Dagstuhl Seminar
on
High Integrity Programmable Electronic Systems [1]

William John Cullyer, University of Warwick
Wolfgang A. Halang, FernUniversität Hagen
Bernd J. Krämer, FernUniversität Hagen

Schloß Dagstuhl, February 27 – March 3, 1995

[1]The organisers of this seminar are grateful to the Society of Friends and Supporters of the FernUniversität for the financial support covering part of the travel expenditure of participants from Middle- and Eastern European countries.

Contents

Introduction 6
Final Seminar Programme 7

Abstracts of Presentation:
F. Saglietti, Decision Rules Supporting the Design of Fault-Tolerant Software 9
A. Berztiss, Unforeseen Hazard Conditions and Software Cliches 10
H. Wedde, Incremental Experimentation: A Methodology for Designing and Analyzing Distributed Safety-Critical Systems 11
B.A. Wichmann, Establishing a Single Market for Safety Critical Software 12
M. Pezzè, A Formal Approach to the Development of High Integrity Programmable Electronic Systems 12
E. Pofahl, Methods Used for Inspecting Safety Relevant Software 13
M. Heisel, An Approach to Develop Provably Safe Software 14
L. Barroca, Application of Formal Methods: an Architectural Approach 16
H. Langmaack, The ProCoS-Way Towards Correct Systems 17
R. Baber, The Suitability of Various Notational Forms and Languages for Specifications 19
N. Völker, Reasoning about Simple Reactive Programs 19
F.W. von Henke, Formal Methods between Research and Practice 20
R. Baber, Correct by Design: Simple, Practical, Mathematically Rigorous Program Development, an Example 20
W. Ehrenberger, Human Understanding and Program Complexity 21
W. Dzida/C. Hoffmann, Designing for Risk Mangement 22
G.H. Schildt, Safety Critical Application of Fuzzy Control 23
R. Hampel/N. Chaker, Application of Fuzzy-Control in Safety Related Systems for Process- and Energy-Technology 23
D. Hutter/B. Langenstein/C. Sengler/J. Siekmann/W. Stephan/A. Wolpers, Verification Support Environment (VSE) 24
C. Bron, Exceptions and (Real-Time) Control Systems 25
D. Weber-Wulff, Mechanical Verification with the Boyer-Moore Prover: Usage and Problems 27
V. Stavridou, Formal Methods for Safety Critical Systems 29
W.J. Cullyer, The Use of Ada for Emulation of Formal Specifications . . 30
G. Heiner, Approaches to Safety Engineering of PLCs 30
W.A. Halang, A Safety Licensable PES 31
L. Motus, Specification of Safety and Reliability Requirements for Control Systems 32
L. Trybus/M. Śnieżek/Z. Świder, Redundant Controller Triggered with a

Communication Channel 33
M. Goedicke, Designing Software Systems for High Integrity 33
R. Tol, Formal Design of a Real-Time Operating System Kernel 35
B. Krämer, Concurrency and Distribution in Safety-Critical Systems ... 35

List of Participants

Introduction

Software is increasingly being used in safety-critical applications where failure could cause loss of human life, personal injury, or significant material damage. High integrity programmable systems denote a class of software controled applications that are characterized by a sensible interplay of heterogeneous technologies (software and various forms of hardware), high requirements on the dependability of all system components, including the safety, security, adequacy and correctness of the embedded software, and – depending on national regulations – the need to undergo extensive certification procedures. Examples of high integrity applications occur in process control (e.g., in chemical industry or nuclear power generation), traffic control, or in medical systems.

High integrity programmable electronic systems for safety critical control and regulation applications form a new field that stands at the very beginning of its treatment in research, development, and teaching. The significance of this subject arises from a growing awareness for safety in our society, on the one hand, and from the technological trend towards more flexible, i.e., program controlled, technical devices, on the other hand. A major objective is to reach the state that such systems can be constructed with a sufficient degree of confidence in their dependability that enables their licensing for safety critical control and regulation tasks by the pertaining authorities on the basis of formal approvals. But authorities are currently still very reluctant in approving safety related systems whose behaviour is exclusively program controled, leading to the unsatisfactory situation that safety licensing, in general, is still denied for highly safety critical systems relying on software with non-trivial complexity. The reasons lie mainly in a lack of confidence in complex software systems and in the high effort needed for their safety validation following current practices. Although formal specification and verification techniques are increasingly accepted as an important approach to achieve high integrity software, their use in practice is still limited due to the lack of effective tools and the need for special expertise.

In this context, the seminar aimed at the evaluation and comparison, of existing, more or less, formal methods with respect to their use in practice and indicating directions for future development. The seminar thereby spanned several dimensions of computer and computing science including safety and fault tolerance strategies, formal methods, languages with high integrity features, human factors in risk reduction and program understanding, software verification, safety-oriented software architectures and operating system kernels, and hardware correctness. These dimensions were supplemented with application experiences of licensing authorities and were confronted with particular requirements and characteristics of the application domain such as

fuzzy-ness, distribution, or predictability and timeliness of behaviour.

Hagen, May 1995 B.J. Krämer

Final Seminar Programme

Monday, February 27

09.00-09.40 B. Krämer, Introduction
09.40-10.30 F. Saglietti, Decision Rules Supporting the Design of Fault-Tolerant Software
11.00-11.40 A. Berztiss, Unforeseen Hazard Conditions and Software Cliches
13.30-14.30 H. Wedde, Incremental Experimentation: A Methodology for Designing and Analyzing Distributed Safety-Critical Systems
14.30-15.20 B.A. Wichmann, Establishing a Single Market for Safety Critical Software
15.40-16.30 M. Pezzè, A Formal Approach to the Development of High Integrity Programmable Electronic Systems
16.30-17.50 E. Pofahl, Methods Used for Inspecting Safety Relevant Software
19.30-20.20 M. Pezzè, Demonstration of Petri Net Tool Cabernet

Tuesday, February 28

09.00-09.40 M. Heisel, An Approach to Develop Provably Safe Software
09.40-10.30 L. Barroca, Application of Formal Methods: an Architectural Approach
10.45-11.30 H. Langmaack, The ProCoS-Way Towards Correct Systems
11.30-12.00 R. Baber, The Suitability of Various Notational Forms and Languages for Specifications
14.00-14.40 N. Völker, Reasoning about Simple Reactive Programs
14.40-15.20 F.W. von Henke, Formal Methods between Research and Practice
15.50-16.40 R. Baber, Correct by Design: Simple, Practical, Mathematically Rigorous Program Development, an Example
16.40-17.20 W. Ehrenberger, Human Understanding and Program Complexity
17.20-18.00 W. Dzida, Designing for Risk Mangement
19.30-21.00 Evening Session on Testing and Standards

Wednesday, March 1

09.00-10.00 G.H. Schildt, Safety Critical Application of Fuzzy Control
10.00-10.40 R. Hampel, Application of Fuzzy-Control in Safety Related Systems for Process- and Energy-Technology
11.10-11.50 J. Siekmann/W. Stephan, Verification Support Environment (VSE)

