Exploiting a Graphical Programming Paradigm to Facilitate Rigorous Verification of Embedded Software

WOLFGANG A. HALANG, BERND KRÄMER AND LESZEK TRYBUS[1]

Faculty of Electrical Engineering, Fern Universität, D-58084 Hagen, Germany
[1]*Department of Electrical Engineering, Rzeszów University of Technology, PL-35-959 Rzeszów, Poland*
Email: wolfgang.halang@fernuni-hagen.de, bernd.kraemer@fernuni-hagen.de, zaiprz@plumcs11.umcs.lublin.pl

A computing architecture enabling economical safety licensing of software embedded in safety-critical technical systems is defined. The architecture relies on mature methods and technology only. In particular, it includes a highly ergonomic but rigorous validation method, called diverse back translation. For safety-related program controlled electronic systems, safety licensing of software is extremely critical, since it is far from being as dependable as hardware. The presented approach deviates from classical construction and validation techniques by enforcing the re-use of pre-engineered and verified off-the-shelf application-oriented standard software function modules and by employing a graphical programming paradigm.

1. INTRODUCTION

In society, there is growing concern for safety and environmental issues producing an increasing demand for dependable technical systems preventing loss of human lives and environmental disasters. To enable the flexible adaptation of system functions to new needs and to enhance the productivity of system development processes, computer based systems are increasingly being applied for both control and automation functions under real time constraints. These systems have the special property that hardware and software are closely coupled to complex mixed-technology systems such as manufacturing systems, process or traffic control systems.

When it comes to dependability evaluation, hardware turns out to be subject to wear. Faults occur at random and may be of a transient nature. To a very large extent, these sources of non-dependability can successfully be coped with by applying a wide spectrum of redundancy and fault tolerance methods. Software failures, on the other hand, are neither caused by wear nor by environmental events such as radiation or electric impulses. Instead, all errors are requirements analysis, design or programming errors, i.e. of systematic nature, and their causes are always (latently) present. Dependability of software cannot be achieved by reducing the number of errors it contains close to zero by testing, reviews, or other heuristic methods, 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 in general cases one has to yield to the complexity problem, in this paper we shall show a workable way of safety licensing by exploiting the intrinsic properties of a special, but not untypical and from the viewpoint of safety engineering very important case, that was identified in industrial control problems. Here complexity turns out to be manageable, because the application domain demands software of limited variability only. Moreover, this software may be implemented in a well structured way by graphically interconnecting carefully designed and rigorously verified "software ICs".

There are already a number of established methods and guidelines, such as [1], which have proven their usefulness for the development of high integrity software employed for the control of safety-critical technical processes. Prior to its utilization, 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. For, object code must be considered in 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 customized computer control system, which supports diverse back translation, was developed. Diverse back translation is a software verification method whose utilization in the traditional context turns out to be extremely tedious and time

consuming. Licensing authorities, for example, report up to 2 person months' inspection effort for just 4 kbytes of machine code [2]. Based on a graphical programming paradigm and the reuse of validated standard building blocks for software, however, diverse back translation becomes very easy, economical and time efficient. The presented approach is unique in providing support for software verification already in the architecture. The leading idea followed throughout this design was to combine mature software engineering and verification methods with architectural support. Thus, the semantic gap between software requirements and the capabilities of the execution environment could be closed, thus relinquishing the need for non-safety-licensable compilers and operating systems.

2. EVALUATION OF SOFTWARE VERIFICATION METHODS

Testing in all its varieties and peer reviews are by far the most widely used methods for software verification. Although being quite effective means of detecting errors, in general these methods are not capable of certifying correctness. Often testing is not exhaustive enough, because of the large or even infinite number of cases to be covered.

Formal verification techniques are more and more accepted as an important approach to achieve reliable software, particularly in security and safety-critical application domains. They use mathematical techniques to verify the correctness of software rigorously. This, however, is also the reason for their major drawback: the application of formal verification techniques requires special expertise, and the usually rather lengthy program correctness proofs may contain errors, which may remain undetected by peer review and may survive for long times.

