, , 1–17 ()

A Highly Dependable Computing Architecture for Safety-Critical Control Applications

BERND J. KRÄMER bernd.kraemer@fernuni-hagen.de
NORBERT VÖLKER norbert.voelker@fernuni-hagen.de
FernUniversität, Faculty of Electrical Engineering, D-58084 Hagen, Germany

Received January 15, 1996; Revised June 17, 1996

Editor:

Abstract. More and more technical systems are supervised, controlled and regulated by programmable electronic systems. The dependability of the entire system depends heavily on the safety of the embedded software. But the technological trend to entrust software with tasks of growing complexity and safety relevance conflicts with the lacking acceptance of rigorous proofs of software safety.

Based on an international standard for higher level programming languages for programmable logic controllers (PLC, IEC 1131-3), a mathematically based method for validating the behavioral correctness and the functional safety of graphical designs of safe-critical control applications is introduced. The design elements taken from a domain specific module library are proven correct and safe only once. The functional correctness and satifaction of safety requirements of new application graphical programs can then be shown effectively by reference to the proven properties of the library components used. This approach is part of an comprehensive computing architecture for safety critical control programs which is presented in a survey.

Keywords: Safety-Critical Real-Time Systems, PLC Programming, Dependable Software, Modular Verification, Higher Order Logic Theorem Proving

1. Introduction

Programmable Electronic Systems are spreading to more and more fields of everyday life. They can be found in household appliances, motor vehicles of higher price ranges, emergency shut-down systems of nuclear power plants, medical treatment systems, traffic control systems, or process automation.

Severe accidents in the chemical and nuclear power industry, spectacular plane crashes and intrusions into company and government computer networks have increased the suspicion generally nursed in society that insufficient quality of the software embedded in these technical applications is the main cause of system unreliability. Thus, the technological trend to more flexible and complex systems and devices is opposed by the growing safety awareness of modern society and a critical attitude towards software taking over control in technical solutions. In addition, in Germany and other countries, technical devices and equipment that can endanger human life or the environment have to be licensed formally by regulatory authorities before being taken into use and later during their life-time.

These authorities are still quite reluctant licensing exclusively program controlled safety related automation systems. In general, no safety-critical systems containing software of non-trivial complexity are licensed, as yet, because: rigorous proof techniques and robust tools that can be used effectively by practitioners in regulatory authorities and in the application domain are not available; although existing design guidelines and testing procedures may help to prevent or detect design and programming errors, they cannot guarantee the absence of faults that may cause a disaster; the method of diverse back-translation currently used by the TÜV Rheinland and other regulatory bodies is very time consuming and does not scale up. In this approach, the inspector tries to reconstruct a specification from the assembly or machine code manually and check it for conformity with the developer's specification. The method is effective in the sense that it is even able to detect compiler errors but experience has shown that it does not scale up. Up to two man-months are needed, for example, to verify four kByte of machine code.

1.1. Hardware versus Software Safety

Safety techniques of traditional engineering disciplines deal mainly with random failures of hardware. Design faults are not taken into consideration seriously because it assumed that they can be avoided by systematic design and validation techniques and because the extensive use of hardware components has demonstrated their dependability.

For hardware, this assumption is justified due to the relatively low system complexity and because a particular hardware design is typically produced in large quantities. These assumptions cannot be transferred to software intensive systems since software does not wear out, and programming faults are always systematic. The largest source of errors are design and specification faults, particularly non-considered situations, ambiguous or contradictory requirements.

In spite of some remarkable progresses in the fields of standard software and libraries of re-usable software modules, the majority of today's program systems still represents individual solutions that have no extensive usage history. Moreover, software contains an extremely large, often even infinite number of discrete system states which in addition — in contrast to hardware — rarely are regular. This fact heavily reduces the possibilities to run exhaustive program tests or to create realistic test conditions.

1.2. Formal Methods

Experiences in other engineering disciplines suggest that mathematically based methods provide an adequate approach for precise specification, design, and quality assessment of dependable program systems. It is even expected that the use of formal methods for the construction and validation of certain safety critical application classes will be required explicitly by legislature, as it already the case in the

security domain [15]. Further evidence was collected recently during an international seminar entitled "Functional Safety of Program Controlled Electronic Systems" [4].

Unfortunately it must be realized that formal methods are hardly used in software practice. Few exceptions include the formal specification and verification of the emergency shut-down system of the Darlington reactor in Canada [14]; the use of formal methods systems in the development of IBM's CICS system in Great Britain [16]; or the extensive use of verification systems for military applications with strong data security requirements in the USA and Canada.