Thursday, March 2

09.00-09.30 C. Bron, Exceptions and (Real-Time) Control Systems
09.30-10.15 D. Weber-Wulff, Mechanical Verification with the Boyer-Moore Prover: Usage and Problems
10.45-11.30 V. Stavridou, Formal Methods for Safety Critical Systems
11.30-12.15 W.J. Cullyer, The Use of Ada for Emulation of Formal Specifications
14.00-14.40 G. Heiner, Approaches to Safety Engineering of PLCs
14.40-15.20 W.A. Halang/H.-P. Meske, A Safety Licensable PES
15.20-16.00 B.A. Wichmann, Establishing a Single Market for Safety Critical Software
16.30-17.15 L. Motus, Specification of Safety and Reliability Requirements for Control Systems
17.15-18.00 L. Trybus, Redundant Controller Triggered with a Coomunication Channel
19.30-21.00 D. Weber-Wulff, Demonstration of Boyer-Moore Applied to Compiler Verification

Friday, March 3

09.00-09.40 M. Goedicke, Designing Software Systems for High Integrity
09.40-10.30 R. Tol, Formal Design of a Real-Time Operating System Kernel
11.00-11.30 B. Krämer, Concurrency and Distribution in Safety-Critical Systems
11.30-11.40 W.J. Cullyer, Risk Areas 1995-2000

Abstracts of Presentation

Decision Rules Supporting the Design of Fault-Tolerant Software

Francesca Saglietti, Institute for Safety Technology (ISTec) GmbH, Garching

Approaches to improve and evaluate software reliability are still among the most crucial topics in today's software engineering. One of the strategies to prevent critical failures in spite of unavoidable faults aims at tolerating them using built-in redundancy of information and data processing. Such a fault-handling policy is generally known as software diversity. Software diversity essentially consists in developing more program variants intended to fulfill the same (or an equivalent task); their results are subjected to an adjudication mechanism, which determines the acceptable one(s) on the basis of a (relative or absolute) testing procedure. The application of diversity to achieve software fault tolerance has proved to be an efficient way to increase software reliability. Experimental evidence has shown it to partly complement (but in general not replace) the more conventional constructive strategies for fault avoidance and analytical techniques for fault detection. This makes software diversity particu
larly suitable for safety-related applications requiring an ultrahigh reliability degree.

Evidently, the achievement of fault tolerance strongly depends on the level of dissimilarity among the alternative variants. Theoretical studies conducted in this area and supported by experimental observations confirm the effectiveness of increasing dissimilarity among diverse variants by intentional intervention in the development process, in other words, by "enforcing" diversity in a deterministic way. As any forced degree of diversity requires additional effort to define the necessary design parameters, one of the most essential decisions to be taken in this context concerns the most promising diversity level(s) to be enforced: this leads to a trade-off between the cost required to introduce and enforce diversity at a given development level and the expected fault tolerance capability.

The intention of this contribution is to provide a systematic method to support rational decision-making during the design of fault-tolerant software. It concerns both qualitative and quantitative considerations on software diversity, and assists in deciding whether to apply diversity and which fault tolerance technique to select. This is done by analysing software diversity in terms of fault types tolerated and of cost efficiency.

The first part of the work presents a distinction of faults in terms of several attributes: a.o. their origin, manifestation continuity, multiple occurrence, effect and environment. For each fault type identified the dissimilarity policy most promising for its tolerance is proposed; examples or references on particular dissimilar methodologies are meant to provide support for the method practicality. The second part analyses cost efficiency of two-fold diverse software systems when compared with non-redundant ones. This is done by modelling the underlying V&V process in terms of required target, available effort and expected fault tolerance achievement. The study is concluded with an optimizing strategy supporting decision-making in favour of, or against diversity during the earlier development phases.

Unforseen Hazard Conditions and Software Cliches

Alfs T. Berztiss, University of Pittsburgh, University of Stockholm

Earlier a research agenda was proposed on issues that relate to safety-critical software. This discussion of safety was based on the following topics: completeness of requirements, readable representation languages for specifications, tools for the validation of specifications, validation of the responsiveness of a system, verification of an implementation, robustness of control software, effect of common-cause failures, reuse of reliable software, the development process for safety-critical software, and ethical issues. Here safety-critical software is looked at from a different perspective — how is software to cope with an unforeseen condition that constitutes a hazard? An unforeseen or unusual condition (UC) arises when the software requirements fail to consider some system states, particularly states into which the system finds itself on account of malfunctioning of the environment in which the software is embedded, or the malfunctioning of the hardware-software interface. After a review of the research agenda, we outline a two-level architecture for safety-critical software. Under this, a primary software system performs the control functions normally required of safety-critical software. A secondary system independently monitors the total controlled-controlling system. This secondary system is developed using design cliches selected from a safety library. Each design cliche is to cope with a specific kind of unforeseen or unusual condition, e.g., a meter reading that remains unchanged over a period of time. The use of the specification language SF is advocated for defining the design cliches. In this a UC is dealt with by means of a three-stage process: recognition, analysis, and correction. Examples of this

process are given for several types of UCs. We suggest that a study group be formed that is to assemble a library of safety cliches.

Incremental Experimentaton: A Methodology for Designing and Analyzing Distributed Safety-Critical Systems

Horst F. Wedde, Jon A. Lind, Andreas Eiss, Universität Dortmund

Safety-critical systems are distributed *hard* real-time systems, i.e. tasks may become critical in the sense that at such a point of time its deadline *has* to be met, otherwise disastrous consequences for the whole system have to be confronted. In our own research we have pioneered concepts and directions for a comprehensive treatment of this matter. In addition to hard real-time requirements, safety-critical systems have to satisfy *dependability, reliability,* and *fault tolerance requirements,* and they typically operate in environments with a considerable amount of unpredicatable events. (One form of unpredictability results from the autonomy of the subsystems, i.e. from their lack of information about the next actions of the other subsystems.)
Measures to meet dependability or safety requirements are mostly in conflict with those for achieving hard real-time responsiveness: Whatever policy would be chosen to improve the system performance with respect to the one class of requirements, would have to be paid for by a performance decrease regarding the other class of criteria. As a consequence a comprehensive formal design (or a *closed-form solution)* which covers both classes of requirements is not conceivable and a comprehensive design of distributed safety-critical systems can in principle be done only through *heuristic* approaches. Moreover, both the design strategies and the system behavior have to be *adaptive.*
A heuristic methodology for systematically dealing with this problem, termed **Incremental Experimentation**, is described. Roughly speaking a system design would start from a "simple" model of the final system to be built. This model would undergo a systematic validation and evaluation resulting in an *educated* model refinement and extension. Further iterations would eventually allow us to come up with a realistic system prototype. The technical details of Incremental Experimentation are developed in the context of the MELODY project which has gone through five distinctive stages of development which lead us very close to a target prototype.

Establishing a Single Market for Safety Critical Software

B.A. Wichmann, National Physical Laboratory, Teddington

Currently, the market for safety critical software within the European Union is divided by industrial sector and country. If a single standard for such software could be agreed to which objective assessments could be made, then the sector and country boundaries could be removed or reduced.
The paper surveys some existing standards and concludes that objective assessment would be possible to the civil avionics standard DO-178B. However, this standard does not have consensus support in other sectors and hence it seems unlikely that the existing barriers to a single market will be reduced in the short term.

