A fail safe programmable logic controller

Marek Śnieżek a,*, Josef von Stackelberg b

a Faculty of Electrical and Computer Engineering, Rzeszów University of Technology, 35-959 Rzeszów, Poland
b Faculty of Electrical and Computer Engineering, Fern Universität, 58084 Hagen, Germany

Abstract

To architecturally support the programming of safety related control applications in the graphical language function block diagram and the verification of such software meeting the requirements of Safety Integrity Level SIL 3, a dedicated, low complexity execution platform is presented. Its hardware is fault detecting to immediately initiate emergency shut-downs in case of malfunctions. With their low processing speeds, currently available fail safe comparators constitute bottlenecks and, in case of malfunctions, do not distinguish between comparison errors and comparator errors. To solve these problems, a novel fail safe comparator of two binary inputs is presented, which does not only indicate a result, but also its status. Built in a modified CMOS technology, it can match the execution speed of digital computers. In contrast to all earlier designs of fail safe comparators, by employing ternary logic it provides three different output values, allowing to distinguish between the three indications “inputs equal and comparator working properly,” “inputs unequal and comparator working properly” as well as “comparator malfunctioning.” By design, there is no semantic gap between the programming and machine execution levels of the controller, enabling the safety licensing of application software by extremely simple, but rigorous methods, viz., diverse back translation and inspection. Operating in a strictly periodic fashion, the platform exhibits fully predictable real time behaviour.

© 2003 Elsevier Ltd. All rights reserved.

Keywords: Safety related control; Safety Integrity Level 3; Function block diagrams; Software verification; Programmable logic controller; Fail safe comparator; Ternary logic; Result and status indication

1. Introduction

A workable programmable electronic system for use in industrial automation is presented, which can be licensed for the higher Safety Integrity Level 3 in full, i.e. not only its hardware, but also its software, which is more important as software dependability still needs to catch up with the one already achieved by hardware. The leading idea followed throughout its design, and the originality of the approach, was to select programming and verification methods of utmost simplicity and, hence, highest trustworthiness, and to custom tailor an execution platform for them. Thus, the semantic gap between software requirements and hardware capabilities is closed, relinquishing the need for not safety licensable compilers and operating systems.

2. Function block diagrams

A programming paradigm allowing for software to be easy to grasp and to verify with respect to both source and object code is available in form of the graphical language function block diagram (FBD) defined within the standard IEC 61131-3 (IEC, 1992). With its long tradition in control engineering, graphical programming in form of function block diagrams as depicted in Fig. 1 is already well established in automation technology.

The function block diagram language consists of four different structural elements, only:

(1) instances of functions and function blocks, i.e. rectangular symbols,
(2) dataflow lines, i.e. connection lines,
(3) names, i.e. identifiers, and
(4) (external) connection points.

Functions and function blocks are highly application-dependent and re-usable elementary units of application programming on a higher abstraction level. In principle,
they are subroutines having inputs and outputs of arbitrary data types, and are able to perform arbitrary processing functions. Functions do not have any internal states. After being executed they yield exactly one data element as a result, which may be multivalued. Multiple named instances, i.e. copies, can be created of function blocks. Each instance possesses an associated designator and a data structure, which contains its output and internal variables as well as possibly its input variables. All values of the output variables and the internal variables in such a data structure persist from one execution of a function block instance to the next. Therefore, invocation of a function block with the same arguments does not necessarily yield the same output values. This is necessary to be able to express feedback and internal storage behaviour. Only the input and output values are accessible outside a function block instance, i.e. a function block’s internal variables are hidden from the outside and are, thus, strictly protected. By the connection lines within a function block diagram a data flow is represented.

Graphical, system independent program development in form of function block diagrams is very easy, and takes place in two steps:

1. only once building a library of functions and function blocks, and
2. application specific interconnection of functions and function block instances.

Many automation programs including safety related ones have the form of sequence controls composed of steps and transitions. While in a step, an associated program, called action, represented as a function block diagram is being executed. For safety related applications, linear sequences of steps and alternative branches of such sequences as shown in Fig. 2 are permitted, only. Parallel branches in sequential function charts must either be implemented by hardware parallelism or already resolved by the application programmer in form of explicit serialisation. Also, for clarity as well as for easy comprehension and verification only non-stored actions may be used. All other types of actions as defined in IEC 61131-3 can be expressed in terms of non-stored ones and reformulated sequential control logic.