Reasons why many practitioners abstain from formal methods are certainly unrealistic expectations about the contribution of formal methods because people are often confusing correctness with adequacy. Establishing correctness means to compare two formal objects, e.g., a specification and a corresponding program, and answer the question: "Are we doing the thing right?". This is, what formal methods can help to achieve. Determining the adequacy of a specification, design, or a program, however, means to answer the question: "Are we doing the right thing?" by comparing the formal object's behavior with our mental expectations. Here, formal methods are of lesser help because they cannot guarantee that our understanding and model of the real world is appropriate. The accident of the Lufthansa plane caused by a design error in Warsaw for some years ago is an example for the fact that correct implementations of inappropriate software specifications[1] cannot be discovered by formal verification. Formal proof techniques can only verify mathematical objects like requirement, design or program specification concerning properties such as consistency, soundness and completeness. Another reason for disappointment results in the fact that many formal methods and their associated tools reach their limits as soon as they are applied to problems of a complexity usual in practice. Also the requirements for a formal basic training of the users of formal methods, which often are too high, form an acceptance threshold still being too high today.

1.3. Overview

This contribution describes methodological aspects of a high integrity systems project. We focus on the development and analysis of software for programmable logic controllers (PLC) with high safety requirements. The language concepts for PLC programming were defined in the international standard IEC 1131-3 [1]. They are especially suited for industrial automation projects which we chose as a focal application domain. The limited complexity of algorithms and dats structures used and the rich body of domain knowledge enables us to tailor suitable and mature formal methods to the work procedures of practitioners in the field.

In the following section, we outline the IEC languages relevant for our approach. Section 3 presents an overview of a safety-oriented computing architecture which is partly realized. In Section 4, we introduce a simple example along which we discuss the formal specification and verification of graphically designed application

programs with recourse to already proven properties of their component modules taken from an application-specific library of standard building blocks. Finally, we discuss further work aiming at the completion of our high integrity systems development architecture.

2. Language Concepts of the IEC Standard

The standard IEC 1131-3 contains four compatible languages. The *Function Block Diagram (FBD)* and the *Sequential Function Chart (SFC)* languages have a visual representation, while *Structured Text (ST)* is a conventional higher level programming language with a textual syntax. Function block types are the central language elements of FBD. The standard includes a set of function block types realizing basic processing functions of process automation. Each function block type encapsulates a data structure which is divided into input, output and internal state variables of arbitrary types. Output variables are functionally dependent on the input and state variables. Outputs and inputs of function block instances can be "wired together" to a diagram of function blocks forming the actual application program.

Fig. 1 shows a hierarchical function block type, called `measure`, which is built up from an instance of the standard type `IN_A`, named `input`, and two instances of the standard type `SAM`, named `highlimit` and `lowlimit`. `IN_A` converts an analogue value read from address `HWADR` to a numerical value. This value is output at port `X` after having been scaled to the boundary values available at the ports `XMIN` and `XMAX`. The unit of the measured value is determined by the character string provided at input t `XUNIT`. Function block type `SAM` serves as a boundary value switch for the value at the `X` port. It also enables the storage of alarms or messages. Signal `X` is compared with the boundary value which arrives at port `S`. The Boolean value `LOW` determines whether `S` is interpreted as an upper or a lower bound. In the former case, the Boolean output `QS` switches from 0 to 1 in case `S` is exceeded and returns to 0 in case the value at port `X` drops under `S-SHYS`. Similarly, when `LOW` is 1, the Boolean output `QS` switches from 0 to 1 once the value at port `X` drops below the value at port `S` and returns to 0 when it exceeds `S+SHYS` again. `SHYS` describes the switching difference or hysteresis. Port `SHYS` may remain unconnected and is then assumed to be 0.

In fig. 2 the interface definition or signature of `SAM` is described in ST. ST is a block structured language which mainly follows Pascal concerning notation and language concepts. In addition, it offers a task concept to handle parallel real-time processes. Apart from being defined by graphical composition using the FBD language, the behavior of a hierarchical function block type can also be determined by an ST program. The program text then occurs as a function block *body* between the interface declaration and the keyword `END_FUNCTION_BLOCK`.

The second graphical process control language of the standard, SFC, can be regarded as an industrial application of Petri nets. The language concepts in SFC include transitions, steps and actions. They serve to co-ordinate the processing of function block instances which are regarded as asynchronous sequential processes.

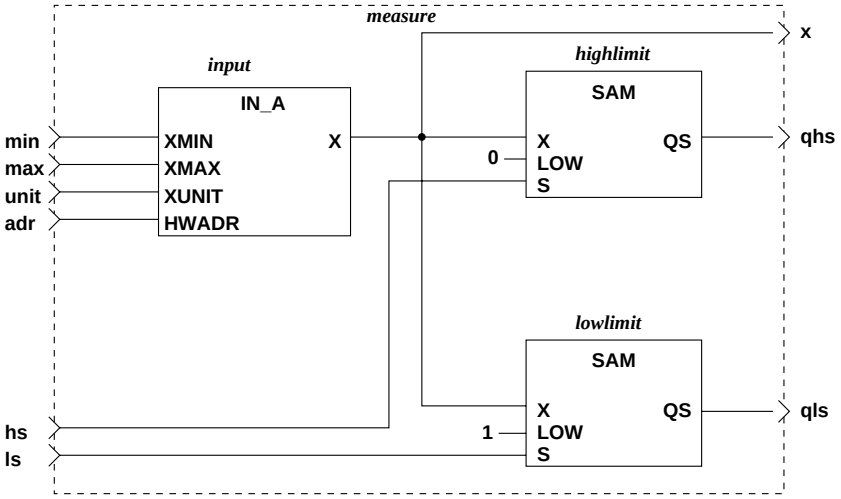

Figure 1. Derived function block type `measure`

```
FUNCTION_BLOCK SAM

 VAR_INPUT          (* input declarations *)
    X: NUM;         (* input value *)
    LOW: BOOL;      (* type of boundary value *)
    S: NUM;         (* boundary value *)
    SHYS: NUM :=0   (* hysteresis of the boundary value S *)
 END_VAR;

 VAR_OUT
    QS: BOOL        (* boundary value infringed *)
 END_VAR;

body

END_FUNCTION_BLOCK
```