A Formal Approach to the Development of High Integrity Programmable Electronic Systems

Mauro Pezzè, Politecnico di Milano

Human lives and large economic interests rely on the correct behavior of high integrity programmable electronic systems. The high costs of failures of such systems require sophisticated techniques for validating correctness properties. Development techniques successfully used in other application areas do not always adapt to the concurrent and real-time characteristics of such systems and do not support adequate validation techniques. Formal methods have been frequently advocated as providing a potential solution to meeting the high reliability standards required in this application domain, preventing ambiguities in specifications, and supporting powerful semantic checks non applicable to informal and semiformal specifications. Despite their advantages, formal techniques are seldom applied in practice due to the lack of powerful composition and decomposition mechanisms, the lack of flexibility, the complexity of the analysis techniques, the gap between formal specifications and final implementation, the absence of adequate tool support. The recent solutions proposed to overcome the main limitations of formal techniques did not succeed yet in assessing the use of formal techniques in significant industrial sectors. Although part of the insuccess of introducing formal techniques in industry is due to non-technical reasons (e.g., trade-off

between advantages and costs and reluctance to face the risks of introducing new techniques in the development process) part of the insuccess is still due to incompleteness of the results and limited integration of different studies in a common framework.
In the last years we studied solutions to all the above mentioned problems in a coherent framework and we experimented their validity with industrial case-studies using a prototype of an integrated environment for the development of safety critical real-time software. The proposed solution is based on high-level timed Petri nets, and comprises techniques for time reachability analysis, hierarchical decomposition rule that preserve safety and temporal properties, a formal framework for adapting Petri nets to the preferred end-user operational notation, and design notations compatible with the specification framework.

Methods Used for Inspecting Safety Relevant Software

Ekkehard Pofahl, TÜV Rheinland, Köln

The technical supervisory agencies TÜV (Technische Überwachungs Vereine) in Germany inspect software in many different applications. Typical applications range from software controlling railway switches and operating systems for PLCs (Programmable Logic Controllers) to software in microcontrollers for furnaces and lightgrids. Also commercially used software, where the focus is more in the field of user friendlyness than safety, is inspected by TÜV.
There are a few methodologies for inspecting software in safety relevant areas. One of the most work intensive in the method of "diverse backtranslation". This method uses the binary code of a software to reconstruct from it the specification. The several steps from binary code to the final specification are supported by a set of tools (editors, compilers, discompilers, etc.).
Another typical method for validation and verification of software is analysis and test. The first step of this method is to proof the validity of the software specification. After this step static analysis tools are used to proof that the specification is indeed implemented in the software. The results from the analysis of the specification and the results from the static analysis of the source code are used to dynamically check the software by means of white, or gray box testing. Special attention it put to diagnostic and fault handling routines. Some tests are also derived solely from the specification (black box test).

The inspections are done according to several national and international standards. The most important standard is the DIN V 19250, Fundamental Safety Aspects To Be Considered For Measurement And Control Protective Equipment (Grundlegende Sicherheitsbetrachtungen fur MSR-Schutzeinrichtungen) and DIN V VDE 0801 Principles For Computers In Safety Related Systems (Grundsatze fur Rechner in Systemen mit Sicherheitsaufgaben).

An Approach to Develop Provably Safe Software

Maritta Heisel, Technische Universität Berlin

Overview. A procedure to develop provably safe software is presented. It is based on well-established tools and techniques to set up formal specifications in the specification language Z and the program synthesis system IOSS (Integrated Open Synthesis System) designed by the author. IOSS allows developers to implement Z specifications in a provably correct way. Each step is described in detail and complemented by some proof obligations that have to be met if safety is to be guaranteed. As an example, a microwave oven is considered. Carrying out the procedure for this example gives rise to discuss the relation of software safety vs. correctness and availability. We discuss how the formal part of the development can be confined to safety-critical requirements, in order to make the approach applicable to larger systems.

Steps and Proof Obligations. The steps to be performed and the corresponding proof obligations are shown in the following table.
The first three steps give a guideline how to set up the specification of a system. In general it will not be possible to carry out these steps independently of each other and without iteration. Instead, a process resembling the spiral model of software development will have to be employed. The last three steps describe how to perform the transition from a mere specification to a totally correct program.
Step 3 is specific to the development of safety-critical systems. In this step the sensors must be modeled that enable the system to detect situations to which it must react. It must also be specified *how* the system reacts to the possible sensor values and/or failures. To prove that the system reacts in a deterministic way is not strictly necessary. This requirement is only stated because deterministic behaviors can be better analyzed than nondeterministic ones.

In Step 5, the Z specifications are transformed into so-called *programming problems*, the input format for IOSS. A programming problem basically consists of a precondition and a postcondition which is divided into an invariant and a goal. Moreover, it is specified which part of the state may be changed. IOSS supports synthesis of imperative programs in a partially automated way (Step 6). For each developed program, a correctness proof is generated. IOSS fits well with Z because both explicitly deal with states.

No.	Step	Proof Obligations
1	Define the legal states of the system.	Show that the initial state is legal.
2	Define the actions the system can perform.	Analyze the conditions under which the actions transform legal states into legal states.
3	Define the interface of the system with the outside world.	Show that the internal system operations are only invoked if their preconditions are satisfied. Show that for each combination of sensor values exactly one internal operation is invoked. Show that – if the sensors work correctly – the system faithfully represents the state of its environment.
4	Refine the data of the specification so that data structures of the target programming language can be used.	Show the correctness of the refinement.
5	Transform the specification obtained in Step 5 into a form suitable for the program synthesis system.	Show the correctness of the algorithm performing this task.
6	Use the synthesis system to obtain a proven correct implementation of the specified system.	Proof obligations are generated by the synthesis system.

Safety vs. Correctness. One might consider safety a weaker requirement than correctness. The example of the microwave oven, however, shows a different picture. Its safety requirements are that (i) the microwave is always switched off when the door is open, and (ii) the light is on when the microwave is switched on. To ensure safety, there should be an "emergency shutdown" that switches off the microwave as soon as a failure of the door

sensor is detected. This situation is not taken into account when only the correctness of the software is of interest because correctness is a relation solely between a specification and a program. Hardware failures are of no interest in correctness considerations. Hence, we think that the development of safe software has to proceed differently: hardware failures must explicitly be taken care of by the software.

Safety vs. Availability. The example shows that availability and safety can be conflicting goals. Of the safety requirements stated above, (i) is certainly more important than (ii). If the light bulb breaks down, a reasonable decision might be not to invoke the "emergency shutdown" but to sacrifice the less important safety requirement to increase availability of the oven.

Reducing Formal Verification Work. For devices like a microwave oven, a complete formal treatment certainly can be recommended because the control software is relatively simple. The cost for a formal safety proof should be much less than potential damages. For larger systems, however, a complete formal treatment might not be feasible. In this case, our approach can be applied nevertheless. It is possible to formalize and prove only selected properties of the system and treat the other requirements with traditional techniques.