3. An execution platform

When designing an execution platform as closely matching and supporting software represented in form of function block diagrams and sequential function charts, it was not the objective to save hardware costs, but to facilitate the understandability of object programs and their execution process.
This led to the architecture depicted in Fig. 3 with, conceptually, two different processors: a control flow processor (master) and a function block processor (slave). These two processors are implemented by separate physical units. Thus, a clear and physical separation of concerns is achieved: execution of function blocks in the slave processor, and all other tasks, i.e., execution control, sequential function chart processing, and function block invocation, assigned to the master. This concept implies that application code is restricted to the control flow processor, on which project specific tasks, i.e., execution control, sequential function chart processing of function blocks in the slave processor, and all other activities, after the latter have established the correctness of the modules and their translation 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, can be written into EEPROMs 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 EEPROMs in the target systems.

Besides program memory, the masters’ address spaces also comprise RAM memory and FIFO-input/output registers, command registers, two step related registers each, viz., step identifier and step initial address, and transition condition registers. Furthermore, there are program counters and single bit step-clock-occurred registers, which are not programmer accessible. Additionally, in the masters’ address spaces other units are memory mapped to create and receive control signals for the access of ROM, RAM, and FIFO-queues. To fulfill their purpose, the master processors need just two instructions, viz., MOVE and STEP. The MOVE instruction has two operands, which directly point to locations in address space. Thus, the memories and the above mentioned registers can be read and written. A read from a FIFO-input register implies that the processor has to wait when the input FIFO-queue register is empty. In case of writing into an output FIFO-queue register, the processor also has to wait when the register is full. Execution of a MOVE implies program counter incrementation.

The programs executed by the master processors consist of sequences of steps. Behind the program segment of each step a STEP instruction, with a next-step-address as operand, 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 operation as programmable logic controller. 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). If the execution of a segment does not terminate within a step cycle, an error signal is generated, which indicates an overload situation or a run time error. Then, program execution is stopped immediately, and suitable error handling

---

Fig. 3. Block diagram of fault detecting masterslave PES.
behaviour is only important for input and output operations, temporal predictability is achieved as follows. Input data are read by the drivers at the beginning of each cycle and stored en bloc in two independent RAM buffers assigned to the respective slaves. The cycle start is signalled by the step-clock-occurred register. Only after that, the data are made available for further processing, thus providing predictability in timing. Output data generated by the slaves are latched in registers before the end of every cycle. When the step-clock-occurred register is set, the data are first checked for equality in fail safe comparators and, subsequently, they are transferred to output ports to become effective to the environment. However, if output bytes are not identical, an error signal is generated leading to a system stop.

The FIFO-queue and output comparators mentioned above are the components of a global comparator unit, which also receives operation monitoring signals from processor watch-dog timers and some correctness signals from other units. Based on all these, a global correctness signal, in other words, a negated global error, is generated and fed back to all units of the programmable electronic system. Naturally, each one can operate if this signal indicates “no error.” Otherwise, the system stops and the outputs are set to safe states. The global error signal is also output, allowing to trigger some external hardware as well.

4. State of the art of industrial fail safe modules

Semiconductor-based computer technology cannot offer solutions matching the multitude of applications found elsewhere. Primarily, this is due to the fact that the classical control components switches, relays, thermostats etc. have high probabilities of assuming a certain “natural” switching state in case of failure, namely the state of disconnection. Therefore, it is impossible to associate an unambiguous error state with a semiconductor component. One way to solve this problem is to use redundant controller structures. In technical applications this means that control components are replicated to fulfill the same task. Furthermore, switching to a state other than a safe one is subject to monitoring with majority or unanimity voting. Controlled devices or processes are brought to safe states in case of disagreement.

