Applying Modern Software Design Principles: A CAN Tool Based on *Extensibility*

Markus Weseloh and Roland Rüdiger[1]

Software tools can be helpful in understanding the behaviour of complex technical systems. Quite often, however, present-day tools have to be accepted by the user on an "as is" basis without an easy way to customize the tool. New ideas and techniques of modern software technology promise a much higher degree of adaptability of software systems and, in particular, of graphical user interfaces (GUIs). This paper presents an architecture and a prototypical realization of an experimental tool designed along the lines of these novel ideas. The tool is based on a hierarchy of model layers for analyzing the temporal behaviour of CAN-systems. The underlying mathematical key ideas include: worst case analysis of response times, probabilistic error model, and quality measure of timeliness. The architecture allows to easily extend the system and to exchange, among other things, the basic equations, the algorithms, and the probability distributions involved without affecting other parts of the system.

1 Introduction

During the last years many new ideas have evolved in software technology. Typically and most often, these are applied to problems close to software technology itself, e.g. data structures, algorithms, compiler construction, etc. Therefore, it seems worthwhile to see whether these ideas could also be of benefit to other areas, e. g. technical computer science and whether they could be an alternative to methods of *ad-hoc* programming still common to many technically oriented areas.

One example where some of these new conceptions can be applied successfully appears to be the design and implementation of a tool for analyzing the temporal behaviour of CAN systems. This paper presents an architecture and a prototypical realization of an experimental tool designed along the lines of some novel ideas of software technology. The tool is based on a hierarchy of layers each of which models some specific aspects of the temporal behaviour of CAN-systems. The paper is organized as follows: Section 2 gives a brief account of the notion of *software extensibility* with special emphasis on the ideas developed by the group of Niklaus Wirth and Jürg Gutknecht at the Swiss Federal Institute for Technology (ETH) Zürich, in the Oberon project. Section 3 describes these layers informally and Section 4 introduces the design of the large scale program structure and discusses some details of the hierarchy including the basic structure (Layer 0) and the key ideas such as worst case analysis of response times (Layer 1), probabilistic error model (Layer 2), and quality measure of timeliness (Layer 3). The following Section 5 presents our realization of essentially two tool variants, based on the languages Oberon (Section 5.1) and Java (Section 5.2), which have been implemented so far or are—at the time of writing—in the state of being implemented. Finally, Section 6 concludes the paper with a summary and some perspectives on future work.

2 Software Extensibility

Extending a software system means to add functionality to an existing system. The obvious and naive way is, of course, to edit the source—if it is available—and extend it by paste-and-copy techniques. One problem with this procedure is that it is very likely to re-introduce bugs into a program which

[1]Postal address: Prof. Dr. habil. Roland Rüdiger, Institute for Distributed Systems, University of Applied Sciences (FH) Braunschweig/Wolfenbüttel, Salzdahlumer Straße 46/48, D-38302 Wolfenbüttel, Germany

might have been well designed and tested and to destroy its structure. Nowadays, there exists a wide variety of techniques to re-use existing software components without changing or even recompiling the source code.

The Oberon project was one such approach that resulted in a system of "exemplary lucidity, efficiency, and compactness" (Reiser in [Rei91]). A technically detailed report is [WG92], the first user guide and programmer's manual of the Oberon system in its original form is given in [Rei91], and an introduction of the Oberon language may be found in [RW92]. One of the main guidelines in the project was to achieve *extensibility*, which soon got some specific meaning and potential in both the operating system and the language Oberon. Writes Gutknecht [Gut96]: "Perhaps to our own surprise, we soon recognize that the new construct of type extension in combination with the old concept of procedure variable provides an absolutely sufficient language framework for the creation of amazingly rich and flexible object-oriented sceneries ... ." *Extensibility* has been discussed in various contexts, see [Fra94, Szy96]; a review may also be found in [Rü96]. Another evolutionary step is the Oberon System 3 with its *gadgets* [Gut94, Gut96, FM98], which has been chosen as a design and implementation basis for the work presented on this paper.