Limitations of the Approach. The approach outlined above concentrates on the software aspects of safety-critical systems. Nothing can be guaranteed about the correct functioning of the hardware. For instance, if the sensors yield false values, the system can enter a non-safe state because the software controls the system according to the sensor values. This limitation cannot be overcome by means concerning the software alone.
Moreover, it is not possible to deal with absolute time measures in the formalisms we have chosen. If it is, e.g., necessary that a component reacts within 2 ms, then this cannot be guaranteed with our approach. The maximum execution time of the specified operations cannot be specified in Z, and we are not aware of any formal methods that allow one to *prove* maximum execution time of programs[2].
As a result, the kind of safety our approach can guarantee is relative. Since we can only guarantee that the states before and after execution of an operation are safe, this execution must be sufficiently fast, because in the intermediate states that occur during execution, safety cannot be guaranteed. It is up to

[2]This is true even for formalisms designed to deal with time, like temporal logic or the duration calculus; again, these limitations come from the fact that the formalisms do not consider the hardware on which the programs are executed.

the system designers and implementors to judge if this is the case. Here, traditional methods like testing are indispensable.

Application of Formal Methods: an Architectural Approach

Leonor Barroca, The Open University, Milton Keynes

The application of formal methods to safety-critical systems is still in its first steps. Some cases have been recently reported but there is recognition that some major obstacles have to be overcome. We believe that one of these obstacles is that no single notation exists that can cover the different stages of development and different views of the system. In this talk we discussed the principles behind our approach and proposed a set of formal techniques to represent different aspects of systems and provide the basis for resoning about safety properties.
This set of techniques, proposed under the name of *Architectural Specification Method*, has been put together with the aim of combining the use of accessible techniques with the possibility of formulating rigorous arguments about design decisions and tracing these decisions against requirements. These points are particularly important for the production of safety arguments in the assessment process. The ArchSM uses a graphical notation based on Timed Statecharts and Real Time Logic (RTL) for describing the system's temporal properties, and Z for describing functionality. The system structure is represented by a subset of DORIS Real Time Networks. From as early as possible in the requirements definition the time critical properties are formally stated; a model of the required behaviour (including timing behaviour) is built, and the consistency of the model with the critical timing properties is checked.
The principles that have driven this work are based on supplementing existing good practices with the use of formal specification and concentrating on well understood pieces of the whole system that are critical. We believe that this is the way for an effective use of formal methods in industry.

The ProCoS-Way Towards Correct Systems

Hans Langmaack, Christian-Albrechts-Universität, Kiel

ProCoS is the EU-ESPRIT-BRA-project “Provably Correct Systems”. Participants are U. Oxford (C.A.R. Hoare, coordinator), DTU Lyngby (A.P. Ravn), U. Oldenburg (E.-R. Olderog) and U. Kiel (H. Langmaack). A system as investigated by ProCoS is a technical system with embedded controlling processors, sensors, actuators, channels, timers and physical environment, especially a real time automating system (hybrid system) with explicit parallelism and lower and upper time bounds. Correctness is closely related to safety, security and dependability; if requirement specifications are complete enough the latter properties can be logically implied from correctness. ProCoS goal is to contribute to mathematical foundation for analysis and synthesis and to mathematical principles, techniques and tools for systematic and correct design and construction especially of safety critical systems. For the ProCoS-idea R. Boyer’s and J.S. Moore’s CLInc-stack has been starting model: a translator for Micro Gypsy, an assembler for PITON, a hardware design for the FM8502-processor and an operating system kernel. ProCoS for its own tower has put two more levels onto the stack, requirements and specifications:

1. Requirement Language RL, which is the time interval Duration Logic DL with its Duration Calculus DC.
2. Specification Language SL, bridging the state based RL and the event oriented PL by specifications, each consisting of
   - regular event trace expressions (from CSP)
   - communication specifications (action systems)
   - time constraints.
3. Higher Programming Language PL with parallel processes and lower time bounds as in occam, and newly added upper time bounds and alternatives with combined timer/input guards.
4. Machine Language ML is the code of the transputer, a RISC-processor from INMOS with eight in/out channels connected to other transputers, sensors or actuators.
5. Hardware Description Language HDL, programs of which are netlists for field programmable gate arrays.

As a case study, a gas burner is discussed with capture of both functional and safety requirements R in DL, with natural laws or guaranteed engineers’ experiences (assumptions) A in DL, with architectural design D in DL and implementation

$$D \Rightarrow (A \Rightarrow R)$$

derived by DC-rules (Lyngby) and with specification in SL and program in PL refining D by inference rules

$$(\pi \Rightarrow \Sigma) \wedge (\Sigma \Rightarrow D)$$

(Oldenburg). Software and hardware compilers have been constructed which translate into machine code m in ML with

$$m \Rightarrow \pi$$

(Kiel) and netlists n in HDL with

$$n \Rightarrow \pi$$

(Oxford). We identify programs and their semantic predicates, so semantic brackets are left away from m and n. Compiling specifications have been proved correct; common semantic basis of all correctness proofs for inference rules and compilers is DL.

The Suitability of Various Notational Forms and Languages for Specifications

Robert L. Baber
VDI CEng FBCS, Bad Homburg, Germany

The purposes of a specification in engineering practice are to facilitate communication and reaching agreement as well as to provide the technical basis for a contract or order and for verifying the adequacy of the delivered system, i.e. the fulfillment of the contract. Parties to the discussion and negotiation of a specification include technical and business representatives of the potential supplier, the purchaser and possibly other interested parties.
In this short, provocative group experiment three different forms of a specification for a safety function in a nuclear reactor control application are presented. The lack of agreement among the participants regarding which form for the specification best fulfills the purposes of a specification as outlined above leads to the conclusion that no one form is adequate or sufficient. Different participants in the discussion and negotiation of the specification, because of their different backgrounds, needs, orientation and goals, require different languages and forms of communication.

Reasoning about Simple Reactive Programs

Norbert Völker, FernUniversität Hagen

Machine supported verification and transformation of programs is an important means to increase the reliability of programmable systems. In this talk, we present an approach which is based on the use of higher order logic (HOL) and aims at reactive systems. For the specification of such systems we use HOL extended by linear time operators. Programs are expressed in the synchronous data flow language Lustre. As theorem proving assistant we employ the HOL instantiation of the Isabelle system, a general logical framework. Both the syntax and semantics of Lustre are implemented in Isabelle/HOL. This is made relatively easy by using the Isabelle/HOL facilities for defining datatypes and primitive recursive functions. As an example development, we give a sketch of the correctness proof of a timer element.

Formal Methods between Research and Practice

Friedrich W. von Henke, Universität Ulm

Formal methods have an important potential of playing a crucial role in developing safety-critical systems (both software and hardware). We briefly summarize salient features of the specification and proof systems EHDM and PVS, highlight some major applications of those systems in the field of safety-relevant system designs, and discuss some observations concerning areas in which further research is needed to advance the use of formal methods in realistic contexts. These include in particular aspects of constructing realistic system models, composability and reusability of models, and integration of formal and semi-formal methods.