Diverse back translation is a software verification method developed in the course of the Halden experimental nuclear power plant project by TÜV Rheinland [3]. Owing to the requirements mentioned in the introduction, its main idea consists of reading machine programs out of computer memory and delivering them to a number of different review teams working without any mutual contact. By hand, these teams disassemble and decompile the code, from which they finally try to regain specifications. A safety licence is granted to a software if its original specification agrees with all inversely obtained respecifications. 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 safety requirements and the usual machine instructions implementing it. On the other hand, diverse back translation does not require expert knowledge. This is especially important taking the legal quality of safety licensing into consideration, because judges and juries did not need to rely on expert advice, but could draw their own conclusions in related law suits.

3. SOFTWARE FOR PROGRAMMABLE CONTROLLERS

Since the early 1970s, when predecessors of today's programmable controllers appeared on the market, the graphical programming languages Ladder Diagram (LD) and Function Block Diagram (FBD), as well as the textual languages Instruction List (IL) and Structured Text (ST) have established themselves in a large number of varieties along with several versions of Sequential Function Chart (SFC) languages. Ladder diagrams employ contacts and coils, instantaneous or time-lagged, as familiar from relay rung circuits. Function block diagrams look like networks consisting of blocks, which are software equivalents of conventional control elements and devices. The textual languages prevail in the area of programming control computers. Whereas instruction lists resemble assembly language programs, structured texts are dedicated high level languages. Sequential function charts are used on the level of program organisation, i.e. on a higher level than the other languages, and enable to formulate sequence oriented programs using steps, transitions, and actions as expressive elements.

Present day programmable controllers can be divided into two classes, namely, programmable logic controllers (PLC) and multifunction controllers (MFC). Due to their short scan time, say 10 ms, PLCs are used in automated manufacturing where logic control prevails, whereas MFCs having much longer scan times, roughly 0.5 s, are employed for process control and continuous regulation. Both PLCs and MFCs are now used by people of rather different skills, viz. workers, technicians, and engineers. Since they are easily understood by all these groups, the graphical languages, LDs and FBDs, are most common in practice, with LDs having turned out typical for PLCs, whereas FBDs prevail in MFCs. However, more and more PLCs are now also equipped with FBDs to simplify programming. The FBD language requires little training and provides real flexibility, by allowing only slightly modified software to be used in different applications.

A function block (FB) is a programming unit which, when executed, yields one or more values. It may have a memory for some internal or transient data. Instances of a block may be created to assure reusability. Certain blocks may be tied to a controller's resources such as sensors, actuators, communication channels, or a man–machine interface. They represent I/O blocks in FB diagrams. The blocks not tied to resources are called algorithmic. Normally, they constitute the majority in a diagram. Depending on the complexities of their algorithms these blocks are divided into simple and complex. The former include arithmetic operations, logic gates, switches, comparators, flipflops, memory cells and the like. Among complex blocks one can typically find PID controllers, servopositioners, ladder rungs, truth tables and sequencers.

Programming in any FBD language is straightforward.

One must select appropriate resource blocks, create instances of algorithmic blocks required by the control strategy, make connections (a net list), and specify the execution order. The whole procedure closely resembles the traditional assembling of control systems from hardware elements and devices, i.e. wiring analogue regulators, operational amplifiers, electromagnetic relays, TTL elements etc. (see e.g. [4] or any other text on classical control equipment). Here, however, the FBs are "elements and devices". Programming of typical controllers proceeds in a conversational manner through a series of questions and answers. Some CAD packages have recently become available, which create FB diagrams on a screen and then transfer the resulting net lists to controller memory.

Owing to the reusability of FBs only the net lists must be changed if one moves from one application to another. Already tested blocks diminish the need for developing new software. The design process becomes faster and costs are reduced. Of the four groups of control languages, FBDs favour reusability the most. As we shall see in the sequel, this feature is crucial to the software licensing problem.

