안전성이 요구되는 응용분야에 사용하는 프로그램 가능한 전자시스템

정 순 기† Wolfgang A. Halang†† Coen Bron†††

요 약

프로그램 가능한 논리제어기(PLC : Programmable Logic Controller)의 활용을 위한 낮은 복잡성과 결함탐색 기능을 갖는 컴퓨터 구조를 설계하였다. PLC의 반복적인 운영모드와 명세단계, 응용분야에 따라 표준화한 소프트웨어 기능모듈들의 상호연결을 기본으로 하는 그래픽 프로그래밍 패러다임(paradigm)이 구조적으로 지원된다. 그러므로 설계과정에서 프로그래밍 단계와 가장 단순하면서도 엄격한 Diverse Back Translation 방법에 의해 응용소프트웨어의 안전성 인증을 가능케하는 컴퓨터 실행(machine execution) 단계 사이의 의미상 격차(semantic gap)가 제거된다.

A Programmable Electronic Systems Dedicated to Safety Related Applications

Soon-Key Jung†, Wolfgang A. Halang†† and Coen Bron†††

ABSTRACT

A low complexity, fault detecting computer architecture for utilisation in programmable logic controllers is designed. The cyclic operating mode of PLCs and a specification level, graphical programming paradigm based on the interconnection of application oriented standard software function modules are architecturally supported. Thus, by design, there is no semantic gap between the programming and machine execution levels enabling the safety licensing of application software by an extremely simple, but rigorous method, viz., diverse back translation.

1. Introduction

In society, there is an increasing awareness of and demand for dependable technical systems in order not to endanger human lifes and to prevent environmental disasters. Computer based technical systems, which are increasingly being applied for both control and automation functions under real time constraints to enhance flexibility and productivity, have the special property and they consist of hardware and software. Hardware is subject to wear and faults occurring at random, which may be of a transient nature. These sources of non-dependability can (to a very large extent) successfully be coped with by applying a wide spectrum of redundancy and fault tolerance methods. The advent of the VIPER microprocessor [5], the correctness of whose design was formally proven, signifies a leap forward in the direction of eliminating design errors in programmable electronic systems as well. In software, on the other hand, there are no faults caused by wear, environmental events etc. Instead, all errors are

†정 회 원 : 충북대학교 컴퓨터공학과 교수
††비 회 원 : Professor, Faculty of Electrical Engineering Fern Univ. of Hagen, Germany
†††비 회 원 : Professor, Dept. of Computing Science Univ. of Groningen, The Netherlands

논문접수 : 1994년 4월 25일, 심사완료 : 1994년 12월 6일

design errors, i.e., of systematic nature, and their causes are always (latently) present. Hence, dependability of software cannot be achieved by reducing the number of errors contained by testing, checks, or other heuristic methods to a low level, which is generally greater than zero, but only by rigorously proving that it is error free. Taking the high complexity of software into account, only in exceptional cases this objective can be reached with the present state of the art. Whereas other researchers have the general situation in mind and, therefore, have to yield to the complexity problem, it is our intention to make an important step into the direction of designing a workable and useful programmable electronic system, which can be safety licensed in its entirety, by exploiting the intrinsic properties of a special, but not untypical case, that was identified in industrial control problems. Here the complexity turns out to be manageable, because we restrict our attention to rather simple computing systems in the form of programmable logic controllers, and since application domains exist demanding software of limited variability only, which may be implemented in a well structured way by graphically interconnecting carefully designed and rigorously verified "software ICs". Despite the mentioned restrictions of generality, the here described work is scientifically relevant and technologically useful, since its application area comprises the above mentioned technical systems in charge of safety critical control functions.

There are already a number of established methods and guidelines, such as IEC 880 [3], which have proven their usefulness for the development of high integrity software employed for the control of safety critical technical processes. Prior to its application, such software is further subjected to appropriate measures for its verification and validation. However, according to the present state of the art, these measures cannot guarantee the correctness of larger programs with mathematical rigour. Moreover, prevailing legal requirements demand that object code must be considered for the correctness proofs, since compilers-or even assemblers-are themselves far too complex software systems, as that their correct operation could be verified. Therefore, depending on national legislation and practice, the licensing authorities are still very reluctant or even refuse to approve safety related systems, whose behaviour is exclusively program controlled. In general, safety licensing is denied for highly safety critical systems relying on software with non-trivial complexity.