Correct by Design: Simple, Practical, Mathematically Rigorous Program Development, an Example

Robert L. Baber, VDI CEng FBCS, Bad Homburg

Today's software design errors can be attributed to the same cause as design errors in other technical fields in earlier centuries: the lack of suitable support for practical engineering work based upon scientific principles.
In this paper it is argued that such a scientific foundation exists for the software field and that it can and should be used more widely in practical software development work. Some of the implications of that foundation for designing correct software are described. An example, in which a routine for a generic class written in the object oriented language Eiffel is designed, illustrates one of the many possible ways these concepts can be profitably used to guide the program design process. In particular, the hypotheses of the relevant proof rules are employed as design requirements for the individual parts of the program to be designed. By satisfying these requirements at each step in the design process, the designer ensures that the program will be provably correct by design.

Human Understanding and Program Complexity

Wolfgang D.Ehrenberger, Fachhochschule Fulda

Up to now, the human mind is the main productivity factor for making software. It is also the dominating factor for the verification of software and for its maintenance. Our mind is limited in various regards. Due to this we can understand some types of software easier than others. For maintenance and licensing of software knowing its understandability in a quantitative way would be helpful. The standard IEC 880 for software in nulear reactor protection systems e.g. requires software parts of high safety relevance to be simpler than normal ones.
This leads to the question of how to measure simplicity of software. In the literature several measures exist. Among the oldest ones are those of Akiama, McCabe and Halstead. More recently McCall, Rechenberg and many others have suggested new metrics. Zuse has assembled most of the suggested results. Kitchenham has investigated software development in connection with metrics. All these metrics, however, commonly neglect the human way of thinking and understanding.
This contribution tries to fill this gap. It is believed that understanding software is closely related to abstracting meanings from the code. It is the human mind that does this abstraction during the process of understanding. A certain model of the human mind is assumed. It postulates that specific basic mental actions are used during the abstraction process in understanding

programs. These actions are: reading, matching certain aspects with aspects that are already known, combining aspects, loading and storing them, searching and finding something from the long term memory. Psychologically speaking this model is based on the chuncking theory of understanding.
Furthermore it is assumed that the mentioned actions can be connected with a probability of being performed correctly and that they can be counted. In case of remembering things over a certain number of lines of code this number influences the probability of correctness of the action. In practical cases it is recommendable to draw graphs that illustrate the way understanding works (cf., e.g., Fig. 1). The introduced measure takes the form of real numbers between of $\{0..1\}$. 0 reflects the event of incorrect action, 1 reflects correct action. This approach can also be used to quantify the mental effort connected with program proofs. It turns out that understanding a proof may be more demanding than understanding the underlying software itself, because of the large number of mental comparisons needed. Understanding proofs, however, may not put such high demands on the long term memory or the knowledge of the reader than understanding programs; because the proof contains the intended computation result in their logical form already.

Designing for Risk Mangement — The Human Operator as an Imponderable Risk Factor in Safety-Critical Systems?

Wolfgang Dzida, Claus Hoffmann, Gesellschaft für Mathematik und Datenverarbeitung mbH, St. Augustin

From our review of our accidents we conclude that there is an increasing tendency in the design of safety-critical systems to substitute human control by mechanisms of process control engineering, the software of which should be licensed for its integrity. The rationale for diminshing the operator's control is based on the prejudice that the human operator is an imponderable risk factor.
The technology of interactive software systems may, however, enable the human operator to coordinate his control with technical control mechanisms more effectively — particularly in risky situations. The rationale for this approach is based on the requirements that technical control mechanisms shall not incline the operator to learn helplessness in risk situations. It is essential to respect the principle that responsibility cannot be delegated to machines.

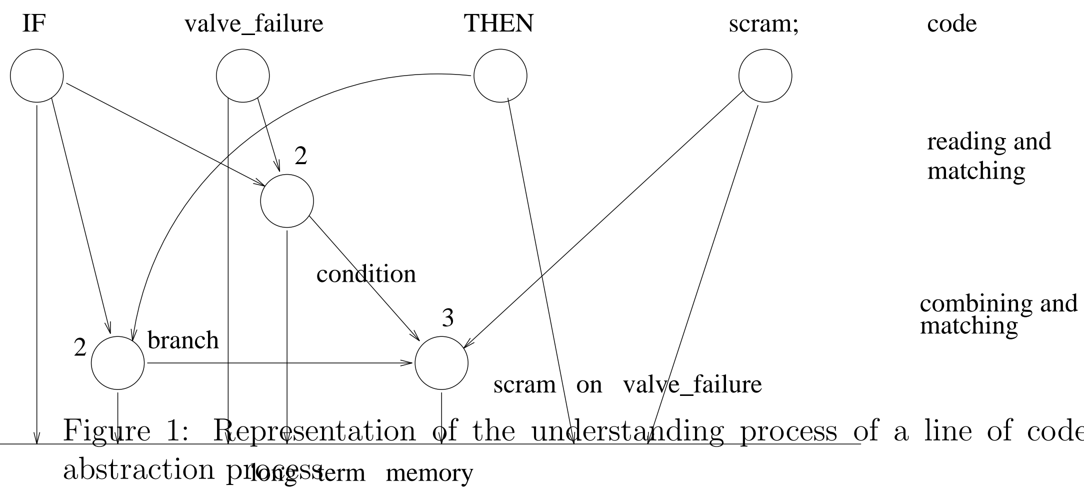

Figure 1: Representation of the understanding process of a line of code; abstraction process

The focus of this approach is not on the design of user-centered control panels or devices, because there is no doubt about the feasibility of such means. Nevertheless, lack of knowledge of the operator's potential of interventions into a safety-critical process may make the designer reluctant to implement them. Hence, this approach provides an analysis of risk situations as well as the operator's means to master the process by risk management. For familiarity the analysis deals with the risk management when driving a car. Means of risk management in a high-tech car are described and generalized to serve as paradigms of risk managements in various areas of the process industry and airplanes as well.

Safety Critical Application of Fuzzy Control

Gerhard-H. Schildt, TU Wien

After an introduction into safety terms a description of fuzzy logic was given. Especially, for safety critical applications of fuzzy controllers a certain fuzzy controller structure was described. Following items were discussed: Configuration of fuzzy controllers, design aspects like fuzzification, inference engine strategies, defuzzification, and types of membership funcions. As an example a typical fuzzy rule set was presented. Especially, real-time behaviour of fuzzy controllers was mentioned. An example of a fuzzy controller for temperature control in a reactor together with membership functions, inference engine strategy, and rule base was presented.