The set of function blocks available in a particular controller depends on its application area. Automated manufacturing based on PLCs focuses on logic and sequence operations, so ladder rungs, truth tables, sequencers with interlocks, decoders, multiplexes, counters, etc. are typical FBs. Here integer arithmetic is sufficient. The blocks provided by MFCs for process control include PID algorithms, servo-positioners, filters, set-point drivers, clock programmers, etc. Here floating point numbers are necessary. The sets of FBs offered by leading manufacturers are "locally complete" in the sense that they are capable of solving virtually all individual problems which occur in a particular application area, taking available resources, scan time and memory (net list) size into account. The "virtually all" statement is a matter of experience grown over the years from numerous applications. The FBs available now in Honeywell's TDC, Siemens's AS, Bailey's INFI or Foxboro's SA control systems are typical examples.

Within this framework, a large number of manufacturer dependent FB sets was developed in the past. An analysis reveals two opposite extremes which specify the range of approaches taken, i.e. either a small number of highly sophisticated FBs, or a large number of quite simple FBs. The FBs of all Honeywell TDC systems are examples of the first approach. The recent TDC 3000 Process Manager has only five types of FB (slots): digital composite, logic slot, process module, regulatory PV and regulatory control. Each type admits selection of a specific algorithm with various parameters, whose number sometimes exceeds 50. Similarly sophisticated FBs are typical for big systems, including Siemens's AS, Bailey's INFI, and Foxboro's IA. Not surprisingly, simple FBs are the domain of low cost automation, i.e. instruments sized according to DIN standards and small PLCs. Here the blocks have at most a few inputs, and none or up to 10 parameters. The FBs of Hartmann and Braun's Digital P-controllers and of Moore's MYCRO instruments belong to the most simple. Somewhat more complex are those of Siemens's SIPART multifunction unit and SIMATIC S5-90/95U PLCs.

It is obvious that a set of sophisticated FBs requires extensive training. However, the user's prior background in control systems can be limited, since sophisticated FBs take into account all details anyway. The other extreme, although attractive at first glance, requires a much better background, since the user him- herself has to develop reasonable control strategies using very simple components (blocks). Such non-compatible solutions to automation problems, apart from hardware-related peculiarities, effectively tie the user to a particular manufacturer. Hitherto, FB sets were not uniform and, hence, application software was not portable among different controllers.

A problem which has steadily been gaining importance is (fast) logic control and (slow) continuous regulation performed by a single controller, because installations where manufacturing strongly interacts with technological processes have become more and more common. MFCs are capable of handling both logic and regulation, however, at the expense of relatively long scan times. On the other hand, PLCs cannot perform many floating point calculations, because the scan time would quickly exceed acceptable limits. Hence, so far the two tasks have to be handled jointly by PLCs and MFCs communicating over peer-to-peer channels. The development of such systems is time consuming and costly, single controllers would be much more appropriate. It is clear, however, that the FB sets of such controllers would have to be "globally complete" to solve the problems in each application area, which means blocks and arithmetic functions suitable both for automated manufacturing and process control.

4. A RE-USE ORIENTED SOFTWARE ENGINEERING PARADIGM

To solve the problems of software reusability, portability and global completeness, an international committee was created, which in 1992 produced the international standard [5]. The third part of it deals with languages for programmable controller. As until now, the IEC 1131-3 languages are graphical, Ladder or Function Block Diagrams, and textual, Instruction List and Structured Text, with, if necessary, supervisory functions formulated as a Sequential Function Chart. The IEC 1131-3 deals with modularization and structured programming, defines the languages' scope for increased portability, and provides integration into a globally complete environment. The benefits of uniform programming are particularly important for the users, who will now need one period of training and familiarization only, and work more efficiently and economically by not being tied to particular manufacturers. The software

developed will generally be used again. The IEC 1131-3 represents a leap forward in the quality of control programming languages.