To provide a remedy for this unsatisfactory situation, the architecture of a customised real time computer control system is developed, which can carry out safety related functions within the framework of distributed process control systems and programmable logic controllers. It explicitly supports sequence controls, since many automation programs including safety relevant tasks are of that kind. The architecture features full temporal predictability, determinism, as well as supervision of program execution and of all other activities of the computer system, and supports the software verification method of diverse back translation, whose utilisation turns out to be very easy, economical, and time efficient. The presented approach is unique as the first effort to provide support for software verification already in the architecture.

The leading idea followed throughout this design is to combine already existing software engineering and verification methods with novel architectural support. Thus, the semantic gap between software requirements and hardware capabilities can be closed, relin-

guishing the need for not safety licensable compilers and operating systems. By keeping the complexity of each component in the system as low as possible. the safety licensing of the hardware in combination with application software is enabled on the basis of well established and proven techniques.

2. A Software Engineering Paradigm

Standardisation bodies of the Society for Measurement and Automation Technology and of the chemical industries in Germany have identified and defined a set of 67 application specific function modules suitable to formulate-on a very high level employing the graphical "Function Block Diagram" and "Sequential Function Chart" languages recently defined by the IEC International Standard 1131-3 [4]-the large majority of the occurring automation problems [9]. To give an impression of these modules' functionality, we provide the following list :

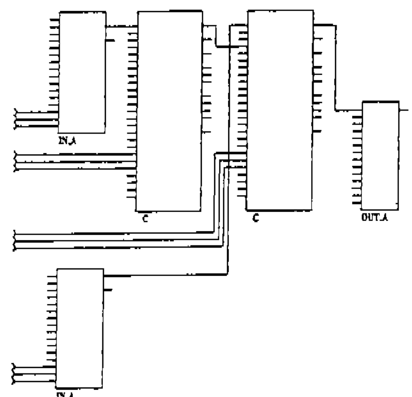
- Monadic mathematical functions
- Polyadic mathematical functions
- Comparisons
- Monadic Boolean function
- Polyadic Boolean functions
- Edge detectors
- Selection functions
- Selection by 1 out of N bits
- Counters, monostables, bistables, timers
- Process input/output
- Network communication input/output
- Dynamic elements and regulators
- Conditioning for display and operation

Written in the [4] high level language "Structured Text", these software modules are usually quite short : their source code does not exceed two pages. Therefore, their correctness can be formally proven, e.g., using predicate calculus, but also symbolic execution or, in some cases, even complete test. Note that the rather costly safety licensing of a function module set needs to be carried out only once, after a certain set has been identified and standardised for a given application area. The licensing costs can be spread over many implementations leading to relatively low costs for each single automation project. In order to give another typical example, the programming of emergency shut-down systems, which is usually performed graphically in form of functional logic diagrams to describe the mapping from Boolean inputs to Boolean outputs as functions of time such as, for instance,

if a pressure is too high
then a valve should be opened
and an indicator should light up *after* 5 seconds

even requires as few as only four function modules, viz., three Boolean operators and a timer. For an in-depth treatment of the function module concept we refer to [10].

The above mentioned analysis of process automation suggests to introduce a new programming paradigm, viz., to compose software out of high level user oriented building blocks instead out of low level machine oriented ones. Whereas a single machine instruc-



(Fig. 1) A graphically formulated program

tion taken out of a program context does not reveal its purpose, the occurrence of a certain function module instance usually gives already a clue about the problem, its solution, and the module's role in it. Therefore, we select basic function modules as elementary units of application programming. Essentially, for any application area, there will be specific sets of basic functions modules, although certain functions like analogue and digital input and output have general relevance.

For the formulation of automation applications with safety properties, basic function modules are only interconnected with each other, i.e., single basic functions are invoked one after the other and, in the course of this, they pass parameters. Besides the provision of constants as external input parameters, the basic functions' instances and the parameter flows between them are the only language elements used on this programming level. The software development is carried out in graphical form, using an appropriate CAD tool : the instances of the basic functions are represented by rectangular symbols and the data flows are depicted as connecting lines (Fig. 1). Then, a compiler transforms the graphically represented program logic into object code. Owing to the simple structure, this logic is only able to assume, the generated programs contain no other features than sequences of procedure calls and some internal moves of data.