Application of Fuzzy-Control in Safety Related Systems for Process- and Energy-Technology

R. Hampel, N. Chaker, Hochschule für Technik und Wirtschaft Zittau/Görlitz

For improvement the safety and reliability it is necessary to use more and better information about the state of the process. Currently we have three components of technical activities:

- Control-Systems

- Limitation-Systems
- Safety-Systems

For Control and Safety Systems we use simple and robust technical design and control-algorithm with a small number of input and output signals. That is not enough for limitation systems. So we need also informations about changing rates, process history, non measurable values and so on.
The paper shows, that we can use for such a system with success the Fuzzy-Logic. The main problems are to generate the knowledge basis and the rule basis. For demonstration the way of development a Fuzzy-Limitation-System we use a very simple heating-system-model. For this case the aim is to limit the cover temperature after a fast decreasing of the cooling mass flow rate.
One of the main results of the investigation is, that the same structure and the same algorithm is applicable for the normal controller and the limitation system.
With the help of the Fuzzy Shell, which is included in the Simulation-System DynStar we investigated the influence of the membership function of the controller output area (two dimensional controller). So we could show, that we can reduce the number of the free parameters for optimation the Fuzzy-Controller. It is very important for the application. In the same way it is possible to complete the rule basis, if we have a partly unknown knowledge basis. In each case we use the information about the form of the controller output area.

Verification Support Environment (VSE)

Dieter Hutter, Bruno Langenstein, Claus Sengler, Jörg H. Siekmann, Werner Stephan, Andreas Wolpers, DFKI GmbH, Saarbrücken

We give an overview of the VSE system that was developed for the German Information Security Agency (BSI). This tool supports the formal development of provably correct software components. VSE is based on a method for programming in the large that provides means for structuring specifications as well as the implementation process. Formal developments following this method are stored and maintained in an administration system that guides the user and maintains a consistent state. Integrated deduction systems provide proof support for the various deduction problems arising during the development. In parallel to the development of the system itself two large

case studies were conducted in collaboration with one of the industrial partners.

Exceptions and (Real-Time) Control Systems

Coenraad Bron, Groningen University

Exceptions in Sequential Processes. Let us consider the specification of a component in a sequential program: `{P}S{Q}` which states that a component `S` has to terminate in a state satisfying the predicate `Q` when it is executed, starting in a state satisfying the predicate `P`. An implementation of `S` is called *correct* if it (can be proved that it) meets this specification. Using Dijkstra's weakest preconditions, the relation between `P, Q` and `S` is expressed by: $\mathtt{P} = wp(\mathtt{S},\mathtt{Q})$. The latter relation also implies `{not P}S{not Q}` i.e. `S` can *not* satisfy its post condition whenever its precondition is not satisfied. This is popularly expressed as "Garbage in, garbage out!", but preferably we would have an implementation of `S` that is *robust*, which can formally be expressed by the additional specification: `{not P}S{false}.` A correct implementation of this specification (surprisingly) satisfies: `{true}S{Q}.`
The importance of robustness is immediately clear if we consider software components that are developed independent from their application, as is the case for libraries. Such components cannot produce "garbage". As an example, consider a function for the calculation of a square root of a real number. The only sensible specification for such a component can be: `{true}sqrt(x:real){sqr(sqrt)=x}`, where = stands for approximately equal.
Any implementation of robustness must see to it that that a robust component does not terminate (in the normal sense) if it cannot satisfy its postcondition. A proper, but not very practical way to accomplish this is to have in the implementation of `sqrt` before the actual calculation starts, a loop like: `WHILE x < 0 DO;`
The concept of robustness is closely related to IF-statements in Dijkstra's guarded command language. It is explicitly stated that a statement of the form: `IF b1 -> s1 [] b2 -> s2 [] ... FI` is aborted when it is started in a state for which `NOT b1 AND NOT b2 AND NOT ...` holds.
Aborting a program and providing a (hopefully) illuminating error message is in effect a form of control, with the additional effect that the original specification of the program which dealt with 'normal' cases only is now relaxed in the sense that an alternative result, viz. the error message, is an accepted alternative.
Here we are at the heart of both the concept of *exceptions* and *exception handling*. Exceptions can be viewed as (named) violations of preconditions that are vital for operations to produce their required results. They are indispensable in the creation of robust software. Any form of abnormal

termination that takes place when an exception is detected can be viewed as the *handling* of that exception. In languages that provide an exception handling mechanism, it is recognized that –as long as exceptions exist– the handling might as well be brought under program control, than left to some magical mechanism which is supplied at the interface of operating system and application program. Note that this interface cannot be defined sharply. Any application may act as a subsystem of a cyclical nature, just like the operating system itself.

Exceptions in Control Systems. The behaviour of the controlled system is viewed as a set of co-operating and mutually synchronised processes. Each process consists of two parts: a0 the controlled process, which is a *sequence* of events and activities that takes place in the real world, and b) the controlling process. This is a –computer embedded– process which proceeds in synchrony with the external process, as enforced by the sensors and actuators by means of which it communicates with its contolled process.
The correctness of a controlling process depends essentially on three aspects: 1) the internal correctness of the sequential part of the process, 2) the proper synchronisation among the controlling processes, and 3) the validity of assumptions made about the behaviour of the external world. The last point is where controlling processes really differ from sequential processes. Whereas for the latter the correct behaviour depends solely on the validity of a precondition, the assumptions on which controlling processes are based must be permanently checked. This concept has been worked out in detail and the terms *constraint* and *constraint monitoring* have been introduced. Constraints become *active* as soon as a process enters a region (of code) for which the validity of that constraint is essential, and they must –in principle– be checked permanently. Due to the properties of our model, however, we may always assume that a process for which a constraint is violated has already proceeded up to its next interaction point, and therefore constraints need only be checked at interaction points. An additional benefit of this conclusion is, that wherever constraint violations are detected, the internal state of the processes affected is always consistent.
Essentially different from positions discussed in lieterature is the idea that constraints can be shared, i.e. all processes which are dependent on a particular constraint will be simultaneously in that constrained region, and upon violation, each process will make its own contribution to resetting the (relevant part of the) system into an acceptable state. Note that, although exception handling (and constraint violation handling) are forms of forward error recovery, in the case of cyclic control systems forward error recovery at the same time constitutes backward error recovery, so it should be possible to program recovery of a subsystem in such a way that a *local* constraint

violation has no serious effect on the overall operation of the system.

Mechanical Verification with the Boyer-Moore Prover: Usage and Problems

Debora Weber-Wulff, Technische Fachhochschule Berlin

In this presentation I first gave an overview of the Boyer-Moore theorem prover NQTHM and the computational logic is uses. The Boyer-Moore logic is a collection of recursive, side-effect free functions stated as s-expressions in the LISP-like language of the prover. Lemmata, usually stating the equality of two terms or the implication of one term from another, are also represented as s-expressions.
A publically available copy of Bob Boyer and J Moore's theorem prover NQTHM or Matt Kaufmann's interactive proof checker version PC-NQTHM can be obtained from Internet host ftp.cli.com (192.31.85.129) by anonymous ftp. A World Wide Web home page is offered by Computational Logic at http://www.cli.com.
A short example proof, the associativity of times, was printed in a handout and discussed in detail, examing the different transformations used in attempting to prove a lemma: decision procedures for propositional calculus, equality, and linear arithmetic; term rewriting based on axioms, definitions and previously proved lemmata; application of verified user-supplied simplifiers called "metafunctions"; variable renaming to eliminate "destructive" functions in favor of "constructive" ones; heuristic use of equality hypotheses; generalization by the replacement of terms by type-restricted variables; elimination of apparently irrelevant hypotheses; and mathematical induction. A list of theorems that have been proven was discussed, they include