A module is a unit capable of realising certain, mainly simple, logic functions (And, Or, Not). Fail safe modules are developed in special technologies which assure predictable behaviour of the outputs in case of any failure. Normally, such behaviour is equivalent to fall into the switched-off state. Galvanic separation of inputs and outputs is a common feature. There exist a few families of

is carried through by external fail safe hardware. Normally, however, segment execution terminates before the instant of the next step cycle signal. Then, the processors wait until the end of the present cycle period. When the clock signal finally occurs, the step-clock-occurred registers are set. According to the contents of the transition condition registers 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 counters are re-loaded from the step-initial-address registers, or if another segment’s initial program address is loaded from the STEP instruction’s operand called next-step-address. Since only one step is active at any given time, and since program branching is only possible in this restricted form within the framework of executing STEP instructions, this mechanism very effectively prevents erroneous access to code of other (inactive) steps as well as to program locations other than the beginnings of step segments.

The design objective for providing 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 arrive. 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 fast comparators placed into the FIFO-queues. Since the responsibility for detecting errors in the system rests on these comparators, they need to meet high dependability requirements and are, therefore, implemented in fail safe technology as described in the next but one section. A comparator is connected to two FIFOs’ outputs. 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.

Communication with external technical processes takes place through fault detecting input/output driver units 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, an error signal is generated leading to a system stop. To achieve full determinism of execution time behaviour, the basic cycle was introduced as maximum step execution period. Although it 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,
fail safe modules applying different principles of operation. HIMA Planar Logic (HIMA, 1991, 1992) and GTI MagLog 24 (GTI, 1993) are well established in the market.

Currently available fail safe logic families, of which comparator circuits can be built, exhibit far too long reaction times, viz., several milliseconds. In the controller presented here, however, the reaction time of a comparator must correspond to the speed of data transfer between master and slave, i.e., a few microseconds per single comparison. This eliminates HIMA or MagLog modules from consideration, creating the need to design and construct a sufficiently fast comparator.

Moreover, at their outputs, available comparators only provide two values signifying either (1) inputs equal and comparator properly working, or (2) inputs unequal or comparator malfunctioning, i.e., it is not differentiated between comparison errors and comparator errors. If these alternatives were distinguished, they could be handled in more specific ways. To solve these two problems of insufficient speed and indiscriminate output information, a novel fail safe comparator of two binary inputs was developed, which does not only indicate a result, but also its functional status. Built in a modified CMOS technology, it works in the MHz range and can, thus, match the execution speed of digital computers. In contrast to all earlier designs of fail safe comparators, by employing ternary logic, it provides three different output values, allowing to distinguish between the three indications:

1. inputs equal and comparator working properly,
2. inputs unequal and comparator working properly, and
3. comparator malfunctioning.

These signal states may be sensed and further processed by conventional semiconductor circuits without any problems.

5. Fast fail safe comparator with ternary output

Since the two families of industrial fail safe modules outlined above provide far too slow switching speeds as required by a feasible co-operation in environments characterised by integrated semiconductor circuits, a novel fail safe comparator (von Stackelberg & Halang, 1998) of two binary digits as depicted in Fig. 4 was designed. It combines high speed in the MHz range with a ternary output to indicate both comparison result and functional status. Consisting of altogether 37 individual modules, it is arranged in six stages with respect to functional operation, and four levels deep to implement fail safe behaviour. The individual modules are, to a certain extent, basic circuits of binary and ternary digital logic and, on the other hand, circuits especially designed for the purpose considered here. The comparator is operated with two voltages, viz., +1.5 and +5 V relative to reference potential. As a digital circuit, there are only the three states determined by the operating voltages and zero potential—intermediate states do not occur. As customary in digital electronics, these voltage levels are associated with

Fig. 4. Structure of the fail safe comparator.
logical signal states. In this special case, these are the ternary values:

- minus for the most negative voltage level, i.e. the reference potential 0 V,
- zero for the voltage level lying between the two extremes, i.e. for +1.5 V, and
- plus for the most positive voltage level, i.e. for +5 V.

As binary values are used:

- LOW for the more negative one of the two voltage levels applied to the circuit, which specifically may be 0 or +1.5 V, and
- HIGH for the more positive one, which specifically may be +5 V.

In the sequel, we shall use the terms mentioned above instead of the potentials at the inputs and outputs of the individual circuits and the comparator.