This high -or even specification -level programming method is very similar to the programming language LUCOL [8], which was specifically developed for safety critical applications. LUCOL modules correspond to the basic functions and LUCOL programs are sequences of module invocations with data passing as well. This similarity could be utilised to validate programs, which are produced by interconnection of basic functions, since the validation tool SPADE [1] was already successfully applied for this purpose[8]. SPADE works especially well on procedures, which are closed in themselves and possess well defined external interfaces, such as represented by basic function modules. Although the static analyser SPADE and the comparable MALPAS [7] have proven to be highly valuable tools, they cannot establish the correctness of software on their own.

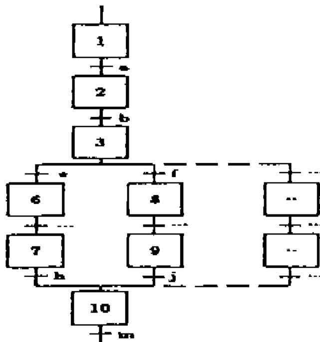
Fortunately, an -apparently still impossible and, therefore, presently unavailable- safety licensed compiler transforming graphical software representation into object code is not a necessary precondition to employ this high level programming paradigm. The application software may be safety licensed by subjecting its loaded object code to diverse back translation, a verification method which was developed in the course of the Halden experimental nuclear power plant project [6]. This technique consists of reading machine programs out of computer memory and giving them to a number of teams working without any mutual contact. All by hand, these teams disassemble and decompile the code, from which they finally try to regain the specification. A safety licence is granted to a software if its original specification agrees with the inversely obtained re-specifications. Of course, the method is generally extremely cumbersome, time consuming, and expensive. This is due to the semantic gap between a specification formulated in terms of user functions and the usual machine instructions carrying them out. Applying the programming paradigm of basic function modules, however, the specification is directly mapped onto sequences of procedure invocations. The object code consists of just these calls and parameter passing. It takes only minimum effort to interpret such code implementing just module interconnections and to redraw graph-

ical program specifications from it. If the implementation details of the function modules are part of the architecture they remain invisible from the application programming point of view and do not require safety licensing in this context.

· The software verification method of diverse back translation [6] is greatly facilitated by the problem oriented architecture introduced in the next section. Owing to the employment of basic function modules with application specific semantics as the smallest units of software development, the effort for the method's utilisation is by orders of magnitude less than in the cases reported by [2]. Furthermore, the employed principle of software engineering reduces the number of possibilities to solve a given single problem in different ways.

Therefore, it is considerably simpler to check the equality of reversely documented software with an original program. Finally, tools for the graphical back translation of memory resident programs are already part of the standard support software of distributed process control systems, thus facilitating the application of diverse back translation for verification purposes.

Many automation programs including safety relevant applications have the form of sequence controls composed of steps and transitions. With our architecture we support linear sequences of steps and alternative branches of such sequences (Fig. 2). Parallel branches in sequential function charts should either be implemented by hardware parallelism or already resolved by the application programmer in the form of explicit serialisation. While in a step, an associated program, called action, developed according to the above paradigm is being executed. Also, for purposes of a clear concept, of easy conceivability and verifiability, and in order not to leave the Petri net concept of sequence controls, we only permit the utilisation of non-stored actions. All other types of actions as defined in IEC 1131-3 [4] can be expressed in terms of non-stored ones and re-formulated sequential control logic.



(Fig. 2) Sequential function chart

3. A Safety Oriented Architecture

As architecture for our safety oriented programmable logic controller we select an asymmetrical configuration of four processors which consists of two pairs of master/slave processors. For the envisioned purpose only a very simple master processor with just two instructions is required. Thus, the utilisation of the diverse back translation method for the verification of application software is greatly facilitated. For most application areas in process automation, the slave processor must have general purpose capabilities. The objective of our PLC suggests to employ the VIPER 1A [5] chip in the slave, because the VIPER is the only available microprocessor whose design was formally proven correct, and in its 1A version it supports fault detecting operation in dual channel configuration. The VIPER 1A provides, namely, information on its current internal state at dedicated output pins. If two of these processors operate as a redundant pair synchronously executing identical programs, each VIPER 1A is able to

continuously compare its internal state with that of the other one. Both processors are stopped upon a mismatch of states. In case a new processor is developed to be employed as slave, maximum simplicity and a minimum application specific instruction set ought to be emphasised. If possible, the instruction set should then correspond to the low level IEC 1131-3 [4] programming language "Instruction List", which essentially represents a single address assembly language.