Figure 2. Signature of function block type `SAM`

Fig. 6b shows an example with four steps `s0` to `s3`. The transitions separating these steps are enabled by Boolean conditions such as `lmax` and `tmax`, by preceding steps, and by the actions *fill*, *heat* and *discharge* associated with steps `s1`, `s2`, and `s3`, respectively. The double framed box states that `s0` is the initial state.

3. A Safety Oriented Computer Architecture

Different possibilities for the formal specification and verification of standard and application specific function modules serving as library elements have been demonstrated in previous work. A first prototype design and validation environment tailored to the graphical notation of higher PLC languages was introduced in [8]. Similar to the approach in [6], algebraic data structuring concepts were used for requirement specification. The properties of elementary function modules are verified with the help of structural induction techniques supported by term rewrite systems. For the analysis of the dynamic behavior of entire function block diagrams, higher Petri nets marked with instances of abstract data types were suggested. In [10] the components of an emergency shut-down system used in reactor control were specified in higher order predicate logic and verified mechanically using the Isabelle/HOL theorem proof assistant.

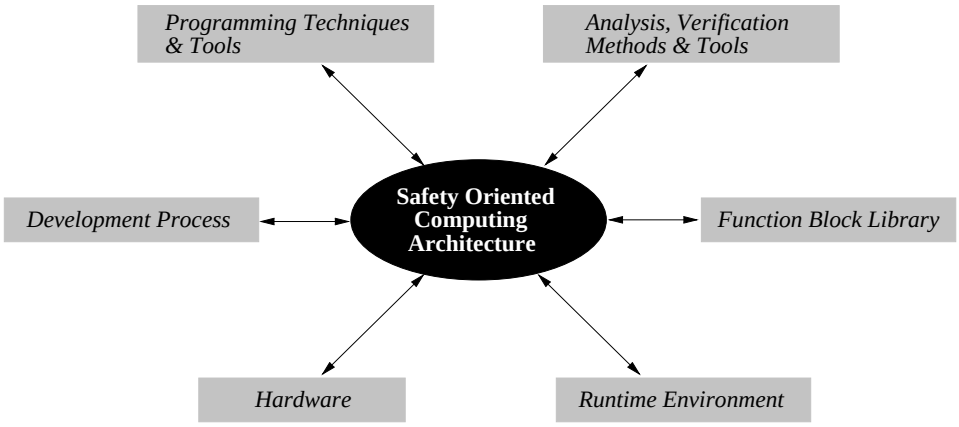

Figure 3. Structure of a safety oriented computing architecture

The suggested specification, design and validation method is supported by the computing architecture presented in fig. 3. The architecture includes ergonomically designed tools, a library of re-usable software modules as well as a simple specialized run-time environment. The latter is tailored for the language concepts of the standard and runs on a fault tolerant hardware (cf. [6]). Two prototype versions of the hardware architecture are operational. They are supported by a simple runtime environment. It was designed to serve the needs of the IEC standard and reduce the potential for introducing errors.

A customized computer supported process model organizes the entire development and licensing process. It leads application developers in companies and engineers affiliated with regulatory authorities through the prescribed process steps. In addition, it executes clerical development steps automatically and records the entire construction and licensing process. The essential components of the develop-

ment process have been described in [9]. An executable refinement of this process model can be designed for concrete application domains and work procedures with reasonable expenditure. The basic idea is to formulate the individual steps of an application specific development or licensing process as production rules [11].

Relying on the standard IEC 1131-3, a joint commission of the German Associations of Engineers (VDI) and Electrical Engineers (VDE) jointly produced the design guideline VDI/VDE 3696 [2]. This guideline provides a basis for the planning and configuration of process control systems. It supplies a vendor independent description language and a collection of standard function module definitions. As a preliminary work for the construction of an application specific module library, the module specifications named in the guideline design were investigated in [17]. Throughout this study

- incomplete information about the algorithmic definition of certain function modules such as time components, standard PID regulators, or characteristic components, were discovered and lacking information was added;
- ambiguities, contradictions, redundancies and design flaws in individual module descriptions were identified and recommendations for their removal were developed;
- a large number of module definitions was formally verified.

The approach to construct application programs by re-use of library components reduces development time and effort. In combination with the use of formal methods, it also provides the basis for effective formal verification and validation of critical properties of new applications because we can rely on proven properties of the modules involved. It is a desired side effect of this approach that the costs for the verification of library modules can thus be shared among many applications.