3 The Model Layers

During the last decade several models have been proposed which have proven to be helpful in analyzing and evaluating the temporal behaviour of CAN systems. In a couple of papers by Wang *et al.* [WLHS92] and Tindell *et al.* [TBW94, TB94, THW94, TBW95] a formalism for analyzing the worst case behaviour of response times has been described.

It has been claimed that this model gives an adequate description of periodic as well as sporadic CAN messages. The basic underlying assumption is that "a given message m cannot be generated more than once every T_m time units" (assumption 1 in [Law97, p. 254]). Although this is certainly a useful approximation in many situations, one has to keep in mind, that, on the one hand, a Poisson stream of messages does of course not meet this assumption and, on the other hand, any other distribution lacks the memoryless property; so, if any such distribution would be realistic, the process, which generates sporadic CAN messages, must have some internal memory to stick to the constraint of this assumption. In this case, however, a sporadic message can, in a worst case analysis, indeed be treated like a periodic message, because *no more than once* in a worst case scenario is the same as *exactly once*. Therefore, in the present paper, this model simply is termed "deterministic".

Based on this model the stochastic error model by Navet *et al.* [NS97, NSS99] introduces a major new aspect; in their model the realtime behaviour is governed by a probabilistic law. The essential idea is that the occurrence of a large number of errors will always result in a violation of timing constraints. So, it is natural to determine, for each CAN message class, the limiting maximal number of errors such that its deadline is met. Based on some assumptions on stochastic properties of error events this then allows to determine the probabilities that the individual deadlines will be met.

In [Rü98] this model is used to establish a quality measure of timeliness in noisy environments: following an idea of traditional queuing theory a cost function can be introduced; if one of the CAN messages occasionally misses its deadline there will be (fictitious) costs resulting from this event. The resulting average can be taken as a measure of the system's realtime quality.

All of these models require that some numerical problems will be solved including a fixed point problem and the calculation of convolution powers. So, from a practitioner's point of view, the models would be useless unless the algorithms involved will be implemented in a tool.

A close examination of the structure of the models described so far reveals a remarkable property: the models form, in a natural way, a hierarchy of layers with an internal inheritance structure and, furthermore, this structure can easily be put into an extensible form as described in Section 2. In the following the model layers will be described informally. Details and the formal machinery, which will not be repeated here, can be found in [Rü98] and the references given therein.

4 Design of the Module System and Inheritance Structure

Figure 1 depicts a skeleton of the module system and the mapping of the layered structure as described in Section 3 onto the module system. To keep the figure simple some details including the modules related to the gadgets of Oberon System 3 (see Section 5.1) have been omitted. Figure 2 visualizes the inheritance relations between the key structures CANConfiguration and CANMessage by means of Venn-diagrams. A particularly flexible implementation of structures as in Figure 1 and Figure 2 makes use of language constructs and/or library support including type extension, dynamic types, messages and message handlers, and up-calls or, alternatively, methods of creating objects at run time of dynamically defined types. We shall not enter into these details in the present paper.

We proceed with a description of the structure of Layers 0 to 3; some more details will be explained in Section 5.1.

4.1 Layer 0 – Basic Structure

The basic attributes describing a CAN configuration and the corresponding CAN message classes form part of Layer 0. A configuration consists of a list of messages each of which is characterized by its priority and its name. Within this layer, a configuration can be regarded as a structure with the sole property of being a "container" of this kind.

4.2 Layer 1 – Deterministic Model (Worst-Case-Analysis)

Layer 1 refines the description of Layer 0 by adding to the CAN message descriptor other typical fields such as: number of data bytes s, period T, jitter J, deadline D, queuing delay (interference time) I, and response time R. The configuration descriptor is extended with the configuration's bit time τ_{bit} and a flag indicating which of the variants *standard CAN* or *extended CAN* is being used. These quantities are coupled by a set of equations which can be solved numerically. These equations also form part of Layer 1.