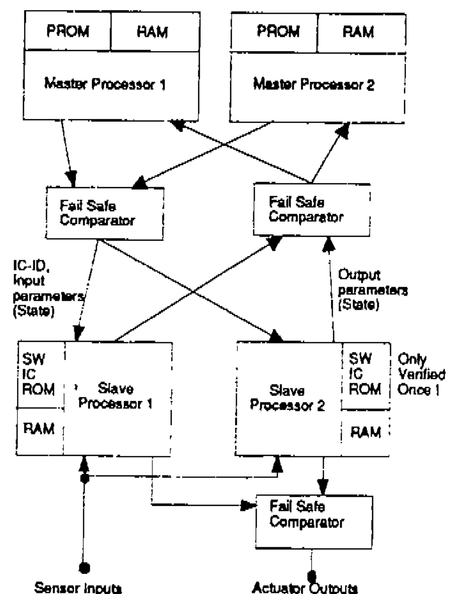
We assume that a technical process is to be controlled by a distributed computer system. First, all safety relevant hardware and software functions and components are clearly separated from the rest of the automation system and, then, also from each other. Under any circumstances it should be the objective in this process to keep the number of these functions as small as possible. Each of the thus identified functions is assigned to a separate dedicated processor in the distributed system [6].

Since it is not our objective to save hardware costs, but to facilitate the conceivability of the implemented software and of its execution process, we design an architecture with, conceptually, two processors:-a control flow processor (master) and-a basic function block processor (slave).

These two processors are to be implemented by separate physical units. In fact, there are even more processors to meet the fault detection requirement. Thus, we achieve a clear and physical separation of concerns : execution of the basic function modules in the slave processor, and all other tasks, i.e., execution control, sequential function chart processing, and function module invocation, assigned to the master. This concept implies that the application code is restricted to the control flow processor, on which the project specific safety licensing can concentrate.

There is special architectural support for the cyclic operating mode of programmable logic controllers implemented in the master processor. To enable the detection of faults in the hardware, a dual-channel configuration is chosen, which also supports diversity in form of different master processors and different slave processors. All processing is simultaneously performed on two processors each and all data communicated are subjected to comparison. At least one of the master processors should have the extremely simple organisation described below. (Fig. 3) gives a conceptual diagram of the master/slave PLC architecture.

The basic function processor performs all data manipulations and takes care of the communication with the environment. The master and slave processors communicate with each other through two FIFO-queues. They execute programs in co-ordination with each other as follows. The master processor requests the slave to execute a function block by sending the latter's identification and the



(Fig. 3) Configuration of a PLC with master/slave processors

corresponding parameters and, if need be, also the block's internal state values via one of the FIFO-queues to the slave processor. Here the object program implementing the function block is performed and the generated results and new internal states are sent to the master processor through the other FIFO-queue. The elaboration of the function block ends with fetching these data from the output FIFO-queue and storing them in the master's memory. A number of fail safe comparators checking the outputs from the master processors before they reach the slaves and vice versa completes a fault detecting two-channel configuration.

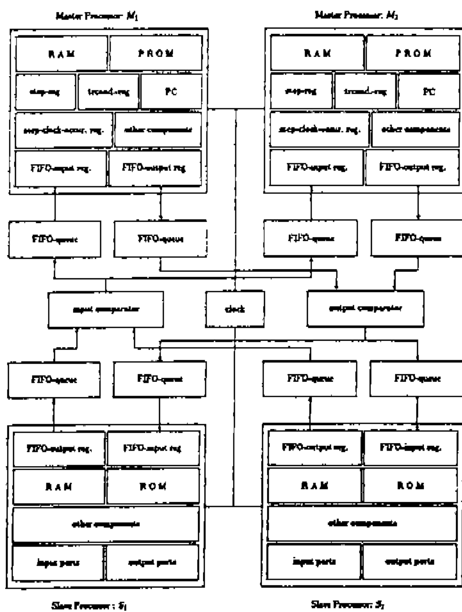
In order to prevent any modification by a malfunction, in our safety oriented architecture all programs must be provided in read only memories (ROMs). For practical reasons, generally there will be two types of these memories. There is no program RAM at all. The code of the basic function modules resides in mask programmed ROMs, which are produced under supervision of and released by the licensing authorities, after the latter have rigorously established the correctness of the modules and the correctness of the translation from Structured Text into object code. On the other hand, the sequences of module invocations together with the corresponding parameter passing, representing the application programs at the architectural level, are written into (E)PROMs by the user. This part of the software is subject to project specific verification again to be performed by the licensing authorities, which finally still need to install and seal the (E) PROMs in the target process control computers. The master/slave configuration is chosen to physically separate two system parts from one another : one whose software only needs to be verified once, and the other one performing the application specific part of the software.

