



## ARTICLE IN PRESS



ELSEVIER

Science of Computer Programming III (IIII) III–III

Science of  
Computer  
Programming

www.elsevier.com/locate/scico

1

# Automated verification of function block-based industrial control systems

3

Norbert Völker<sup>a, \*</sup>, Bernd J. Krämer<sup>b</sup>

5

<sup>a</sup>Department of Computer Science, University of Essex, Colchester CO4 3SQ, UK

7

<sup>b</sup>Faculty of Electrical Engineering and Information Engineering, FernUniversität,  
58084 Hagen, Germany

---

### Abstract

9 The international standard IEC 61131-3, which supports Brad Cox' concept of "Software-ICs"  
11 for industrial control programming, is increasingly being used in safety-related application do-  
13 mains. They include safety-instrumented functions, such as burner management, emergency shut-  
15 down and gas leak detection, but also complex automation processes controlling, e.g., chemical  
production plants. For such highly dependable applications, code inspection and testing, the pre-  
dominant quality assurance techniques used in practice today, are, in general, not sufficient to  
demonstrate the functional correctness and safety of an application.

17 This paper presents a theorem prover-based verification technique as a supplementary validation  
19 measure. The verification task is separated into the a priori verification of reusable function  
21 blocks, which are usually maintained in domain-specific libraries, and a separate compositional  
23 proof of individual application programs. Core concepts of the standardized languages, their  
semantic embedding into higher order logic, and the verification approach are illustrated with  
a small example. Some design ideas for a verification tool usable by automation engineers and  
safety licensing authorities conclude the contribution. © 2001 Published by Elsevier Science  
B.V.

25 *Keywords:* Safety-critical control systems; Dependable software; PLC programming; IEC  
61131-3; Compositional verification; Higher order logic; Theorem proving

---

### 1. Introduction

27 Programmable logic controllers (PLCs) form a growing market of special purpose  
29 hybrid systems integrating micro-electronic and software components. PLCs are par-  
ticularly suited to solve application problems in machine logic, process automation,  
manufacturing and data acquisition. They were developed to replace traditional hard-  
wired switching networks based on relay or discrete electronic logic.

---

\* Corresponding author.

E-mail addresses: norbert@essex.ac.uk (N. Völker), bernd.kraemer@fernuni-hagen.de (B.J. Krämer).

1 The rapid development of PLC systems in the 1980s led to a wealth of incompatible  
2 vendor-specific PLC programming languages within the process industries impeding  
3 the design of more complex, open and distributed control applications. In response to  
4 this situation, the international standard IEC 61131-3 for PLC programming [13] was  
5 developed by the IEC [14]. The standard applies to a wide range of programmable  
6 controllers. It harmonizes the way engineers look at industrial control by standardiz-  
7 ing the programming interface and a large collection of reusable components, called  
8 function blocks.

9 The standard provides a class of five purpose-built languages that overlap concep-  
10 tually and share a subset of programming elements. Three languages of the stan-  
11 dard, function block diagram (FBD), ladder diagram (LD) and sequential function  
12 chart (SFC) have a graphical appearance. FBD embodies Brad Cox' concept of  
13 "Software-ICs" and supports component-based application programming. The SFC no-  
14 tation, which derives from Petri nets [27], is mainly used for depicting the dynamic  
15 behavior of control systems including alternative and concurrent execution  
16 steps.

17 New capabilities of PLCs, the comfort of the PLC languages, and strong economical  
18 demands led to the current situation that we are increasingly depending on PLC-based  
19 systems for control and automation functions in safety-related applications. Examples  
20 include (air) traffic control, patient monitoring, process automation in chemical and oth-  
21 ers industries, and emergency shut down systems in power generation and in production  
22 line control.

23 The growing awareness of our society of the need to protect the environment, a  
24 higher sensitivity to accidents caused by ill-designed technology or processes, and a  
25 declining trust in marketing statements of manufacturers produce an enormous pressure  
26 to increase the dependability of safety-related applications. In practice, however, we  
27 observe a lack of rigorous proof techniques and robust tools, which can be used effec-  
28 tively by practitioners in industry and regulatory authorities. Existing design guidelines,  
29 code reviews [7] and testing practices may help to detect design and programming er-  
30 rors. But they cannot guarantee the absence of software faults, which may cause a  
31 disastrous effect [18], because exhaustive testing is limited to rare cases.