4.3 Layer 2 – Probabilistic Error Model

Layer 2 models a situation where external noise generates error messages on the CAN bus. This situation has to be described by a probabilistic error model. The configuration descriptor remains unchanged; the CAN message descriptor has to be extended with several additional fields: the maximal number of errors keeping the response time just below the deadline D, the corresponding response time R_{max}, and the probability that the number of error messages is equal or below this maximum. To calculate these one has to accept some specific distribution functions related to error events, such as the distribution of "interarrival times" of error events, the nature of errors (single errors or error bursts) and—in the case of error bursts—of the burst size. Mathematically, the problem consists in calculating a convolution power; this requires considerable computational efforts if the exponent is large, i.e. if the number of errors keeping the response time just below the deadline is large. Obviously this is the case if the bit time is small.

4.4 Layer 3 – Model Incorporating a Quality Measure of Timeliness

Finally, the model of Layer 3 introduces the notion of a cost function: a CAN system which forms a vital part of a larger system might trigger an event disastrous to this system if timing constraints are violated. The resulting expected costs can be taken as a quality measure of timeliness of the CAN system with low costs representing good real-time behaviour, of course. This approach

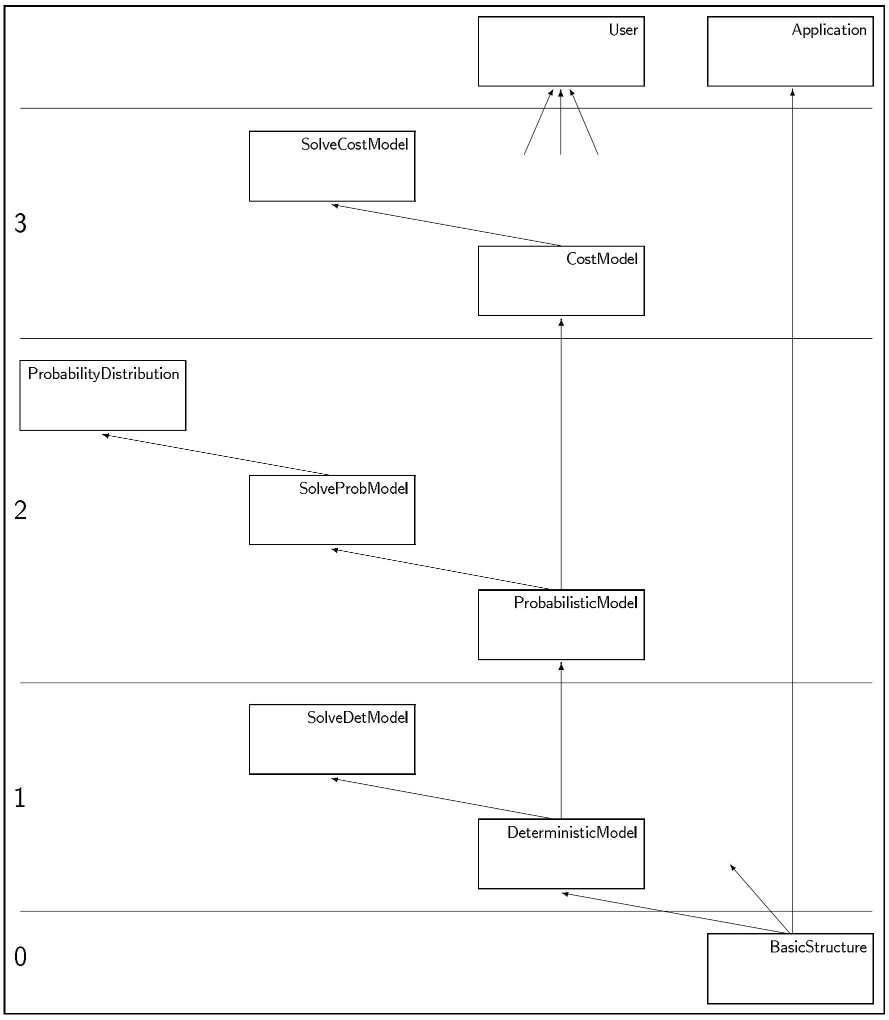


Figure 1 Import relations of basic module structure

is particularly adequate in all situations in which one has to deal with probabilities and where consequently absolute thresholds cannot be guaranteed. The field *expected costs* forms another attribute of the configuration descriptor; the user-defined cost related to each of the message classes has to be added to the message descriptors.