4. Verification of Hierarchical Function Modules

Our main approach to verification is based on higher order logic (HOL) theorem proving [13]. This means that both specifications and implementations are modeled in a very powerful, general purpose logic. The correctness of an implementation then becomes a mathematical theorem which can be proven using standard mathematical reasoning. Since all proofs are checked by the Isabelle/HOL system, we can be very confident that no invalid deductions occur during the proof process.

Our computation model is based on the cyclic execution of reactive systems [12]. Controllers are described essentially as net-lists of function blocks. Function blocks are entities which have their own private memory that persists from one invocation to the next. In accordance with guideline VDI/VDE 3696, the only form of communication between function blocks is the passing of parameters from output to input ports. Therefore the evaluation order of function blocks within one cycle does not matter as long as causal dependencies between function blocks are respected. Such

dependencies exist if a function block A provides input for another function block B, Therefore the execution of A must be completed before that of B is started. Currently, we are restricting ourselves to single tasks. Hence we do not need to address issues such as scheduling or processing of interrupts.

The underlying processing model contains two principal components (see Fig. 4)

- a plant, also called "unit under control", and
- the controller.

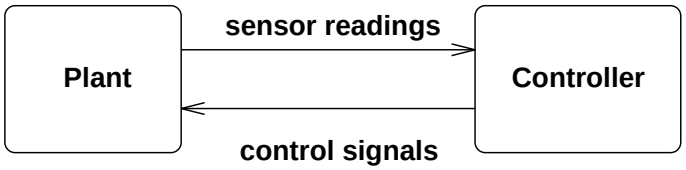

Figure 4. Control process model

In every cycle, the current reading of a number of plant sensors are digitized and communicated to the controller. After a certain delay, the controller responds by sending control signals back to the plant. The state of the plant is modeled via a number of real-valued variables. Its behavior will usually be given as a set of linear differential equations. Our approach does not require that the behavior of the plant is deterministic. Instead, it is sufficient to have a number of inequalities which limit the maximal possible variation of the plant variables per cycle.

Upper bounds for the delay in the response of the controller can be calculated by summing up upper bounds for the individual delays caused by the execution of each activated function block. In general, these delays depend on the current inputs and states of the function blocks. However, in practice, it is usually both possible and sufficient to give constant upper bounds for the delays of all components and thus also for the whole controller.

Qualitative time behavior is expressed by Linear time Temporal Logic (LTL) formulas (cf. e.g. [12]). Timers are treated as nondeterministic functions subject to the condition that the values of subsequent readings increase monotonously as long as there is no reset. Of course, we also assume that the readings of timers are compatible with the bounds on the function block delays. We believe that this simple model of time is adequate for most control applications.

The reactive model sketched above allows modular verification. Thus, base components can be verified first. In subsequent steps, these results can be used in the proof of the correctness of the entire control system. In this inductive process requirements specifications of base components are combined by logical conjunction, while a composite program is formed by identifying input and output variables of connected components according to their wiring in the given diagram.

For the mechanical support of the proof method, we are working on an extension of a method for the verification of elementary function blocks which already has been implemented with the help of the proof system Isabelle/HOL [10].

4.1. Example

As a simple example, we consider the process control for a container automation taken from VDI/VDE guideline 3696. The physical configuration is depicted in fig. 5. In the VDI/VDE guideline the automation task for this example is described

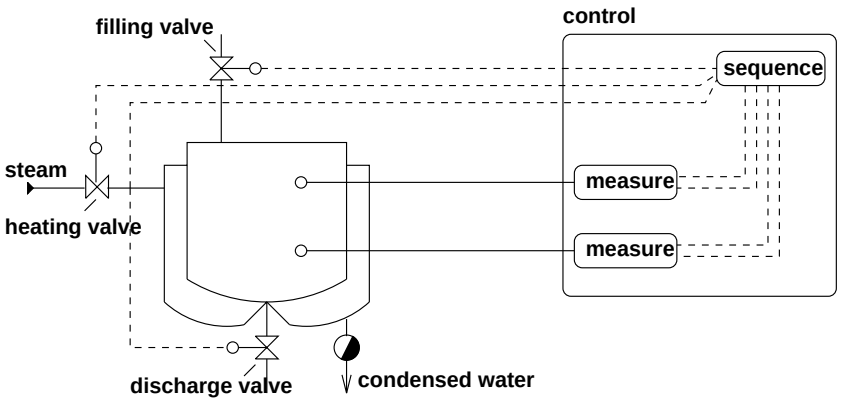

Figure 5. Automatic Level and Temperature Control

informally as follows [2]:

> If a start button is pressed and held, and if the temperature is below 40^o C, the filling valve opens and remains open until a liquid level of 90 % is reached. Subsequently, the container casing is heated with steam by opening the heating valve until 90^o C are reached. Now, the emptying valve is opened until the filling height has fallen below 5 %; then the control goes back to its initial step and the whole process is repeated.

The controller for this system depicted in fig. 6a is composed of two instances of function block `measure`, which was shown in fig. 1a, and an instance of function block `sequence`, whose structure is illustrated in fig. 6b.

