivered data may also be displayed by a spreadsheet program.

For tracking of data flow "Message Sequence Charts" (MSC) are produced (Fig. 2-3), but on request any other format can be supported as well.

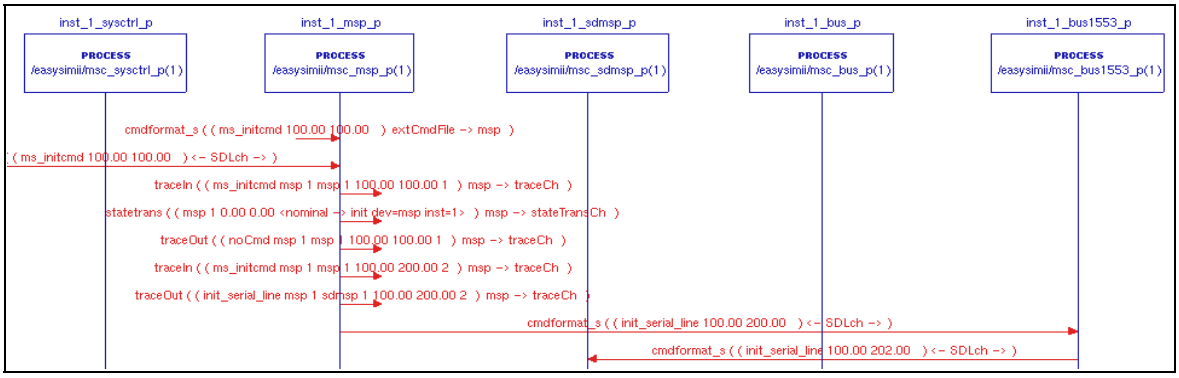


Fig. 2-3: A Sample Message Sequence Chart

Timing diagrams (Fig. 2-4) allow to track the data flow over time at certain locations which may be processes or devices like buses. A user may click on a certain event and the contents of the message will be displayed (Fig. 2-5). In case of a long message also the attached data may be displayed. By filter criteria a user may select a certain subset of the data flow.

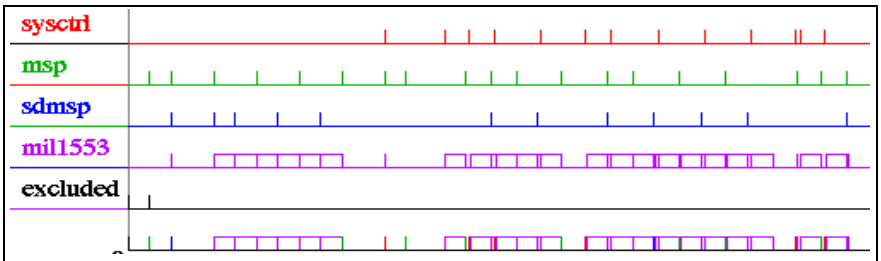


Fig. 2-4: A Sample Timing Diagram

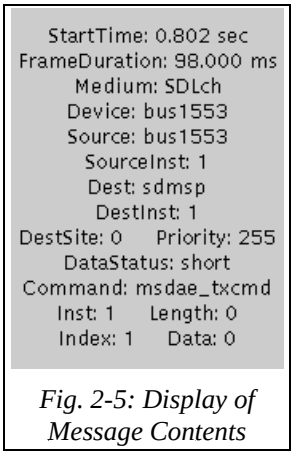


Fig. 2-5: Display of Message Contents

2.4 Real-Time Processing

The ISG toolset supports real-time processing. Timer events and timeouts can be created and be reset, cyclic tasks are supported and their completion within the given deadline can be monitored.

2.5 Fault Injection

Fault injection is supported by: (1) injection of erroneous commands / data, and (2) loss of signals / message. Fault injection may either be done explicitly by a user or automatically by the toolset.

2.6 Automated Testing

By a configuration file a user may specify which data flow and which process shall be subject of automated testing. This specification may be given explicitly or by rules implying wild cards. Then the ISG toolset will establish test commands which are cyclically issued until a certain coverage is achieved or another stop criterion is fulfilled.

2.7 Integration with Existing Software

The ISG toolset provides the capability for integration of existing software. This may happen in twofold manner: (1) by linking this software directly with the ISG generated software, or (2) by remote access. In case of event-driven simulation it is possible to synchronise all executables.

2.8 Benefits

By instantaneous and automated generation of the program the effort for implementation of the first version and of architectural changes is reduced because just a run of the ISG tools is required to get an executable. This way a number of iterations can be performed at the beginning of the lifecycle and continuously until the end of the lifecycle. This allows to immediately evaluate the behaviour and performance of a certain architecture and to find the optimum solution at a minimum of costs.

The specification may be given incrementally and an engineer can get a first feedback before the end of the first day of a project.

Also, the algorithms can be inserted incrementally and the stubs as generated by ISG may be replaced step by step by the real functions. After each such step the testing and validation procedures can immediately be applied.