In [Rü98] some case studies based on data previously published in the literature have been discussed in detail (SAE and PSA "benchmarks"). In particular, it has been shown that the concept of a cost function can be useful for comparisons of configurations with different parametrizations. The approach can as well be useful in studying the behaviour of a system near the critical bit time.

5 Implementation Aspects

5.1 Prototypical realization based on Oberon

The essential feature of the CAN Tool presented in this paper is its extensibility. The user can add new CAN model layers to the tool without changing the existing code but simply by extending the tool with new modules. The visualization of the program's output can be achieved by means of standard gadgets which form part of the Oberon System 3. Since it is possible to build GUIs by combining gadgets with a few point-and-click

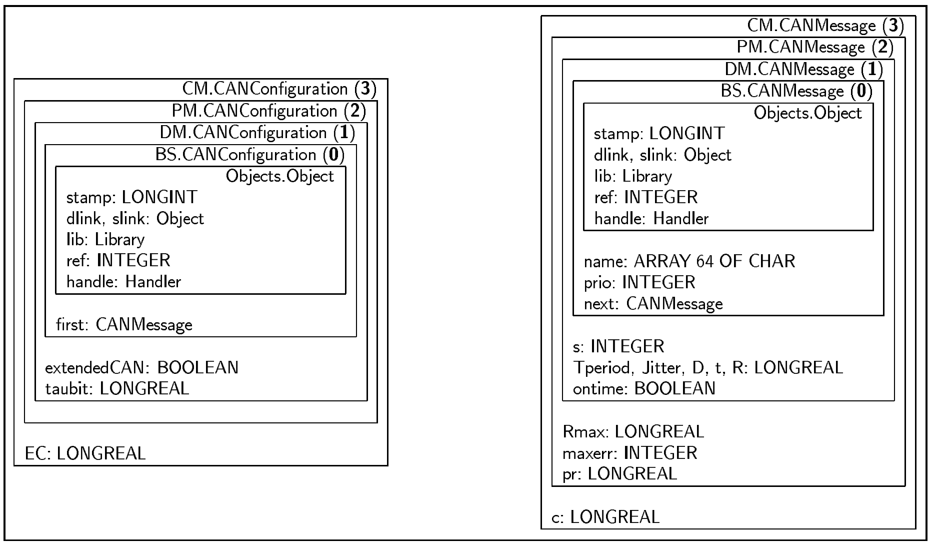


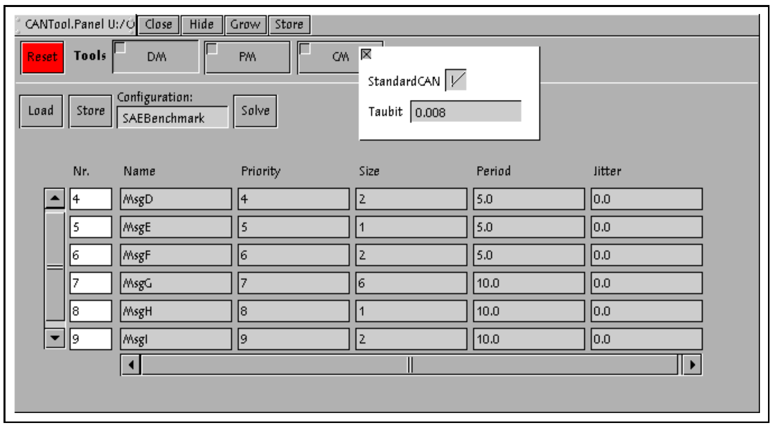
Figure 2 Inheritance structure of CANConfiguration (left part) and CANMessage

actions only, no programming is required in general to visualize the data of a model layer. Although a rich variety of gadgets exists in the Oberon System 3 a pre-fabricated table gadget specifically adapted to visualize output data of the CAN Tool could simplify the construction of a GUI. Therefore, we have designed a new gadget which meets these requirements.