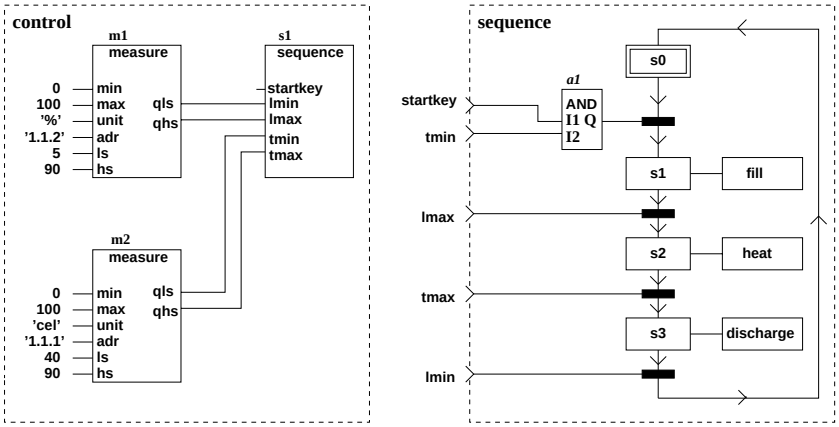

Figure 6. Function block `sequence`

4.2. Specification of Function Blocks

The core of the control system shown in fig. 6a is function block `sequence`. It has five Boolean inputs *startkey*, *tmin*, *tmax*, *lmin*, *lmax* and controls the three actions *fill*, *heat*, and *discharge*. The control algorithm is specified through an SFC with four steps `s0`, `s1`, `s2`, and `s3`. These steps are sequentially dependent on each other so that exactly one of them can be activated at any time.

As mentioned before, our approach to formal verification is based on higher order logic theorem proving. For modeling the behavior of function blocks, we have adopted the relational approach advocated in [13]. A function block with n input and m output ports is represented by an $(n+m)$-ary relation on streams. Every stream parameter refers to the flow of values communicated through the corresponding port. For example, the HOL type of the standard function block `SAM` is

$$\texttt{SAM} :: \texttt{NUM STREAM} \times \texttt{SIGNAL} \times \texttt{NUM STREAM} \times \texttt{NUM STREAM} \times \texttt{SIGNAL} \rightarrow \texttt{BOOL}$$

where `SIGNAL` is an abbreviation for `BOOL STREAM`. As this typing suggests, the identification of function block ports in the HOL representation is by order of parameters and not by name. Streams of some type a are simply modeled as functions from type NAT to a. As usual in HOL, function application is denoted simply by juxtaposition, i.e. $QS\,n$ is the value of the stream QS at time step n. The relation between the values of input and output parameters of `SAM` is characterized by:

$$
\begin{array}{l}
\texttt{SAM}\ X\ LOW\ S\ SHYS\ QS\ ==\\
\forall n.\ QS\ n\ =\ (\ \ LOW\ n \wedge X\ n < S\ n\\
\qquad\qquad \vee\ \neg\,(LOW\ n) \wedge S\ n < X\ n\\
\qquad\qquad \vee\ 0 < n \wedge QS\ (\texttt{pred}\ n)\\
\qquad\qquad\quad \wedge\ (LOW\ n \wedge X\ n \leq S\ n + SHYS\ n\\
\qquad\qquad\qquad \vee \neg\, LOW\ n \wedge S\ n - SHYS\ n \leq X\ n))
\end{array}
$$

The readability of this formula suffers from the frequent occurrence of the variable n which refers to the current point of time. Using the linear time temporal logic operator `ALWAYS` and stream versions of the logical and arithmetical operations obtained by pointwise lifting (cf., e.g., [12]), the following equivalent specification of `SAM` is possible:

$$
\begin{array}{l}
\texttt{SAM}\ X\ LOW\ S\ SHYS\ QS\ =\\
\texttt{ALWAYS}\ \ (QS\ \texttt{Eq}\ (\ \ LOW\ \texttt{AND}\ X\ \texttt{Less}\ S\\
\qquad\qquad\qquad \texttt{OR}\ \texttt{Not}\ LOW\ \texttt{AND}\ S\ \texttt{Less}\ X\\
\qquad\qquad\qquad \texttt{OR}\ \ (\lambda\, n.0 < n)\ \texttt{AND Pre}\ QS\\
\qquad\qquad\qquad\quad \texttt{AND}\ \ (LOW\ \texttt{AND}\ X\ \texttt{Le}\ S + SHYS\\
\qquad\qquad\qquad\qquad \texttt{OR Not}\ LOW\ \texttt{AND}\ S - SHYS\ \texttt{Le}\ X)))
\end{array}
$$

As usual, the expression $\lambda\, n.E$ denotes a λ-binding, i.e. a function with formal parameter n and result E.