In the first stage of the comparator, there are four antivalence gates as shown in Fig. 5, with each of their inputs wired together. This means that the signals at the inputs E1 and E2 of the comparator are evaluated four times. Since the input signals come from binary circuits whose “LOW” and “HIGH” are represented by the voltages 0 and +5 V, the antivalence gates are also applied to these voltages. The result of such a gate is “LOW” if both inputs are the same, and “HIGH” if both inputs are different.

The resulting states “LOW” and “HIGH” are fed into four parallel ternary conjunction gates (Fig. 6) of the second stage. There, the “LOW” signal corresponds to “minus” and the “HIGH” signal corresponds to “plus.” The result of ternary conjunction is “minus,” if all of its inputs are “minus,” and “plus” if all of its inputs are “plus.” In all other cases, the outputs supply the signal “zero.” Since ternary negations are attached at the conjunction gates’ outputs of the second stage, the results of the third, fourth and fifth stages assume the following values: “plus” is yielded if both inputs of the comparator recognise the same signal, and different signals at the comparator’s inputs result in “minus.” Again, the four conjunction gates of the second stage are connected in parallel to have the result of the operation available four times.

In the third stage of the comparator four identity gates as depicted in Fig. 7 are arranged in parallel, too. The function of an identity element is just to convey the digital signal at its input to its output. However, since the comparator’s identity gates are attached to the +1.5 V potential with their negative supply voltage connections, and ternary signals are present at their inputs, there is a shift in signals: “plus” at the inputs becomes “plus” at the outputs, and both “zero” and “minus” at the inputs become “zero” at the outputs. The reason for this is to generate the ternary signals demanded at the output of the comparator from the binary signals at its input. Depending upon the comparator’s input signals, the following states result at the outputs of the identity elements: if both comparator inputs recognise the same signal, “plus” is present at the outputs of the identity elements of the third stage; if both comparator inputs recognise different signals, “zero” is generated. The “zero” signal from the second stage’s conjunction gates indicating a disturbance in the first stage of the comparator is evaluated in the fourth stage and is irrelevant on this signal path.

Switch functions (Fig. 8) are employed in the fourth stage of the comparator. A switch is a basic ternary circuit
switching the signal present at its “signal” input to its output in case it detects a “zero” signal at its “gate” input. If the input at “gate” is not equal to “zero,” “zero” is presented at the output. Since the switch gates used in the comparator have a ternarily negating output, a “minus” signal at their “signal” inputs is transformed to “plus” at the outputs when the “gate” inputs detect “zero.” Therefore, a “zero” generated at the output of a conjunction gate in the second stage leads to a “plus” at the output of the corresponding switch gate in the fourth stage.

In the fifth stage of the comparator there are three different circuits, a conjunction gate, four identity gates (Fig. 9) and differential voltage switches (Fig. 10) each. The conjunction gate’s task is to gather the signals produced in the third stage for the comparison states, and to direct them to the comparator’s output, which means that “plus” signal is presented at this output for the state “equality detected by comparison and comparator in proper order” only when all four identity gates of the third stage indicate “plus.” The identity gates of the fifth stage fulfill the same task as the identities in the third stage, and with each identity gate of the fifth stage the output signal of a conjunction gate in stage two is associated as well. The reason for this is to appropriately prepare the signals generated by the conjunction gates for the differential voltage switches connected behind the identity gates, as the differential voltage switches set, upon detecting different digital signals at their inputs, their outputs to the higher one of the two input signals, respectively.

Both outputs are set to “minus” for equal input signals. The differential voltage switches close a feedback loop between the comparator output and the conjunction gates of the second stage to check the output conjunction gate for proper functioning.

There is a resistor in the sixth stage of the comparator located between the output of the collective conjunction gate of stage five and the output of the comparator to provide correct measuring conditions for the differential voltage switches even when a threshold switch is in conducting state. The function of the threshold switches (Fig. 11) is to put the output of the comparator to “minus” reference potential if the input signal exceeds a given value relative to the reference potential. The inputs of the threshold switches are driven by the four outputs of the switches in the fourth stage and by the four times two outputs of the differential voltage switches of the fifth stage.

Fail safe behaviour of the comparator is achieved as a result of combining the following measures:

1. Each function is provided four times, viz., in four levels in stages one through six.
2. With their signal comparison between the second and the fifth stage, the differential voltage switches constitute a form of functional monitoring which provides a
plausibility check between the comparator and the result of the initial Boolean operation on its inputs.