Three leading manufacturers have already been able to develop controllers with software conforming to IEC 1131-3. These are Eurotherm's PC3000 Production Process Controller, Moore's Advanced Process Automation and Control System, and Philips's P8 Automation System. One can expect that the other companies will follow soon. It is only a question of time until the IEC 1131-3 languages will be taught in college and university courses.

Standardization 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 blocks suitable to formulate in the graphical FBD language as defined by the international standard IEC, 1131-3, the large majority of the automation problems [6] occurring in chemical engineering. The following list of function modules[1] defined in the guideline VDI/VDE 3696 gives an impression of typical functionalities:

monadic mathematical functions: absolute value, cosine, sine, exponential function, natural and base 10 logarithms, square root, limiter, non-linear static function, linear scaling;

polyadic mathematical functions: addition, subtraction, multiplication, division, modulo, exponentiation;

comparisons: equal, greater or equal, greater, less or equal, less, not equal;

monadic Boolean function: negation;

polyadic Boolean functions: conjunction, disjunction, antivalence;

edge detectors: falling and rising edge detection;

selection functions: maximum and minimum selections, selection by 1 out of N bits, demultiplexers for Booleans and numbers, multiplexers for Booleans and numbers, binary selections of a Boolean or a number;

counters, monostables, bistables, timers: up/down counter, flow counter, bistable elements with set/reset dominance, pulse duration modulator, on/off delays, non-re-triggerable monostable element;

process input/output: analogue input/output, binary input/output, digital word input/output, impulse input;

network communication input/output: communication input/output for a Boolean or numerical value;

dynamic elements and regulators: universal and Standard controllers (PID-T1), running time average, dead time, differentiation with lag (D-T1), integrator (I), lead lag (PD-T1), 2nd order;

conditioning for display and operation: limit switch with alarm or message storing, alarm or message storing, trend registration, manual value entry with switch and limitation.

Written in the IEC 1131-3 high-level language Structured Text, these software modules are usually quite short; their source code does not exceed two pages. Unbounded iteration and recursion do not occur in these modules. Therefore, their correctness can be formally proven with bearable effort, e.g. using predicate calculus, but also symbolic execution or, in some cases such as Boolean functions, even complete test.

In order to give another typical example, consider the programming of emergency shut-down systems. Functional logic diagrams, which describe the corresponding mappings from Boolean inputs to Boolean outputs as functions of time, require as few as only four function modules, viz., the three basic Boolean operators and a timer, to be combined.

The above mentioned analysis of process automation suggests the introduction of a new programming paradigm, viz., composing software out of high-level user oriented and reusable building blocks instead out of low-level machine oriented instructions. Whereas a single machine instruction taken out of a program context does not reveal its purpose, the occurrence of a certain function module instance usually gives a clue to the problem, its solution and the module's rôle 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 function 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 to result in diagrams that look like dataflow diagrams used for functional or activity modeling. The boxes in the function block diagram depicted in Figure 1 represent program activities, while lines model unidirectional flow of typed information necessary for function blocks to carry out these activities. Individual function blocks are invoked according to the partial ordering given by the "wiring" and, in the course of this, they pass data along their connecting lines. Lines may branch to represent fan-out, i.e. multiple identical copies of data can be delivered coincidently to other function blocks acting as consumers of those data. But lines may not join in the sense of fan-in or merging of incoming data.

Besides the provision of constants such as `bar` provided at external input `XUNIT` of function block `B1` in Figure 1, the function blocks' instances and the data flows between them are the only language elements used on this programming level. Software development is carried out in graphical form using an appropriate CAD tool. Once a diagram is satisfactory, 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

[1] For an in-depth treatment of the function module concept we refer to [7].

other features than sequences of procedure calls and some internal data moves.

The fundamental feature of this paradigm, which facilitates managing the complexity of large systems, is that diagrams can be "canned" into single function blocks which, henceforth, may occur in other diagrams as more abstract and functionally more complex, but on a higher level application oriented components. Thus, the interconnection complexity of single diagrams can always be kept low. Strict rules control the correct hierarchical (de-) composition of diagrams [8].