This gadget displays the model layer data in a table with lines representing the data of different CAN messages in a CAN configuration and columns representing the data of different attributes of the CAN messages. One special feature of this table is its genericity. No static elements such as the number of lines or the number and the names of the columns appear explicitly in the code. Instead, the table obtains this information from the respective model layer being used.

Before we proceed with an explanation of the implementation of the table gadget we shall take a look at the GUI in Figure 3. This GUI contains different visual objects which enables the user to communicate with the underlying code. In Oberon these objects are called *visual gadgets*. In the first row there are three buttons for selecting one of the given model layers. If the user wants to add a new model layer he has to add a new button and edit its properties; this can easily be done with the *Columbus* tool which is part of Oberon System 3. Below there are another two buttons which allow the user to save and load data of a CAN configuration, i.e. a set of CAN messages with their proper attributes, and a third button for activating the solution algorithms. All of these visual gadgets are static components. The main part of the GUI is reserved for the table gadget where the CAN messages can be edited and the calculated values are presented. The number of attributes in a CAN message depends on the selected CAN model layer; therefore the table gadget must be non-static. When the user presses one of the top most buttons the respective model layer is linked to the table gadget, which adjusts itself appropriately, i.e., the number and names of the columns are set to the number and names of the attributes, respectively.

The implementation is based on the concept of visual-model-gadgets which is one of the crucial ideas in Oberon System 3. Model gadgets can be considered as a container where different kinds of data are stored and visual gadgets are objects which can visualize this data. So, there has to be a link between the visual and the model gadget, but this connection should be as general as possible. Then, for example, we can use a table for displaying CAN messages or any other data



Configuration: SAEBenchmark

StandardCAN ✓

Taubit 0.008

Nr.	Name	Priority	Size	Period	Jitter
4	MsgD	4	2	5.0	0.0
5	MsgE	5	1	5.0	0.0
6	MsgF	6	2	5.0	0.0
7	MsgG	7	6	10.0	0.0
8	MsgH	8	1	10.0	0.0
9	MsgI	9	2	10.0	0.0

Figure 3 Graphical user interface of CAN Tool

that can be represented in a table. This depends on the model gadget which is linked to the visual gadget and which, of course, does not know that its data is written to a table.

In Oberon System 3 messages are used to connect objects to each other.[2] Every gadget is derived from the class Object which owns a method named handler. This method interprets the known messages and delegates the task of interpreting unknown messages to another handler, typically the handler of the superclass. Many predefined messages for standard tasks exist, e.g., to transfer keyboard input or to inform a visual gadget that it should draw itself to the screen. Of course, many of these messages are used in the table gadget, but the only message relevant to the user, who wants to add a new model layer, is TableMsg.

As usual in the Oberon System 3 the visual gadgets take the active part of the communication. They inform the model gadgets that data has to be read or written. This is the task of the TableMsg. When the user enters data into the table, the table sends a TableMsg to the model gadget; this message contains the input data and the destination where the data shall be written to. Similarly, when the table updates itself, the message contains the address of the data to be read. To address a special destination the lines of the table are numbered and the columns are represented by a unique name. This, of course, reflects the use of the table in the CAN Tool since the columns represent the attributes of the CAN messages which are identified by unique names. Reading these names from the model gadget is another important function of the TableMsg.

More details of the (visual) table gadget are not relevant to the user. If he wishes to add a new model layer this has to be implemented in an appropriate model gadget which, among other things, has to react to the TableMsg. First of all there exists a base class for all model gadgets that should be displayed in the table gadget. Its name is Table and it is implemented in a module named TableIO. All table model gadgets must be extensions thereof.