- Mathematics
  - Prime factorization uniqueness
  - Unsolvability of the halting problem
  - RSA public key encryption algorithm is invertable
  - Gauß's Law of Quadratic Reciprocity
  - Chruch-Rosser Theorem
  - Gödel's incompleteness theorem
  - Irrationality of the square root of 2
  - Exponent two version of Ramsey's Theorem
  - Schroeder-Bernstein Theorem
  - Koening's Tree Lemma

  - Group Theory lemmata
  - Wilson's Theorem

- Hardware
  - Hypothetical processor FM8501
  - Motorola MC 68020
  - Processor FM 9001
  - Railroad gate controller
  - Fuzzy logic controller
  - Parameterized hardware modules
  - Synchronous circuits
  - VHDL formalization

- Theorem proving
  - Ground resolution prover
  - Theorem about generalization

- Various
  - Short Stack (Compiler for Gypsy to FM8501 machine code)
  - Towers of Hanoi
  - MACH Kernel specification
  - Scheduling theorem for real-time operating system
  - Implementation of an applicative language with Dynamic Storage Allocation

Problems associated with the prover tend to be found in the large effort in learning how to use the system, in a proper statement of the proof, in "seein" how to conduct the proof, and in the rather crude interface. But it is a very powerful prover and there are a number of libraries available which facilitate the usage. It is a stable system and has often been used outside of Austin, something which cannot be said for every verification system. It is, however, more of a proof checker than a proof discoverer. It is quite useful for hardware proofs, as it is possible to do induction over time or over the structure of a machine.
In the evening a more involved proof was demonstrated, the proof of the add-assign compiler.

Formal Methods for Safety Critical Systems

Victoria Stavridou, Royal Holloway and Bedford New College, Egham

The SafeFM project is pursuing research on the practical application of formal methods in the development and assessment of safety critical systems. The project has been active for over one year and it is the intent of this paper to report on our interim results. We review the aims and background of the project and report on work involving a mix of formal methods technology and existing best practice. In particular, we present a methodology for constructing coherent safety critical system specifications and critically evaluate its application to an avionics case study. We also describe our efforts in producing a practical combination of formal and structured techniques as well as an approach to formal methods tools classification and integrity.

The Use of Ada for Emulation of Formal Specifications

W.J. Cullyer, University of Warwick

To satisfy the needs for emulation of the specifications for a range of avionics subsystems, including Ground Proximity Warning and Airborne Collision Avoidance, a library of reusable software packages has been created in the high order computer programming language, Ada. Using elements from this library it it possible to create emulation programs in a relatively short period of time, which facilitate the investigation of a wide range of navigation, aircraft trajectory and Air Traffic Control (ATC) scenarios. Ground based primary and secondary radars and air/ground data links are not included explicitly in the library, but each aircraft is assumed to be equipped with a height transponder which transmits information both to ATC and to other aircraft in the vicinity. The software runs on a Personal Computer and has been tested by a number of undergraduate students who have created viable emulation packages, with comparatively little training.

Approaches to Safety Engineering of PLCs

Günter Heiner, Daimler Benz AG, Berlin

In the first part of the talk I gave a brief overview of the activities within Daimler-Benz Information Technology Research concerning dependable computing systems.
Then a new approach for developing and assessing PLC (Programmable Logic Controller) applications has been presented. This approach aims at reducing the cost of assessment and licensing of PLC application software significantly. It is based on a "Safe PLC Language" which is a well-structured textual subset of the International Standard IEC 1131 language, its compiler, and a library of verified standard components implementing reusable safety functions. The safe language has been formally defined (in contrast to the IEC language), the compiler has been fully implemented and parts of it have been formally verified.

A Safety Licensable PES

Wolfgang A. Halang, FernUniversität Hagen

The architecture of a special — and necessarily very simple — computer system was developed to carry out safety-related functions within the framework of distributed process control systems or programmable logic controllers (PLCs). According to well established and accepted methods, a correctly working and, due to the provision of dual channels, fault-detecting computer hardware was designed, and built as a prototype. The recognition of faults is the responsibility of a number of comparators realised with fail-safe components of the HIMA planar system.

The emphasis of the concept, however, lies on the software aspect, since the dependability of software does not match the one of hardware, yet. The originality of the pursued approach consists in providing, for the first time, immediate support for software verification already in the architecture. Essential characteristics of this architecture are complete predictability of time behaviour, determinism and surveillance of program execution and of all other activities of the computer system. Diverse back-translation[3] as the most powerful and only verification method for larger programs accepted by the licensing authorities is supported as well as is sequence control, expressed in the form of sequential function charts, occurring in many automation applications including those with safety responsibility. A minimum execution control program was implemented in firmware.

The draft guideline VDI/VDE 3696 of the VDI/VDE-GMA Technical Committee 5.3 shows, for instance, that less than 70 function modules ("software ICs") are sufficient to formulate the great majority of all automation problems occurring in a certain larger industrial domain (chemical engineering). The modules are re-usable in many different contexts because of their simplicity and universality. Owing to their limited complexity, their correctness can be proven with formal methods, e.g., with the help of predicate calculus or symbolic execution. This is necessary, since their correct operation is often crucial to fulfill the severe safety and reliability requirements of entire systems. The same observation could also be made for other industrial application domains, such as emergency shutdown systems, which need as few as four function modules. For each application domain a specific family of function modules is to be identified — only once, and not by the user himself —, to be verified with mathematical rigour and, for safety reasons, corresponding object code is to be provided in ROMs. The correctness proof of the elements in a function block library requires great effort and can only

[3] H. Krebs and U. Haspel: Ein Verfahren zur Software-Verifikation. *Regelungstechnische Praxis rtp* 26, 73 – 78, 1984

be carried out by experts. Nevertheless, this effort is justified by the safety requirements and kept within limits for each single application because of the libraries' re-usability.
Based on such module libraries, safety-related automation programs can be formulated graphically with the help of suitable CAD-tools by just linking module instances, i.e., in the language "Function Block Diagram" (FBD) as defined in the international standard IEC 1131-3. The formal verification of a compiler transforming such a graphical software representation into object code is not possible with the present state of the art. Nevertheless, it can be circumvented by basing program verifications directly on the generated object code, as it is required by the licensing authorities anyway. For this task, the architecturally supported method of diverse back-translation can be implemented efficiently and in a cost-effective way, since only the module connections have to be verified on the level of application programs. This two-step programming paradigm is clearly reflected in the hardware architecture. The processing of single function block invocations is the task of a slave processor, whereas a master processor implements the data flow constituting user programs. Thus, the opportunity arises to safety-license the slave processor with its entire software in a single generic type-approval. The concept makes sure that application programs are to be found in the master processor only, to which project-specific verifications of module connections implementing data flows can be confined: for each single application only the module connections realised by the master processor need to be verified.
The presented, with regard to hardware, firmware and user software safety-licensable programmable logic controller solves an urgent problem of industrial practice: now flexible programmable electronic systems can provably offer the same degree of safety as conventional hardwired controllers.