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. Application software may be safety licensed by subjecting its loaded object code to diverse back translation. Employing the programming paradigm of basic function modules, specifications are directly mapped on to sequences of procedure invocations. The invoked procedures implement the functionality of individual function modules, which are drawn from a library and which are verified only once during their lifetime using both specification and program verification techniques. Therefore, the object code derived from a diagrammatic program only consists of procedure calls and parameter passing code. It takes only minimum effort to interpret such code complementing just module interconnections, and to reconstruct graphical program specifications from it. If the implementation details of the function modules used are part of the firmware, they remain invisible from the application programming point of view and do not require safety licensing in this context.

Using diverse back translation for software verification is further 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 utilization is by orders of magnitude less than in the cases reported by [9]. Furthermore, the employed principle of reuse-oriented software engineering reduces the number of possibilities to solve a given problem in different ways. Therefore, it becomes considerably easy to check the equivalence 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.

5. A SAFETY-ORIENTED ARCHITECTURE

As architecture for a safety-oriented computer control system we select a standard microcomputer endowed with a firmware interpreter implementing a set of basic function blocks. For the envisioned purpose of executing software represented in form of function block diagrams the interpreter is required to perform just two instructions on the user programming level:

- GET < operand-address > and
- PUT < operand-address > .

A function block invocation gives rise to an object program consisting of a number of parameter fetches from RAM or ROM locations carried out with GET instructions and a number of result storage operations with the help of PUT instructions. First, the interpreter fetches the identification of a function block to be invoked. From this the appropriate number of input parameters and, if need be, also of the block's internal state variables is derived, which are then fetched.

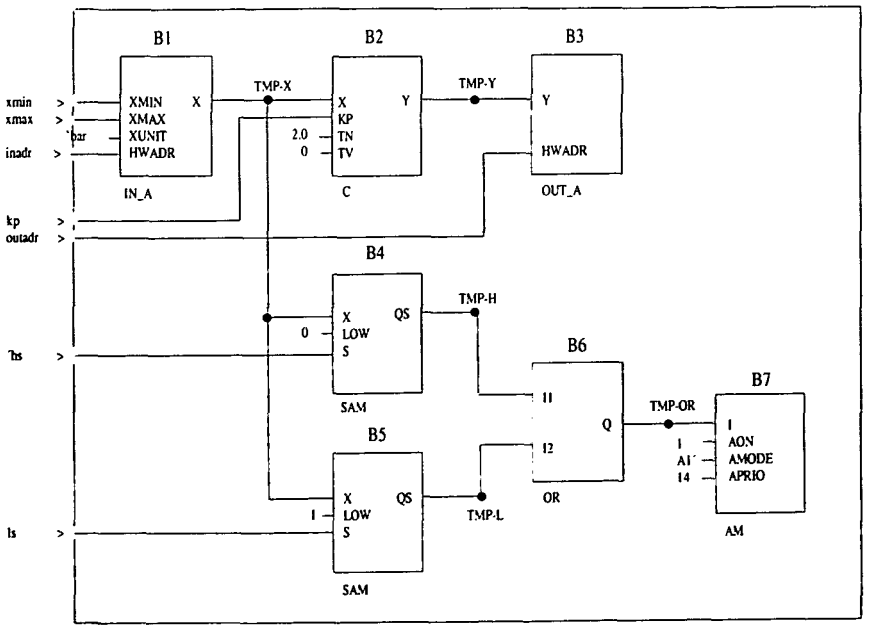

FIGURE 1. A graphically formulated FBD-program for pressure control and supervision.

Subsequently, the object program implementing the function block is executed. In the course of this, the interpreter performs all necessary data manipulations and communications with the environment. The elaboration of the function block ends with storing the results and new internal states generated into RAM.

In order to prevent any modification by a malfunction, in this safety-oriented architecture all programs must be provided in read only memories. For practical reasons, there will generally be two types of memory. On the other hand, there is no program RAM at all. The code of the basic function modules resides in mask-programmed firmware 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 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. This program memory configuration is chosen to physically separate two software parts from one another: one with general scope which only needs to be verified once, and the other one being application specific.