Besides class Table another two classes exist that form a base for all model gadgets that are to represent a CAN model layer. They can be found in module BasicStructure. Their names are CANConfiguration which is derived from Table and CANMessage which is derived from the base class of all model gadgets. Since each model layer is implemented in separate modules, extensions of theses classes can be named equally to ease readability of the code. CANMessage manages the data for one CAN message. Here, the user has to add variables for new attributes and adjust the handler. CANConfiguration manages a num-

[2]The collision of terms in "Oberon messages" and "CAN messages" is unfortunate. To avoid a misunderstanding we shall use these explicitly where required.

ber of CANMessage objects and stores global data of a configuration.

The kind of modification mentioned above can easily be achieved by re-using code of a given model layer. Only a few lines of code have to be added or changed.

5.2 Prototypical realization based on Java

At the time of writing this paper another variant of the CAN Tool is being implemented using the language Java. Although, as discussed in previous sections, an Oberon-based implementation of the CAN Tool has many attractive features Java has gained an enormous importance during the last few years. One of the reasons is that Java-written applications can be shared across the internet. Therefore, we have decided to implement another variant of the tool using the language Java. In passing it should be noted that, inspite of the obvious syntactical similarities between Java and C++, there are several structural relationships and historical connections between Java and Oberon, see [Fra98] for a review.

In principle, the module structure of the Oberon implementation can be transferred to the Java version. Here, the modules are represented by packages and the extensible records are replaced by Java classes. Since the tool does not have to fit into a gadget environment as in Oberon, no message handlers are used.

Essentially, the addition of a new CAN model requires a similar sequence of steps as in the Oberon variant. However, there are substantial differences between Oberon System 3 and Java in a standard environment with respect to creating GUIs. In Java, the user has to inform the applet about the CAN model being used, which is identified by a unique name and make the corresponding code available. This information is handled by parameters inside of the `html` code where the applet is included.

From a user's point of view, the GUIs of the Java and the Oberon variants have essentially the same components: there are buttons for selecting a CAN model, a table where the different CAN message data can be entered and a button for activating the appropriate solution method of the model. Since reading and writing data to an external medium (e.g. a hard disk) is a critical action for an applet, we have not included *store* and *load* buttons in the GUI in the present implementation.

6 Conclusions

This paper presents two variants of an experimental prototype of a tool supporting timing analyses of CAN systems. The architecture of this tool relies on some novel ideas of software technology as conceived and realized during the past decade in a particularly clear and simple way in the Oberon project. One of the variants utilizes the gadgets of the Oberon System 3, the other one is, at the time of writing this contribution, being implemented in the language Java.

Although, presently, there seems to be no widespread industrial or commercial use neither of the language nor the operating system Oberon the Oberon-based variant of the tool might nevertheless be useful for practitioners who do not wish to learn any details of handling the Oberon system: at system startup an application can be activated—similarly to most other operating systems—without knowing any Oberon commands. The real advantage of Oberon is its extreme compactness which is ideally suited for small computers: the compiled code consists of less than 3 MB (a full-size Oberon System 3 including the gadgets subsystem) although the functionality is—at least—comparable to other operating systems.[3]

Several methods exist to analyze the timing behaviour of CAN systems, for a detailed summary see [Rü98]. It would be desirable to extend the tool by adding some of these, e. g. simulation methods and numerical treatment of analytical solutions of priority queues of queuing theory.

[3]A corresponding value of the original, also fully graphics-based Oberon system is mentioned by Reiser in his book [Rei91, p. 13]: it is 150 kB!

References

[FM98] A. Fischer and H. Marais. *The Oberon Companion: A Guide to Using and Programming Oberon System 3.* vdf Hochschulverlag AG, 1998.

[Fra94] M. Franz. Extensible Programming: ein neues Paradigma für die Softwareentwicklung. In *Informatik-Fachtagung Moderne Programmierparadigmen.* FH Braunschweig/Wolfenbüttel, Fachbereich Informatik, 6.-7. Oktober 1994.