In commercially available PLCs, the execution time for a step generally varies from one cycle to the next depending upon the program logic performed and the external conditions evaluated each time. Therefore, the measurement of external signals and the output of values to the process is usually not carried out at equidistantly spaced points in time, although this may be intended in the control software. To achieve full determinism of the time behaviour of programmable logic controllers, a basic cycle is introduced. The length of the cycle is selected in a way as to accommodate during its duration the execution of the most time consuming step occurring in an application (class). It is supervised that the execution time of a step does not exceed this cycle period by awaiting, at the end of the step's program processing and after the evaluation of the corresponding transition condition(s), the occurrence of a clock signal, which marks the begin of the next cycle. An overload situation or a run time error, respectively, is encountered when the clock signal interrupts an active application program. In this case a suitable error handling has to be carried through. Although the introduction of the basic cycle exactly determines a priori the cyclic execution of the single steps, the processing instants of the various operations within a cycle, however, may still vary and, thus, remain undetermined. Since a precisely predictable timing behaviour is only important for input and output operations, temporal predictability is achieved as follows. All inputs occurring in a step are performed en bloc at the beginning of the cycle and the thus obtained data are buffered until they will be processed. Likewise, all output data are first buffered and finally sent out together at the end of the cycle.

In the sequel, we discuss the hardware components

- master processor,
- FIFO-queue,
- comparator, and
- system clock

of this special purpose computer in detail. The complete fault detecting system architecture is depicted in (Fig. 4).



(Fig. 4) Fault detection master/slave PLC architecture

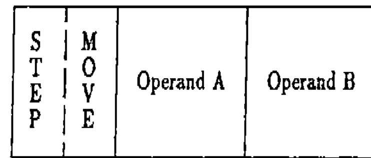
The master processor executes the programs stored in its PROM memory. Besides this program memory, the master's address space also comprises a RAM memory and various registers. The latter are:

1. the FIFO-input register,
2. the FIFO-output register,
3. two step registers, viz., step identifier and step initial address, and
4. the transition condition register.

Furthermore, it has a program counter (PC) and a single bit step-clock-occurred register, which are not accessible to the programmer. The master processor performs two machine instructions only, viz.,

- MOVE and STEP.

The instruction format is shown in (Fig. 5). The PROM address of an instruction to be executed is given by the PC. The instruction type is identified by the instruction word's first bit.



(Fig. 5) Instruction Format

The MOVE instruction has two operands, which directly address locations in the address space of the processor. Thus, the memories and the above mentioned registers can be read and written. A read from the FIFO-input register implies that the processor has to wait when the input FIFO-queue register is empty. In case of writing into the output FIFO-queue register, the processor also has to wait when the register is full. Execution of a MOVE implies incrementation of the program counter.

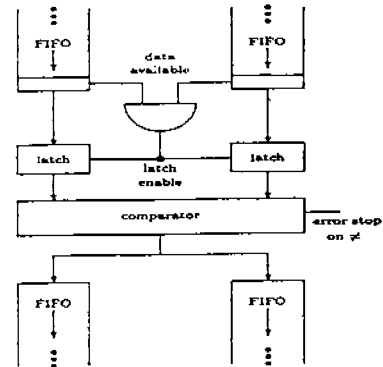
The STEP instruction does not have operands. Since designed for the implementation of PLCs, the programs executed by the master processor consist of sequences of steps. Behind the program segment of each step a STEP instruction is inserted, which checks whether the segment was executed within a step cycle frame or not. The step cycle is a periodic signal generated by the system clock and establishing the basic time reference for the PLC operation. If the execution of a segment does not terminate within a step cycle, an error signal is generated and the program is stopped. Normally, segment execution ter-

minates before the instant of the next step cycle signal. Then the processor waits until the end of the present cycle period.

When the clock signal finally occurs, according to the contents of the transition condition register it is decided whether the step segment is executed once more or whether the execution of the logically subsequent step is commenced, i.e., whether the program counter is re-loaded from the step-initial-address register or if another segment's initial program address is read from a memory location called next-step-address. Since program branching is only possible in this restricted form, erroneous access to code of inactive steps is prevented, thus representing a very effective memory protection mechanism.