6. SOFTWARE SAFETY LICENSING

All elements of an employed function block set contained in a library are first verified with appropriate formal methods. Note that this rather costly safety licensing needs to be carried out only once, after a certain function block set has been identified and standardized for a given application area. The licensing costs can thus be spread over many implementations leading to relatively low costs for each single automation project. 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 processed by the interpreter to diverse back translation, because CAD tools used for graphical programming contain 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 the machine (provided by the interpreter), diverse back translation leads back in one easy step from machine code to problem specification.
- For our architecture, the effort required for the utilization of diverse back translation is by several orders of magnitude smaller than for the regular von Neumann architecture not enhanced with an interpreter for application specific firmware.

As already its name implies, diverse back translation 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—human—licenser performs the back translation as well, compares his or her results with those 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 licenser, who is always indispensable to take the legal responsibility for issuing a safety licence.

7. A WORKED EXAMPLE

We want to illustrate the application of back translation by working out a relatively simple, but realistic example. The different representation levels of a program, viz., Function Block Diagram, net list and object code for the interpreter in our architecture, are shown in full detail. It will become evident that it is straightforward and very easy to draw a function block diagram from a given object program establishing the feasibility of back translation as a software verification method.

Figure 1 shows a typical industrial automation program in graphical form. It performs supervision and regulation of a pressure. The program is expressed in terms of standard function blocks as defined in the guideline [6]. An analogue measuring value, the controlled variable, is acquired by a function block of type `IN_A` from the input channel with address `INADR` and scaled within the range from `XMIN` to `XMAX` to a physical quantity with unit `XUNIT`. The controlled variable is fed

into a function block of type C performing proportional-integral-differential (PID) regulation subject to the control parameters KP, TN, and TV. The resulting regulating variable is converted to an analogue value by a type OUT_A output function block and switched on to the channel addressed by OUTADR. In addition, the controlled variable is also supervised, with the help of two instances of the SAM limit switch standard function block type, to be within the limits given by the parameters LS and HS. If the controlled variable is outside of this range, one of the QS outputs of the two SAM instances becomes logically true and, hence, the output of the type OR function block as well. This, in turn, causes the type AM alarm and message storing function block to create a timed alarm record. The inputs of the standard function blocks comprised by the program which are neither fed by externally visible inputs of the program itself nor internally by outputs of other standard function blocks are given constant values.

```
<<< Component List >>>

     IN_A     B1
     C        B2
     OUT_A    B3
     SAM      B4
     SAM      B5
     OR       B6
     AM       B7


<<< Wire List >>>

 NODE  REFERENCE   PIN #   PIN NAME   PIN TYPE   PART TYPE

[00001] N00001
       B1          5       X        Output     IN_A
       B2          1       X        Input      C
       B4          1       X        Input      SAM
       B5          1       X        Input      SAM

[00002] N00002
       B2          5       Y        Output     C
       B3          1       Y        Input      OUT_A

[00003] N00003
       B4          4       QS       Output     SAM
       B6          1       I1       Input      OR

[00004] N00004
       B6          3       Q        Output     OR
       B7          1       I        Input      AM

[00005] N00005
       B5          4       QS       Output     SAM
       B6          2       I2       Input      OR

[00006] XMIN
       B1          1       XMIN     Input      IN_A

[00007] XMAX
       B1          2       XMAX     Input      IN_A

[00008] 'BAR'
       B1          3       XUNIT    Input      IN_A

[00009] '2'
       B2          3       TN       Input      C

. . .
```

FIGURE 2. Net list representation of the example program.

The net list representation of the above example program as generated by a utility program of the OrCAD schematic capture tool is shown in Figure 2. Net lists constitute textual representations which are fully equivalent—but for geometrical aspects—to the original drawings.

The object code for the interpreter finally obtained by automatic translation of the example program's net list representation is listed in Figure 3. It shows a (readable) assembly language version. Of the different function block types instantiated in the example, C, SAM, and AM have internal state variables, viz., C has 3 and the other two types have 1 each.