[Fra98] M. Franz. Java – Anmerkungen eines Wirth-Schülers. *Informatik Spektrum*, 21(01):23–26, 1998.

[Gut94] J. Gutknecht. Oberon system 3: Vision of a future software technology. *Software — Concepts and Tools*, 15:26–33, 1994.

[Gut96] J. Gutknecht. Oberon, gadgets, and some archetypal aspects of persistent objects. *Information Sciences*, 93(01):65–86, 1996.

[Law97] W. Lawrenz, editor. *CAN System Engineering. From Theory to Practical Applications.* Springer Verlag New York, 1997.

[NS97] N. Navet and Y.-Q. Song. Performance and fault tolerance of real-time applications distributed over CAN (Controller Area Network). *CiA Research Award 1997*, 1997.

[NSS99] N. Navet, Y.-Q. Song, and F. Simonot. Worst-case deadline failure probability in real-time applications distributed over CAN (Controller Area Network). *Journal of Systems Architecture*, 1999. to appear.

[Rei91] M. Reiser. *The Oberon System. User Guide and Programmer's Manual.* Addison-Wesley, 1991.

[Rü96] R. Rüdiger. Programmierung Erweiterbarer Systeme. Die Oberon-Sicht. In *Technische Berichte Nr. 29. 1. Norddeutsches Kolloquium über Informatik an Fachhochschulen (26.-27. April).* FH Hamburg, Fachbereich Elektrotechnik und Informatik, Juli 1996. href=http://www.fh-wolfenbuettel.de/fb/i/organisation/ personal/ruediger/forschung/fh_hh.ps.

[Rü98] R. Rüdiger. Evaluating the temporal behaviour of CAN based systems by means of a cost functional. In *5th International CAN Conference (iCC'98), San José, CA, USA*, pages 10.09–10.26, November 1998. href=http://www.fh-wolfenbuettel.de/fb/i/organisation/personal/ ruediger/forschung/icc98.ps.

[RW92] M. Reiser and N. Wirth. *Programming in Oberon. Steps beyond Pascal and Modula-2.* Addison-Wesley, 1992.

[Szy96] C. Szyperski. Independently extensible systems – software engineering potential and challenges. In *Proceedings of the 19th Australasian Computer Science Conference*, 1996.

[TB94] K. Tindell and A. Burns. Guaranteeing Message Latencies on Control Area Network (CAN). In *1st International CAN Conference (iCC'94), Mainz*, 1994.

[TBW94] K. Tindell, A. Burns, and A. Wellings. Calculating Controller Area Network (CAN) Message Response Times. In *IFAC workshop on Distributed Computer Control Systems, Toledo, Spain*, September 1994.

[TBW95] K. Tindell, A. Burns, and A.J. Wellings. Calculating Controller Area Network (CAN) Message Response Times. *Control Engineering Practice*, 3(8):1163–1169, 1995.

[THW94] K. Tindell, H. Hansson, and A.J. Welling. Analysing Real-Time Communications: Controller Area Network (CAN). In *Real-Time Systems Symposium, Puerto Rico*, Dezember 1994.

[WG92] N. Wirth and J. Gutknecht. *Project Oberon. The Design of an Operating System and Compiler.* Addison-Wesley, 1992.

[WLHS92] Z. Wang, H. Lu, G. Hedrik, and M. Stone. Message delay analysis for CAN based network. In *Proc. of 1992 ACM SIGAPP symposium on applied computing, Kansas City, MO (USA)*, March 1992.

Company:	Institute for Distributed Systems
	University of Applied Sciences (FH) Braunschweig/Wolfenbüttel
Address:	Salzdahlumer Str. 46/48,
	D-38302 Wolfenbüttel, Germany
Phone:	+49 5331 939 689
	+49 5331 939 649
Fax:	+49 5331 939 602
Email:	{M.Weseloh | R.Ruediger}@FH-Wolfenbuettel.DE
Homepage:	http://www.fh-wolfenbuettel.de/fb/i/organisation/personal/ruediger/index.html