At any time the system or program can be executed and provides a feedback about behaviour and performance. This minimises the risks. Moreover, no effort is required to implement the architectural skeleton and the mechanisms for the data flow because this is done automatically by the ISG toolset.

3. CONSTRUCTION RULES AND STANDARDISATION

During a number of previous projects it was observed that reuse suffered from the low degree of standardisation: the more standards are applied the higher is the percentage of reuse. Moreover, if the standards also cover the rules of communication and internal organisation of a process a skeleton can automatically be generated from a minimum of information. A number of standardisation and organisation principles were applied and continuously improved from project to project until the fully automated generation really could take place.

The main point is the introduction of a standard for intercommunication which covers all the information needed for data exchange and distribution, and which can be applied to every project and application. Moreover, it allows to formalise the specification process and to automate the start-up procedure of a new project.

This standardisation allows to introduce a generic, reusable architecture for transparent distribution, management of redundant components and incremental development. It decouples (in object-oriented manner) the data transfer from data processing. and supports any communication topology.

The data exchange format includes information about sender, receiver and the type of data processing at the receiver.

Due to the information about the receiver a routing software package can transparently forward the data across a network. This is similar to CORBA but also supports a heterogeneous set of physical channels like RS232, files, screens, IPC and TCP/IP and fulfills the needs of real-time processing.

By the included information about how to process the data (called a "command") a receiver is advised what to do with the associated data and informed which type of information is included. Hence, the processing capabilities of a system are expressed by a set of commands exchanged between the system's components ("objects" in O-O terminology) or internally used by a component (object).

This principal approach is complemented by introduction of states in order to allow a process to remember for previous actions and to perform different actions according to the history and the resulting actual situation.

Consequently, a system and its capabilities can be defined by a set of states and commands, a number of components (like processes or devices), a set of communication channels and resources (like CPU, buses). This allows to formalise (a) the specification of a system's capabilities and (b) the organisation of a system's structure and development and to efficiently support a user when he is going to start a new project and to built a new system.

Due to the standard format the existing routines for data communication and processing can be reused and a system's properties can be expressed by engineering information like commands, devices, resources and associated data describing topology, functionality and performance.

The ISG procedure minimises the required amount of engineering information and allows to configure and instrument the generated code either for simulation and validation or for the target system.

4. COMPARISON WITH OTHER METHODS AND TOOLS

The ISG/V toolset aims to automate the development procedure, to increase the degree of reuse and to reduce the development risks.

To allow for automation a standardised interface for communication and a formalised, generic concept for construction of the software are introduced. This approach is based on object-oriented ideas like encapsulation, abstraction, information hiding and clear interfaces.

This basic concept was applied to several projects from different application areas and the results have been evaluated. An analysis of the achieved efficiency showed that inheritance and the class concept are not sufficient to reduce the development costs and time. In case of structural changes from one application to another, inheritance and the class concept still require manual intervention to adapt and implement the desired code.

For the projects formal methods were applied for early system validation. The experience related to use of (semi-)formal methods like SDL lead to the idea to replace inheritance and the class concept by formal construction rules and to generate the instances of a class according to given rules rather than to derive them from classes. This allows to cover strucutral differences by an automated approach.

The use of construction rules may be interpreted as a generalisation of the class concept: the inheritance rules are represented by some part of the construction rules, while the remaining part of the rules allow to cover additional, structural instantiation parameter.

As most of the development actiivities are automated, consequently, there is no need to apply most of the concepts and related graphical representations like class diagrams. deployment diagrams etc. Of course, this eases development and reduces effort and costs. Hence, the reduced number of diagrams is an advantage: processing of a diagram consumes time, the lower the number of diagrams to be processed the lower are the costs.

The objects include an extended concept of Finite State Machines (FSM).The extension allows to execute standard, state-independent or high-priority activities in parallel to a FSM. The automata communicate via a standard protocol and standard data format.

Due to the standardised interface the topology of a system can be soft-coded and be defined by rules which associate objects with channels. Even wild cards are allowed. This mechanism also transparently covers channel redundancy and reconfiguration.

The toolset requires engineering information as input only and no experience with software implementation. From this information the code for the whole system is generated immediately from well-readable templates. The code is usable on the target system as well.

The information has to be provided by templates which allow for a formal check of correctness and completeness. By simple directives the user-provided information may be expanded towards automated testing and property checking like for deadlines and timeouts.

By instrumentation of the code data flow and timing diagrams may be generated and information to other verification and validation toolsets may be provided. Also, coverage and performance analysis may be activated.

Due to inherent support of behavioural and performance validation a representative simulation is possible. From simulation an immediate transition to the target system may be done in order to compare the results and to identify differences between the assumed and the real properties.

Compared with other code generators the ISG/V toolset not only generates code, it also provides all files which are needed for execution and manages all steps which follow the provision of the engineering information up to evaluation of the results of execution.

This makes system development very efficient.