The object code listed in Figure 3 illustrates that all function block instance invocations occurring in a program are directly mapped onto procedure calls. Each of them commences with a GET instruction which transfers the identification (e.g. ID-C) of the corresponding block out of an appropriate ROM location to the interpreter. Then, the input parameters are supplied by reading appropriate ROM (for constants) or RAM (for program parameters and intermediate values) cells. Finally, if there are any, the values of the procedure's internal state variables are read from appropriate RAM locations. There is a set of correspondingly labelled (e.g. RAM-loc-B2-isv*i*) locations for each instance of a function block with internal states. When the interpreter has received all this information, it executes the procedure and returns, if there are any, values of output parameters and/or internal state variables, which are then stored into corresponding RAM locations. A connection between an output of one function block and an input of another one is implemented by a PUT and a GET instruction: the former storing the output value in a RAM location for a temporary value (e.g. TMP-X), and the latter loading it from there. In other words, each node in a net list gives rise to exactly one transfer from the interpreter to a RAM cell, and to one or more transfers from there to the interpreter. The implementation details of the various procedures are part of the architecture's firmware and, thus, remain invisible.

According to the above described structure of the

```
GET ROM-loc-ID-IN_A        GET ROM-loc-ID-OUT_A       GET ROM-loc-ID-OR
GET RAM-loc-XMIN           GET RAM-loc-TMP-Y          GET RAM-loc-TMP-H
GET RAM-loc-XMAX           GET RAM-loc-OUTADR         GET RAM-loc-TMP-L
GET ROM-loc-BAR                                       PUT RAM-loc-TMP-OR
GET RAM-loc-INADR          GET ROM-loc-ID-SAM
PUT RAM-loc-TMP-X          GET RAM-loc-TMP-X          GET ROM-loc-ID-AM
                           GET ROM-loc-0              GET RAM-loc-TMP-OR
GET ROM-loc-ID-C           GET RAM-loc-HS             GET ROM-loc-1
GET RAM-loc-TMP-X          GET RAM-loc-B4-isv         GET ROM-loc-A1
GET RAM-loc-KP             PUT RAM-loc-TMP-H          GET ROM-loc-14
GET ROM-loc-2.0            PUT RAM-loc-B4-isv         GET RAM-loc-B7-isv
GET ROM-loc-0.0                                       PUT RAM-loc-B7-isv
GET RAM-loc-B2-isv1        GET ROM-loc-ID-SAM
GET RAM-loc-B2-isv2        GET RAM-loc-TMP-X
GET RAM-loc-B2-isv3        GET ROM-loc-1
PUT RAM-loc-TMP-Y          GET RAM-loc-LS
PUT RAM-loc-B2-isv1        GET RAM-loc-B5-isv
PUT RAM-loc-B2-isv2        PUT RAM-loc-TMP-L
PUT RAM-loc-B2-isv3        PUT RAM-loc-B5-isv
```

FIGURE 3. Object code representation of the example program.

interpreter's object programs, the process of back translation—disassemble and decompile object code—turns out to be very easy. To perform back translation, the first GET instruction is interpreted. It identifies a function block to be drawn into the function block diagram to be set up. By comparing the subsequent GETs with the function block's description contained in the library used, the correct parameter passing can be easily verified. Moreover, for each such GET which corresponds to a proper parameter (and not to an internal state variable) a link is drawn into the diagram. There are two kinds of links. The first one are connections from program inputs or constants to inputs of function blocks or from function block outputs to program outputs. The second kind are, so to speak, half connections, namely, from function block outputs to named connection points in the diagram, i.e. net list nodes, or from such points to function block inputs. When the diagram is completely drawn, the names of these points can be removed. With respect to the internal state variables it needs to be verified that the corresponding RAM locations are correctly initialized and that the new values resulting from a function block execution are written to exactly the same locations from where the internal states were read in the course of the block's invocation. The process of function block identification, parameter passing verification, and drawing of the block's symbol and of the corresponding connections is repeated until the end of the object code is reached.