(3) The modules with the differential voltage switches and the threshold switches are identically replicated four times, since their function is not monitored further. Hence, each device in each module is present four times. Devices whose low resistance state leads to the safe state of the comparator output signal are connected in parallel, while devices whose high resistance state represents the safe state for the comparator output signal are connected in series.

(4) By means of a resistor each input of each module is placed to that potential which constitutes the safe state for the comparator output if the corresponding controlling signal fails. This case could occur, for instance, when a previous output assumes high resistance state. These measures ensure that the comparator always provides either the correct, comparison result, or that it correctly signals the comparator to be out of order, which holds under the condition that not all four equal modules in one stage show the same error. This applies analogously with regard to the devices of the differential voltage switch and threshold switch modules, too.

6. Software safety licensing

Matching their development process, the verification of programs constructed in the form of function block diagrams is carried out in two steps:

(1) Before being released, first all functions and function blocks contained in a library are verified employing appropriate, usually formal methods. Such a rather expensive safety licensing needs to be carried through only once for a certain application area after a suitable set of function blocks has been identified. The licensing costs justified by the safety requirements can, therefore, be spread over many implementations, leading to relatively low costs for each single automation project. In general, rather few library elements are sufficient to formulate all programs in a particular area of automation, e.g. 67 for chemical process engineering (VDI/VDE, 1995). As the details of the function blocks’ implementation on the slave processors are part of the architecture, they remain invisible from the application programming point of view.

(2) Then, for any given application program, only the correct implementation of the corresponding interconnection pattern of invoked functions and function block instances (i.e. a certain dataflow) needs to be verified.

Application software is safety licensed by subjecting the object code loaded into the master processors to diverse back translation, a verification method developed in the course of the Halden experimental nuclear power plant project (Krebs & Haspel, 1984). 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 to the inversely obtained re-specifications. Of course, in general this method is extremely cumbersome, time consuming, and expensive. This is due to the semantic gap between a specification formulated in terms of user functions on one hand and the usual machine instructions carrying them out on the other. Applying the programming paradigm of function blocks, however, a specification is directly mapped onto sequences of procedure invocations and parameter passing. It takes only minimum effort to verify a master program by interpreting such code, which just implements a particular module interconnection pattern, and by re-drawing the corresponding graphical program specification. Diverse back translation is especially well suited for the verification of the correct implementation of graphically specified programs on the architecture introduced above for the following reasons:

- The method is essentially informal, easily comprehensible, and immediately applicable without any training. Thus, it is extremely well suited to be used on the application programming level by people with most heterogeneous educational backgrounds. Its ease of understanding and use inherently fosters error free application.
- The effects of high complexity utility and computer-like programs, whose correctness cannot be established, are verified, too.
- 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 the architecture of the execution platform between the levels interfacing to humans and to the machine, diverse back translation leads back in one easy step from machine code to problem specification.
- For this architecture, the effort required to utilise diverse back translation for the safety licensing of application programs is by several orders of magnitude smaller than for the von Neumann architecture, once a certain set of function blocks has been proven correct.

The application of back translation is now to be illustrated by working out a relatively simple, but realistic example. The program representation levels function block diagram and object code for the master processor 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.

Fig. 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 (VDI/VDE, 1995). An analogue measuring value, the controlled variable,
is acquired by a function block of type \( \text{IN}_A \) from the input channel with address \( \text{INADR} \), and scaled within the range from \( \text{XMIN} \) to \( \text{XMAX} \) to a physical quantity with unit \( \text{XUNIT} \). The controlled variable is fed into a function block of type \( C \) performing proportional–integral–differential (PID) regulation subject to the control parameters \( \text{KP}, \text{TN}, \text{TV} \). The resulting regulating variable is converted to an analogue value by a type \( \text{OUT}_A \) output function block, and switched onto the channel addressed by \( \text{OUTADR} \). In addition, the controlled variable is also supervised, with the help of two instances of the \( \text{SAM} \) limit switch standard function block type, to be within the limits given by the parameters \( \text{LS} \) and \( \text{HS} \). 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 \( \text{OR} \) function block as well. This, in turn, causes the type \( \text{AM} \) 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.