The design objective for providing the FIFOs is to implement easily synchronisable and understandable communication links, which decouple the master and slave processors with respect to their execution speeds. The FIFO-queues consist of a fall-through memory and two single bit status registers each, viz., FULL and EMPTY, which indicate the filling states of the FIFOs. The status registers are not user accessible. They are set and reset by the FIFO control hardware and, if set, they cause a MOVE to a FIFO's input port or from an output port, respectively, to wait until space in the FIFO becomes available or data arrives.

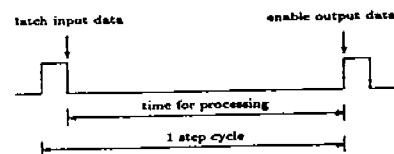
The comparison for equality of the outputs from the two master processors and of the inputs from the two slave processors, respectively, is carried out by the two comparators placed into the FIFO-queues. Since the responsibility for detecting errors in the system lies on these comparators, they need to meet high dependability requirements and must, therefore, be implemented in a fail safe technology. A comparator is connected to four FIFOs, two at its input and two at its output. The first data elements from each input queue are latched and subsequently compared with each other. If both latches do not hold the same value, then an error signal is generated, which stops the operation of the entire system. Otherwise, the value is transferred into both output FIFOs. The comparison of FIFO data is shown in (Fig. 6).



(Fig. 6) Comparison of FIFO data

The system timer generates, besides other periodic signals required for the operation of the hardware, a step cycle clock. This periodic signal marks the time frames dedicated for the execution of all steps (in the sense of PLCs). The step cycle is also applied to set the step-clock-occurred register. (Fig. 7) shows the basic timing diagram for the complete system.

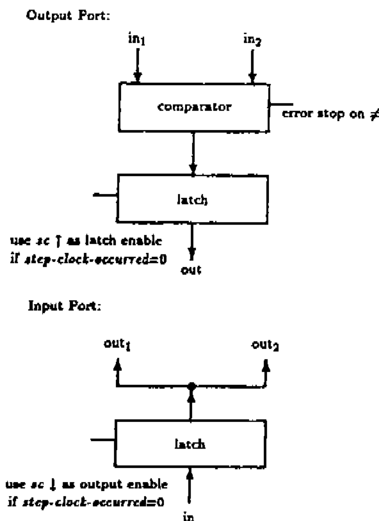
To the end of redundant hardware support and supervision of the sequential function chart control structures we provide the following measures in our safety related architecture. The identification of the step active at a given time is kept in a separate register, which is especially protected. The regis-



(Fig. 7) Basic timing diagram

ter's contents is displayed on an external operator's console.

Our PLC communicates with external technical processes through fault detecting input/output ports (cp. (Fig. 8)) attached to the slave processors. Output data words generated by the two slaves are first checked for equality in a fail safe comparator and, subsequently, they are latched in an output port. If output data are not identical, then an error signal is generated leading to a system stop. Output is not enabled, i.e., does not become effective to the environment, before the step-clock-occurred register is set by the step cycle signal. Also input data words arriving from the external technical process are first latched in an input port. With the next step cycle signal they are made available for input to both slave processors.



(Fig. 8) Output port and input port

4. A Software Development Environment

A set of tool prototypes was developed supporting vendor independent graphical programming of PLCs. In the sequel we give a survey on the functionality of these tools, which are aimed at facilitating methodical and structured top-down design, design in the conceptual phase, and setting up libraries of well tested standard modules for single projects or classes of projects.

When new software is to be developed, application programmers use CAD tools to set up drawings of function blocks and sequential function charts. In particular, they fetch appropriate graphical objects from libraries, place them in worksheets, and link, according to given logic, connectors to these objects. After the drawing process is complete, the worksheets are stored and then submitted to utility programs of the CAD tools, which generate lists of all objects and all interconnection nodes occurring in the drawings, so-called net lists. These lists constitute textual representations which are fully equivalent-but for geometrical aspects-to the original drawings. Then, the lists are submitted to a further postprocessor, viz., a compiler, generating code in the IEC language Structured Text. In particular, the compiler produces complete program units with declarations of input, output, and local variables, of tasks, and of instantiated function blocks, with calling sequences of functions, function block instances, and tasks, and with encoded descriptions of steps, transitions, and actions. The texts of these program units are stored in libraries, which, of course, could also be filled by text editors with hand coded modules. These text elements serve as input for the final postprocessor, i.e., a translator, that generates object programs for our architecture. Due to the restricted form the source programs are only able to assume, the translator produces calling sequences to and parameter passing between the user invoked functions and function blocks, the object code of which is contained in system firmware ROMs of the slave processor.