Using an embedding of the SFC formalism in HOL, a third equivalent formulation of `SAM` via sequential function charts would also be possible. Instead, we will give below some temporal logic formulas which can be derived from the SFC specification of the function block type `sequence` in fig. 6b. These formulas all assume the premise

$$\texttt{sequence}\ \mathit{startkey\ lmin\ lmax\ tmin\ tmax\ fill\ heat\ discharge}$$

where

$$\texttt{sequence}\ ::\ \texttt{SIGNAL} \times \ldots \times \texttt{SIGNAL}\ \rightarrow\ \texttt{BOOL}$$

is the relation on input- and output signals derived from the SFC specification. Further, the stream

$$\texttt{Step}\ ::\ (\texttt{STEP SET})\,\texttt{STREAM}$$

consists of the corresponding set of activated steps of `sequence` over time.

First, the possible sets of active steps of `sequence` are characterized by the formula:

$$\texttt{ALWAYS}\ (\texttt{Step} =_c \{s_0\}\ \texttt{OR Step} =_c \{s_1\}\ \texttt{OR Step} =_c \{s_2\}\ \texttt{OR Step} =_c \{s_3\})$$

where the operator

$$=_c\ ::\ a\ \texttt{STREAM} \times a\ \rightarrow\ \texttt{SIGNAL}$$

compares the values of a stream with a constant:

$$\forall\, n.\ (x =_c a)\, n \ = \ (x\, n = a)$$

Because the steps $s_0, \ldots, s_3$ are pairwise different, the formula above implies that always exactly one of the four steps is active.

In the following, we will drop the outer `ALWAYS` quantifier in LTL formulas and assume an implicit universal quantification over all instances. The LTL formulation of the step transition relation of the SFC specification of function block `sequence` can be derived directly by symbolic evaluation of the specification:

$$\begin{array}{lll}
\texttt{Step} =_c \{s_0\} & \Longrightarrow & \texttt{NEXT}\,(\texttt{UNLESS}\ (tmin\ \texttt{AND}\ startkey)\ (\texttt{Step} =_c \{s_0\})) \\
\texttt{Step} =_c \{s_1\} & \Longrightarrow & \texttt{NEXT}\,(\texttt{UNLESS}\ lmax\ (\texttt{Step} =_c \{s_1\})) \\
\texttt{Step} =_c \{s_2\} & \Longrightarrow & \texttt{NEXT}\,(\texttt{UNLESS}\ tmax\ (\texttt{Step} =_c \{s_2\})) \\
\texttt{Step} =_c \{s_3\} & \Longrightarrow & \texttt{NEXT}\,(\texttt{UNLESS}\ lmin\ (\texttt{Step} =_c \{s_3\})) \\
\texttt{Step} =_c \{s_0\}\ \texttt{AND}\ \texttt{NEXT}\,(tmin\ \texttt{AND}\ startkey) & \Longrightarrow & \texttt{NEXT}\,(\texttt{Step} =_c \{s_1\}) \\
\texttt{Step} =_c \{s_1\}\ \texttt{AND}\ \texttt{NEXT}\ lmax & \Longrightarrow & \texttt{NEXT}\,(\texttt{Step} =_c \{s_2\}) \\
\texttt{Step} =_c \{s_2\}\ \texttt{AND}\ \texttt{NEXT}\ tmax & \Longrightarrow & \texttt{NEXT}\,(\texttt{Step} =_c \{s_3\}) \\
\texttt{Step} =_c \{s_3\}\ \texttt{AND}\ \texttt{NEXT}\ lmin & \Longrightarrow & \texttt{NEXT}\,(\texttt{Step} =_c \{s_0\})
\end{array}$$

The temporal operator `UNLESS` P Q is valid at some point in time, provided Q remains `TRUE` as long as P is `FALSE`. It does not say anything about the truth of P. The `NEXT` operator shifts a stream by one instance "to the left":

$$\forall\, n.\ (\texttt{NEXT}\ x)\, n \ = x(n+1)$$

The initial state and the correspondence between steps and actions complete our list of propositions about the function block `sequence`:

$$\begin{array}{lcl}
\texttt{Step}\ 0 & = & \{s_0\} \\
fill & = & (Step =_c \{s_1\}) \\
heat & = & (Step =_c \{s_2\}) \\
discharge & = & (Step =_c \{s_3\})
\end{array}$$

Composition and instantiation of function blocks lead to analogous operations on relations. Ports which connect components inside a function block can be made invisible to the outside by existential quantification. Assuming a relation `measure` modeling the function block of the same name, the net-list description of `control` in fig. 6a can be specified as follows:

$$\begin{array}{l}
\texttt{control}\ startkey\ fill\ heat\ discharge\ == \\
\exists\ m1.qls\ m1.qhs\ m2.qls\ m2.qhs\ s1.lmin\ s1.lmax\ s1.tmin\ s1.tmax\ . \\
\texttt{measure}\ 0\ \ 100\ \ '\%'\ \ '1.1.2'\ \ 5\ \ \ 90\ \ m1.qls\ \ m1.qhs\ \wedge \\
\texttt{measure}\ 0\ \ 100\ \ 'cel'\ \ '1.1.1'\ \ 40\ \ 90\ \ m2.qls\ \ m2.qhs\ \wedge \\
\texttt{sequence}\ startkey\ s1.lmin\ s1.lmax\ s1.tmin\ s1.tmax\ fill\ heat\ discharge \\
\wedge\ m1.qls = s1.lmin\ \wedge\ m1.qhs = s1.lmax \\
\wedge\ m2.qls = s1.tmin\ \wedge\ m2.qhs = s1.tmax
\end{array}$$

Following our process model of fig. 4, the function block `control` receives analogue input streams from the plant at the addresses '1.1.1' and '1.1.2'. The controller responds by sending the Boolean signals *fill*, *heat*, *discharge* to the plant.