8. EXPERIENCES AND RESULTS

We have built a prototype of a computerized controller according to the architecture outlined above and implemented a function block interpreter performing Boolean negation, conjunction and disjunction as well as time delays. Whereas the first three of these functions are trivial, the delay timer required a formal correctness proof, which was carried out employing HOL and which turned out to be rather laborious and lengthy [10]. The controller's utilization in practice showed that implementing the functionality of a hard-wired emergency shut-down system with a programmable electronic system is feasible, and that the programming paradigm based on formally verified function modules and on application programs verified by diverse back translation can render error free software. The latter, together with a fault-tolerant hardware platform, allows programmable safeguarding systems sharing the fail safe feature with well established hard wired solutions to be implemented.

For ergonomic reasons, complex software should be hierarchically structured in such a way, that the formulations of its modules in form of function block diagrams always fit on one page or a terminal screen. As outlined before, these modules occur on the next higher level of a hierarchical software structure themselves as function blocks, whose interna are abstracted away and which are executed by the interpreter described. Hence, the complexity of a function block diagram will never be greater than approximately ten times the one of the example as given in the preceding section, which results in a verification effort of at most a few hours' work (for any particular function block diagram).

9. CONCLUSION

Economical considerations impose stringent boundary conditions on the development and utilization of technical systems. This holds for safety related systems as well. Since manpower is becoming increasingly expensive, 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 systems.

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 realize 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 utilization in such applications is going to increase considerably—the problem of software dependability will multiply severely.

In the situation as outlined above, this paper addresses a pressing problem. It does not present a solution to all open questions in software safety licensing, 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 will ultimately lead to the replacement of discrete or relay logic by programmable electronic systems executing safety-licensed high integrity software in charge 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.

Our approach particularly observes ergonomic criteria in the design of suitable software technology and an adequate separation of human and computer functions. The construction and validation methods, for instance, rely on a graphical programming paradigm featuring design by reuse of prefabricated and certified components, thus serving simplification and unification without restricting design scopes. Related support tools are interactive, provide graphical user interfaces, and include specification based prototyping capabilities (e.g, the prototype presented in [8]). Once the graphical paradigm stressed here is accepted by system developers and licensing authorities, only minimum organizational

measures should be required to introduce our method, as it relies on traditional technology, such as diverse back translation, but reduces the complexity of its use.

REFERENCES

[1] IEC International Standard 880 (1986) *Software for Computers in the Safety Systems of Nuclear Power Stations*, International Electrotechnical Commission, Geneva.

[2] Pofahl, E. (1994) TÜV Rheinland, Private communication.

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

[4] Hunter, R. P. (1978) *Automated Process Control Systems: Concepts and Hardware*, Prentice Hall, Englewood Cliffs, NJ.

[5] IEC International Standard 1131-3 (1992) *Programmable Controllers, Part 3: Programming Languages*, International Electrotechnical Commission, Geneva.

[6] VDI/VDE Draft Guideline 3696 (1992) *Vendor Independent Configuration of Distributed Process Control Systems*, Beuth-Verlag, Berlin.

[7] Zöller, H. (1991) *Wiederverwendbare Software-Bausteine in der Automatisierung*, VDI-Verlag, Düsseldorf.

[8] Halang, W. A. and Krämer, B. (1992) Achieving high integrity of process control software by graphical design and formal verification *IEE/BCS Software Engineering Journal*, **7**, 53–64.

[9] Dahll, G., Mainka, U. and Märtz, J. (1988) Tools for the standardised software safety assessment (The SOSAT Project), in *Safety of Computer Control Systems*, edited by W. D. Ehrenberger, Pergamon, Oxford, (1988).

[10] Halang, W. A., Krämer, B. J. and Völker, N. (1995) 'Formally verified building blocks in functional logic diagrams for emergency shutdown system design', to appear in *High Integrity Systems*.