For documentation and verification purpos-

es, functions and function blocks are also coded in the language Structured Text and stored in a text library. In order to save a further editing step needed to set up entries for the graphical library of a CAD tool, it is desirable to have a tool available which automatically generates from the modules' source codes corresponding graphical symbols to be used in the drawing process. Therefore, a further tool was developed, which interprets the name and the input and output declarations of a function or function block in order to generate the textual description of an appropriately sized graphical symbol. The output of this tool is finally subjected to a utility of the CAD system used, which translates the descriptions into an internal form and places the latter into its component library.

5. Software Safety Licensing

Using the tool set described above, it is assumed that all elements of an employed function block set contained in a library were first verified with appropriate formal methods. Hence, for any new application program, only the proper implementation of a particular interconnection pattern of invoked function block instances needs to be verified. For this purpose we subject the object code loaded into the master processor to diverse back translation [6], because our tool set contains a number of high complexity utility and compiler-like programs, whose correctness cannot be established rigorously. Although it may be considered as a rather non-elegant brute force method, diverse back translation is especially well suited for the verification of the correct implementation of graphically specified programs on the architecture introduced above. This is due to the following reasons:

– The method is essentially informal, easily conceivable, and immediately applicable without any training. Thus, it is extremely well suited to be used on the application programming level by people with the most heterogeneous educational backgrounds. The ease of understanding and use inherently fosters error free application of the method.

– Since graphical programming based on application oriented function blocks has the quality of specification level problem description, and because by design there is no semantic gap in our architecture between the levels interfacing to humans and to themachine, diverse back translation leads back in one easy step from machine code to problem specification.

– For our architecture, the effort required for the utilisation of diverse back translation is by several orders of magnitude smaller than for the von Neumann architecture.

As already its name implies, diverse back alranslation is a verification method to be carried out with diverse redundancy. Originally, this called for different teams of human inspectors. Since in the case considered here there is only one rather simple reverse translation step, we are optimistic that the licensing authorities will eventually accept the following procedure. Verification by back translation is carried out by a number of different programs, which should be proven in practice, but do not need to be formally verified. Such programs are to yield graphical outputs. An official licensor performs the back translation as well, compares his results with the ones of the verification programs on one hand, and with the original graphical application program under inspection on the other, and, upon coincidence, issues a safety licence. Such a procedure is in line with the dependability requirements for diversely redundant programs demanded by the licensing authorities and necessitates only the minimum of highly expensive human involvement, viz.,

one licensor, who is always indispensable to take the legal responsibility for issuing a safety licence.

6. Prototype Implementation

Employing VIPER 1A microprocessors, we have built a prototype of the PLC architecture described. Its utilisation in practice showed that implementing the functionality of a hard wired emergency shut-down system with our PLC architecture is feasible, and that the programming paradigm based on formally verified function modules can render error free software. The latter together with a fault detecting hardware platform allows to implement programmable safeguarding systems sharing the fail safe feature with the well established hard wired solutions.

7. Conclusion

Economical considerations impose stringent boundary conditions on the development and utilisation of technical systems. This holds for safety related systems as well. Since manpower is becoming increasingly expensive, also safety related systems need to be highly flexible, in order to be able to adjust them to changing requirements at low costs. In other words, safety related systems must be program controlled. Thus, we expect that the use of hard wired safety systems will diminish in favour of computer based ones.

In our society there is a growing concern for safety (which comes hand in hand with the increasing awareness for the environment). This has important consequences for the assessment of computer controlled systems. One has begun to realise the inherent safety problems associated with software. Since it appears unrealistic to abandon the use of computers for safety relevant control purposes-on the contrary, for the reasons mentioned above, there is no doubt that their utilisation in such applications is going to increase considerably the problem of software dependability will exacerbate severely.

In the situation as outlined above, this paper is timely to address a real problem. It does not present a solution to all open questions in safety related computing, but a beginning is made which is practically feasible and applicable to a wide class of common control problems. Hence, we hope that the concept presented here leads to the breakthrough that, ultimately, discrete or relay logic can be replaced by programmable electronic systems executing safety licensed high integrity software to take care of safety critical functions in industrial processes. Meeting the need of society for more dependable computing systems under the prevailing economical restrictions, we expect that the concept will give rise to workable industrial implementations.