4.3. Verification of a Safety Property

As we have already seen in the previous section, some safety properties can be derived from the individual specifications of the involved function alone. For example, the specification of function block type `sequence` implied immediately that no more than one of the three actions *fill*, *heat* and *discharge* can be activated at any time.

As an example for a safety property of the composed system, we will give an informal sketch of the derivation of an upper bound for the liquid level in the tank. For this, we assume that the current value of the liquid level is read with a frequency of $f = 100Hz$. The maximal flow of liquid into the tank is `dfmax` $= 2\%$ per second. Further, we assume that the filling valve reacts with a mechanical delay of `dvalve` $= 0.5\,s$, i.e. it takes this time for the filling valve to open and close completely after the corresponding change of the controller signal *fill*. During the opening or closing of the valve, we assume a linear increase and decrease of the liquid stream, respectively.

In order to obtain an upper bound for the delay in the controller response, we examine the tasks accomplished by the components. In addition to one analog-digital conversion, these consist mainly of a couple of simple arithmetic and Boolean operations. Since none of the programs implementing a function block requires a loop or a backward jump, upper bounds for the execution times can be obtained by simply multiplying the number of instructions with the maximal execution time per instruction. We will require in the following that the controller delay is less than `dctrl` $= 1ms$. Considering the small number of instructions necessary for implementing the function blocks, this delay could even be achieved easily with a slow, low-cost PLC and a program written in a strictly sequential, higher level programming language.

For the following verification, we assume a composition of `control` as in fig. 6a. This is expressed by the predicate

$$\texttt{control}\ \ \mathit{startkey}\ \ \mathit{fill}\ \ \mathit{heat}\ \ \mathit{discharge}$$

where the relation `control` is defined as above. Further we assume that

$$m1.\mathit{qls}\ \ m1.\mathit{qhs}\ \ m1.X\ \ s1.\mathit{lmin}\ \ s1.\mathit{lmax}$$

denote the respective streams at the input/output ports of function block $m1$ and $s1$. Thus, ignoring rounding errors in the digitization step, the last reading of the fluid level in the container is given by the value of the stream $m1.X$.

Recall that our aim is to obtain an upper bound on the liquid level in the tank. Assuming that the tank is initially empty, i.e.

$$(m1.X)\,0 < m1.ls$$

we deduce from the specification of **SAM**

$$(m1.qls)\,0 = \texttt{TRUE}$$

The equality

$$m1.qls = s1.lmin$$

reflects the composition of these two ports in **control**. Using it, we obtain

$$(s1.lmin)\,0 = \texttt{TRUE}$$

Next note that the water level can not rise while the controller is in step $s0$:

$$(x = m1.X \ \texttt{AND}\ \texttt{Step} = \{\texttt{s0}\}) \implies \texttt{NEXT}\,(m1.X \leq x)$$

This implies

$$\texttt{Step} = \{\texttt{s0}\} \implies s1.lmin$$

by induction over the number of instances and using the precondition of the only transition which leads to an activation of step $s0$.

The increase in the liquid level is always bound by **dfmax**

$$\begin{array}{rcl}(x = m1.X) & \implies & \texttt{NEXT}\,(m1.X \leq x + \texttt{dfmax}/f) \\ & = & \texttt{NEXT}\,(m1.X \leq x + 0.02\%)\end{array}$$

The switching from step **s1** to **s2** takes place as soon as $m1.X$ exceeds $m1.hs$. Because of the last two inequalities, simple arithmetic shows that the value of $m1.X$ in this instance must be less than $m1.X + \texttt{dfmax}/f$. Due to the controller delay, the filling will continue for another $1ms$, leading to a maximal intake of $\texttt{dfmax} * \texttt{dctrl} = 0.002\%$. Integration shows that during the closing phase of the filling valve, the liquid level will rise by at most

$$\int_0^{0.5} (2 - 4 * t)\%\ dt \ = \ 0.5\%$$

Since no more liquid is added to the tank in the remaining steps of the control loop, induction over the number of instances proves

$$\begin{array}{rcl}\forall t.\ \texttt{level}(t) & < & m1.hs + \texttt{dfmax}/f + \texttt{dfmax} * \texttt{dctrl} + 0.5\% \\ & = & 90.522\%\end{array}$$

5. Further Work

The results presented in the main body of this paper are supplemented by an ongoing "Study of programming languages with limited features suitable for control applications with safety tasks" supported by the Federal Institute of Labor. This study aims at

- identifying language elements and constructions of higher programming languages which may cause design faults, impede a sufficient analysis of the program code, or render the formal proof of functional correctness, safety and timing constraints difficult or even impossible;

- selecting reliable subsets of suitable programming languages staggered according to safety requirement classes;

- characterizing corresponding restrictions for syntax and type checkers, compilers, and run-time environments for the chosen languages;

- designing comprehensive methods for systematic programming and formal verification of safety-oriented program systems.