Specification of Safety and Reliability Requirements for Control Systems

Leo Motus, Tallinn Technical University

Many systems design methodologies start the development process by concentrating on the functional specification and design, and only at later stages requirements on safety, reliability, fault-tolerance, etc are added. Such approach has proven useful when developing data processing type applications. This paper argues that such practice is not appropriate for developing real-time, safety-critical systems. In the case of real-time systems it is crucial to

merge functional and non-functional (e.g. safety, reliability, fault-tolerance) requirements at the earliest possible stages of systems development so as to be able to assess timing correctness of the system. It is important, for pragmatic reasons, that quantitative timing correctness of the future system is, as much as possible, verified before designing and implementation stages start. If safety and reliability, etc, procedures are added at physical design or at testing stage, the earlier verified timing correctness is violated in unpredictable way. Subtle timing errors can be introduced which are practically impossible to detect by testing. On the other hand, attempts to eliminate such errors may have a snowball effect, requiring massive modifications in already implemented and tested parts of the system. In this paper a formalism (the Q-model) for systems specification and methods for analytical and/or simulational analysis of its timing behaviour are suggested. The same formalism enables, with slight additions, explicit study of safety and reliability properties of the specified system. As another by-product of the proposed approach, a natural co-operation between control and software engineers becomes possible during the systems development.

Redundant Controller Triggered with a Communication Channel

L. Trybus, M. Śnieżek, Z. Świder, University of Technology, Rzeszów

Uninterrupted automatic control is necessary for safety critical systems. This may be provided by multifunction controllers, even DIN-seized instruments. When one controller fails, the other assumes operation. Redundancy implemented by using a communication channel assures continuity of the outputs. This is not the case in standard set-up, where watch-dog output is monitored. In the solution described here, the controllers exchange brief messages every 50 ms, sending values of binary outputs and some other data. If the main controller has not sent at least one correct message within certain period, the back-up one immediately sets its outputs to values received recently. Communication proceeds in master/slave mode. Receiver checks its own transmitter (RS-485).

Designing Software Systems for High Integrity

Michael Goedicke, Universität Essen

For large software systems with high demands regarding safety it is necessary to explicitly describe system structure in terms of components and component connections. We present a concept of an independent software component which requires the description of the provided features and the requirements to other software components without actually referencing other components. We discuss briefly the Π-language which supports the specification of such software components. In this approach a software system is given by identifying the components needed and explicitly defining the connections between them. The resulting component configuration itself forms a component which can be used in other contexts as well. In contrast to many other approaches to software specification and design such an explicit description allows the clear identification of system and subsystem boundaries. This property is especially important in the case of safety critical systems where often components with high demands in terms of safety are combined with components of less criticality. Below we briefly survey the concepts of the Π-language wrt. safety critical systems.
The basic concept of Π is an object-oriented structuring of software, where objects serve as the unit to encapsulate data by operations and where the underlying data types of objects are given in module specifications. Thus each modular system consists of a hierarchy of modules, where each module encapsulates objects of a particular data type. Since the operations on objects can be executed concurrently, these basic building blocks were called *CEM*, which stands for *Concurrently Executable Module and its associated objects*. According to the underlying component concept each Π-component description consists of four sections: the three interface sections export, import and common parameters section and the body section. The common parameters section is import and export at the same time thus each datatype imported via the common parameters section is exported unchanged as well.
The key to independent software components is to describe semantic properties of a component in its interface *formally* and the concept of formal import. The latter means that import defines only necessary properties of other used components. Functional and concurrency properties are defined using equational specifications and predicate path expressions respectively. These properties are defined in partial (overlapping) specifications called views. Currently there are three views of a component specification in Π: the Type View specifying execution independent properties, the Imperative View defining execution related properties and the Concurrency view for specifying permitted execution orderings.
The body of a component defines its implementation in terms of the imported properties of other components stated in the import or common parameters

section. The component realization can be defined directly by specifying its properties using equational specifications in the Type View and alogrithms in the Imperative View. This direct implementation is referred to as a primitive component. In contrast to such a direct realization complex components are defined in the body by connecting components along their export/import interfaces.

The advantage of using such an approach to design specification in general is to offer a wide range of formalisms in an integrated way to describe properties of software components. In addition following the Π-approach outlined above the various relations between specification parts i.e. views, sections and component connections are made explicit. This structuring can be exploited for the reasoning process e.g. regarding safety and other critical system properties.

In addition such an explicit description of software architecture and the strict notion of encapsulation of Π-components allow one also to implement specialized measures in case of connecting safety critical with non critical components. Such measures would protect the critical components against failures or malfunctions in the non critical parts. Such measures would materialze as capsules providing a general scheme which can be used throughout the system under development.

Formal Design of a Real-Time Operating System Kernel

Ronald M. Tol, University of Groningen

In the talk we present a formally developed kernel for use in embedded applications. The kernel is part of a larger Architecture for hard Real-TIme Environments (ARTIE).

A relatively simple formal method, Hoare-logic extended with a Clock-variable to denote real time, is used to specify and verify properties of the kernel.

Our kernel is application-oriented. This is based on the fact that the kernel supports an application-oriented programming language: Hi-PEARL. This means, among other things, that the kernel implements the tasking model as defined in (Hi-)PEARL including different types of task schedules. 'On-the-fly', we have defined the semantics of relevant parts of this programming language.

Also, a feasible scheduling policy has been developed, earliest-critical-deadline-first (ECDF) scheduling. We have shown how an application programmer

can establish that critical tasks of the application program meet their deadlines. For communication and synchronization we have introduced so-called pre-emption points in the code of non-critical tasks. At such a point the running task gives control to the scheduler. In such a way co-operative multitasking is realized. Based on the scheduling policy, a task-oriented memory management scheme is proposed. The resulting algorithm is formally derived and proven correct.
Subsequently, we assessed the timing behaviour both qualitatively and quantitatively. The timing behaviour depends on the application and the capacity of the available resources. It turns out that the size of main memory has a significant influence on timing behaviour.
Finally, we have implemented the kernel and the architecture in a simulation environment. We have evaluated the kernel in a simulation of a typical safety-critical application, a computer controlled railroad crossing. The application is developed using the CASE-tool EPOS.

Concurrency and Distribution in Safety-Critical Systems

Bernd Krämer, FernUniversität Hagen

There is a wide consensus that formal methods for the specification and verification of software used in safety-critical applications may contribute to achieving a high level of software integrity. For sequential software these techniques are relatively mature and they are slowly gaining space in complex practical applications. Examples have been discussed throughout the seminar.
For distributed systems, which dominate in technical application domains, the situation looks somewhat different. Difficulties arise due to additional behavioral issues such as concurrency, nondeterminism, synchronization and the need for continuous service. Properties such as partial correctness and termination, which are well-understood in the context of sequential systems, are replaced by safety and liveness requirements. But these are difficult to prove for realistic systems by use of existing analysis and proof techniques including reachability analysis, model checking, bisimulation or inductive techniques. The purpose of this presentation was to raise these issues and encourage seminar participants to take them into consideration in their future work.