In a constructive way, and using presently available methods and hardware technology only, for the first time a computer architecture was defined, which enables the safety licensing of complete programmable electronic systems including the software. Special emphasis was dedicated to the software side, since it is felt that software dependability still needs to catch up with the one already achieved for hardware. Our solution deviates from the classical approach (as still followed by the mainstream in computer science and engineering) by using hardware as much as possible, but not necessarily in the most (hardware-) cost effective way, and by enforcing the (re-) use of pre-engineered off-the-shelf software modules. The former deviation is in line with the technological development: there is cheap hardware in abundance and it ought to be used to achieve our objec-

tive, viz., to implement inherently safe systems. The other deviation represents leaving the tradition of the von Neumann architecture allowing maximum flexibility-also to commit errors and to be unsafe. Later, when experiences will have been gained and a sound engineering methodology for safety related systems will have been developed, restrictions may be relaxed and more flexible approaches may be devised addressing wider application areas.

References

[1] Clutterbuck, D.L., and Carr , B.A., "The verification of low-level code", IEE Software Engineering Journal, pp. 97-111, 1988.

[2] Dahll, G., Mainka, U., and M rtz, J., "Tools for the standardised software safety assessment (The SOSAT Project)", In Safety of Computer Control Systems, Ehrenberger, W.D. (Ed.). IFAC Proceedings Series, Pergamon Press, Oxford, No. 16, pp. 1-6, 1988.

[3] IEC International Standard 880, Software for Computers in the Safety Systems of Nuclear Power Stations, International Electrotechni-cal Com-mission, Geneva, 1986.

[4] IEC International Standard 1131-3, Program-mable Controllers, Part 3 : Programming Languages, International Electrotechnical Commission, Geneva, 1992.

[5] Kershaw, J., "The VIPER Micro-processor", Royal Signal and Radar Establishment, Malvern, England, Report No. 87014, 1987.

[6] Krebs, H., and Haspel, U., Ein Verfahren zur Software-Verifikation, Regelungstechnische Praxis rtp, 26, pp. 73-78, 1984.

[7] "MALPAS"(Malvern Program Analysis Suite), Rex, Thompson and Partners Ltd., Farnham, England.

[8] O'Neill, I.M., Clutterbuck, D.L., Farrow, P.F., Summers, P.G., and Dolman, W.C., "The formal verification of safety-critical assembly code", In Safety of Computer Control Systems, Ehrenberger, W. D. (Ed.), IFAC Proceedings Series, No. 16. Pergamon Press, Oxford, pp. 115-120, 1988.

[9] VDI/VDE Draft Guideline 3696, "Vendor independent configuration of distributed process control systems", Beuth-Verlag, Berlin, 1992.

[10] Z ller, H.. "Wiederverwendbare Software-Bausteine in der Automatisierung", VDI-Verlag, D sseldorf, 1991.

정 순 기

1971년 고려대학교 졸업
1982년 Dortmund 대학교 졸업, Computer Science, Dipl.-Inform.
1994년 Groningen 대학교 졸업, Computing Science, Dr.
1983년 Philips Data Systems
1984년 정보통신연구소, SE부, 책임연구원
1985년~현재 충북대학교, 컴퓨터공학과 교수
관심분야 : DBS, Real-Time Systems

Wolfgang A. Halang

1976 Dr., Ruhr-Univ. of Bochum, Mathematics
1980 Dr., Univ. of Dortmund, Computer Science
1992 Prof., Faculty of Electrical Eng. FernUniv. of Hagen
1985 R&D Dept., Coca-Cola GmbH
1987 Prof., King Fahd Univ. of Petroleum & Minerals

1988 Visit. Prof., Dept. of Electrical & Computer Eng., Univ. of Illinois at Urbana-Champaign
1989 Process Control Div. of Bayer AG
1992 Prof., Dept. of Computing Science, Univ. of Groningen
Research Interests : Hard Real-Time Systems; innovative and function oriented architectures, application specific peripheral components for process control computers, predictable system behaviour, rigorous software verification and safety licensing.

Coen Bron

Drs., Univ. of Utrecht
1965 Assi. Prof., Dept. of Computing Science, Eindhoven Univ. of Technology
1973 Asso. Prof., Dept. of Computing Science, Twente Univ.
1983 Prof., Dept. of Computing Science, Univ. of Groningen
Research Interests : Software Engineering, OS, Compiler Construction