Starting-point for this study are the Safe Technical Language defined by Daimler-Benz AG [5], the comparative evaluation of programming languages for real time applications presented in [7], and our own preliminary work.

In parallel, an international project is planned in which, for the first time, a complete real-time operating system kernel shall be specified and verified formally.

Theorem prover based verification methods such as the one sketched above are very trustworthy, flexible and support modular proofs. However, in general they require a sophisticated guidance by the user. Hence it is very important to find ways to automates recurring development steps. In this respect, the integration of automatic model checking procedures such as pioneered by N. Shankar for the PVS system seems particularly promising.

Another and for the automation practice decisive step is finally the construction of a complete development environment. According to this approach, it should support a mathematically precise description of safety critical components of PLC-based process control systems and the verification of fulfilling the given safety requirements with the help of formal proof techniques. An ergonomic design of the construction and proof processes and tools matching the working processes in development laboratories and licensing authorities such as Technical Supervising Organizations and internal company quality control groups must be focused here. However, this step can only be executed sensibly in close co-operation with interested vendors, users and evaluators for PLC controllers in safety critical fields, who we are still looking for.

The authors would like to thank R. Lichtenecker and the anonymous referees for their helpful comments on an earlier version of the paper.

Notes

1. The software prevented the pilots to switch the reverse thrust on in time. The reason was that the control software failed to detect the plane's touching of the ground due to the reduced friction caused by rain and the undercarriage's one-sided touching of the ground caused by severe side winds.

References

1. IEC Draft International Standard 1131-3. *Programmable Controllers. Part 3: Programming Languages.* International Electro-technical Commission, Geneva, 1992.
2. VDI/VDE Richtlinie 3696. Herstellerneutrale Konfigurierung von Prozeßleitsystemem. Technical report, Düsseldorf, 1993 (in German).
3. R.M. Cardell-Oliver and C. Southon. A Theorem Proving Abstraction of Model Checking. Technical Report CSM-253, Department of Computer Science, University of Essex, England, 1995.
4. W.J. Cullyer, W.A. Halang, and B.J. Krämer (Eds.). High integrity programmable electronic systems. Dagstuhl-Seminar-Report 107, IBFI GmbH, Schloß Dagstuhl, D-66687 Wadern, Germany, 1995.
5. G. Egger, A. Fett, and P. Pepper. Formal specification of a safe PLC language and its compiler. Technical report, Daimler-Benz AG, 1994.
6. W.A. Halang, S.-K. Jung, B.J. Krämer, and J. Scheepstra. *An Safety Licensable Computing Architecture.* World Scientific, 1993.
7. W.A. Halang and A.D. Stoyenko. Extending PEARL for industrial real-time applications. *IEEE Software*, 10(4):65–74, 1993.
8. W.A. Halang and B.J. Krämer. Achieving high integrity of process control software by graphical design and formal verification. *Software Engineering Journal*, 7(1):53–64, January 1992.
9. W.A. Halang and B.J. Krämer. Safety assurance in process control. *IEEE Software*, Special issue on Safety-Critical Software:61–67, January 1994.
10. W.A. Halang, B.J. Krämer, and N. Völker. Formally verified building blocks in functional logic diagrams for emergency shutdown system design. *High Integrity Systems*, 1995.
11. B.J. Krämer and B. Dinler. Software process environment drives hardware synthesis. In P.A. Ng, F.G. Sobrinho, C.V. Ramamorthy, R.T. Yeh, and L.C. Seifert, editors, *Systems Integration '94*, volume I, pages 354–361, Sao Paulo, Brazil, 1994. IEEE Computer Society Press.
12. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*, volume 1. Specification. Springer Verlag, 1992.
13. T.F. Melham. *Higher Order Logic and Hardware Verification.* Cambridge University Press, 1993.
14. D.L. Parnas, J. van Schouwen, and S.P. Kwan. Evaluation of safety-critical software. *Communications of the ACM*, 33(6):636–648, 1990.
15. H. Pohl and G. Weck (Eds.). Internationale Sicherheitskriterien, Oldenbourg Verlag, München, Wien, 1993 (in German)
16. J. Wordsworth. Practical experience of formal specification: A programming interface for communications. In C. Ghezzi and J.A. McDermid, editors, *ESEC '89 2nd European Software Engineering Conference*, number 387 in Lecture Notes in Computer Science, pages 140–158, Berlin, Heidelberg, New York, 1989. Springer Verlag.
17. G. Wulf. Überprüfung des Richtlinienentwurfs VDI/VDE 3696 und Verifikation der darin definierten Funktionsbausteine. Diplomarbeit, FernUniversität, 1995 (in German).

Received Date
Accepted Date
Final Manuscript Date