The code of this example program for the master processor is listed in Table 1. It shows a (readable) assembly language version in which, for denotational simplification, MOV\E instructions to the (memory mapped) FIFO-input and -output registers are denoted by GET and PUT, respectively. Of the different function block types instantiated in the example, \( \text{C}, \text{SAM}, \text{AM} \) have internal state variables, viz., \( \text{C} \) has 3 and the other two types have 1 each. This object code 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. \( \text{ID}-\text{C} \)) of the corresponding block out of an appropriate location or location to the slave’s input FIFO. 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. \( \text{RAM-loc-}\text{B2-}\text{isv} \)) locations for each instance of a function block with internal states. When the slave processor has received all these data, 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. \( \text{TMP}-\text{X} \)), and the latter loading it from there. In other words, each connection in a function block diagram gives rise to exactly one transfer from the slave’s output FIFO to a RAM cell, and to one or more transfers from there to the slave’s input FIFO. 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 masters’ object programs, the process of back translation—disassemble and decompile object code—turns out to be very easy. To perform back translation, first the STEP instructions are searched, which clearly separate the different (sequential) steps contained in a program from each other. The code between two STEP instructions corresponds to one function block diagram. Then, the first GET instruction is interpreted. It identifies a function block instance 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, 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, 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 locations in the master processors’ RAM are correctly initialised, 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, as well as drawing of

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Object code representation of the example program</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET ROM-loc-ID-IN_A</td>
<td>GET ROM-loc-0</td>
</tr>
<tr>
<td>GET RAM-loc-XMIN</td>
<td>GET RAM-loc-HS</td>
</tr>
<tr>
<td>GET RAM-loc-XMAX</td>
<td>GET RAM-loc-B4-isv</td>
</tr>
<tr>
<td>GET ROM-loc-0</td>
<td>PUT RAM-loc-TMP-H</td>
</tr>
<tr>
<td>GET RAM-loc-0</td>
<td>PUT RAM-loc-B4-isv</td>
</tr>
<tr>
<td>PUT RAM-loc-TMP-X</td>
<td>GET ROM-loc-ID-SAM</td>
</tr>
<tr>
<td>GET ROM-loc-ID-C</td>
<td>GET RAM-loc-TMP-X</td>
</tr>
<tr>
<td>GET RAM-loc-TMP-X</td>
<td>GET ROM-loc-LS</td>
</tr>
<tr>
<td>GET ROM-loc-2.0</td>
<td>GET RAM-loc-B5-isv</td>
</tr>
<tr>
<td>GET ROM-loc-0.0</td>
<td>PUT RAM-loc-TMP-L</td>
</tr>
<tr>
<td>GET RAM-loc-B2-isv1</td>
<td>PUT RAM-loc-B5-isv</td>
</tr>
<tr>
<td>GET RAM-loc-B2-isv2</td>
<td>GET ROM-loc-ID-OR</td>
</tr>
<tr>
<td>GET RAM-loc-B2-isv3</td>
<td>GET RAM-loc-TMP-H</td>
</tr>
<tr>
<td>PUT RAM-loc-B2-isv1</td>
<td>GET RAM-loc-TMP-L</td>
</tr>
<tr>
<td>PUT RAM-loc-B2-isv2</td>
<td>PUT RAM-loc-TMP-OR</td>
</tr>
<tr>
<td>PUT RAM-loc-B2-isv3</td>
<td>GET ROM-loc-ID-AM</td>
</tr>
<tr>
<td>GET ROM-loc-ID-OUT_A</td>
<td>GET RAM-loc-TMP-OR</td>
</tr>
<tr>
<td>GET RAM-loc-TMP-V</td>
<td>GET ROM-loc-1</td>
</tr>
<tr>
<td>GET RAM-loc-OUTADR</td>
<td>GET ROM-loc-A1</td>
</tr>
<tr>
<td>GET ROM-loc-ID-SAM</td>
<td>GET ROM-loc-14</td>
</tr>
<tr>
<td>GET RAM-loc-TMP-X</td>
<td>PUT RAM-loc-B7-isv</td>
</tr>
</tbody>
</table>
the block’s symbol and of the corresponding connections is repeated until a STEP instruction is reached, which terminates the step and, thus, the corresponding function block diagram.

References