32 The main body of this paper explores function blocks and sequential function charts  
33 to develop a modular, theorem prover-based verification framework. By taking compo-  
34 nents from application-specific libraries of verified standard function blocks, the verifi-  
35 cation of new applications is reduced considerably because only the correctness of the  
36 composition has to be established for each new application.

37 In the following section, core concepts of FBD and SFC are introduced. In Section 3,  
38 the higher order logic used to verify the functional correctness and safety of individual  
39 function blocks and entire control applications is discussed. The verification process is  
40 based on a semantic embedding of the selected PLC languages into that logic. This  
41 embedding is explained in Section 4, while our verification approach and the challenges  
42 of handling complex continuous systems are sketched in Sections 5 and 6. The paper  
43 concludes with a brief summary of the verification approach and the sketch of a future

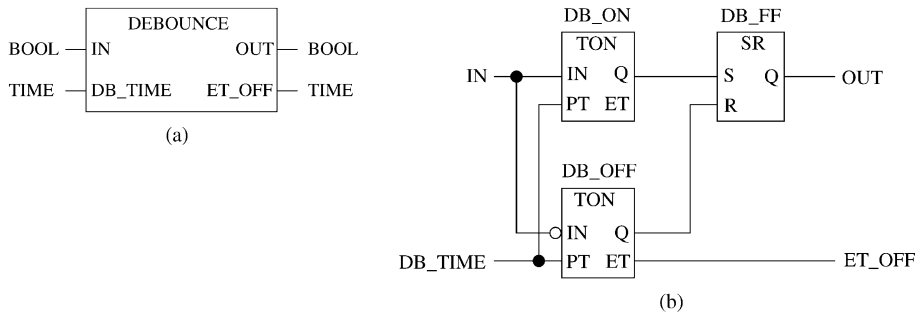


Fig. 1. Function block DEBOUNCE.

1 industrial-strength verification tool, which can ultimately be used by domain experts  
 2 with little or no expertise in software verification.

### 3 2. Function blocks, structured text and sequential function charts

4 Function blocks are program organization units with a private state that persists from  
 5 one invocation to the next. A function block interacts with its environment primarily via  
 6 input and output variables. The standard also allows global variables but our verification  
 7 framework does not support these. If needed, they can be treated as additional input  
 8 and output variables. They simply need to be properly mapped on all function blocks  
 9 of the given control loop by a suitable compilation step. Besides keeping the semantics  
 10 simple, this restriction has the advantage that the execution of function blocks has no  
 11 side effects.

12 From a semantic point of view, function blocks are a special case of deterministic  
 13 reactive modules [1]. According to the model of reactive systems [8], their execution  
 14 takes place in a sequence of rounds. At the start of each round, the input variables are  
 15 read. Then the function blocks private and output variables are updated. This update  
 16 is functionally dependent on the current value of the input variables and the previous  
 17 state of the private and output variables.

18 The description of a function block can be split into the declaration of its external  
 19 interface and a specification of the internal implementation. The former is part of the  
 20 function block signature that specifies the types and names of variables including local  
 21 instances of function blocks. In the context of graphical representations, the input and  
 22 output variables will also be referred to as ports. The interface specification is similar  
 23 to the description of interfaces in other languages such as CORBA-IDL [24]. The internal  
 24 implementation of a function block body can be carried out in any of the five IEC  
 25 61131-3 programming languages or even in other languages such as C or Java.

26 As an example, Fig. 1(a) shows a graphical representation of the external interface  
 27 of the function block DEBOUNCE taken from the IEC 61131-3. DEBOUNCE has two  
 28 input variables, IN and DB\_TIME, of type BOOL and TIME, respectively, and two  
 29 output variables, OUT and ET\_OFF, of the same types.

```

DB_ON (IN := IN, PT := DB.TIME);
DB_OFF (IN := NOT IN, PT := DB.TIME);
DB_FF (S := DB_ON.Q, R := DB_OFF.Q);
OUT := DB_FF.Q;
ET_OFF := DB_OFF.ET;

```

Fig. 2. DEBOUNCE in structured text.

1 An implementation of DEBOUNCE as a function block diagram is depicted in  
 2 Fig. 1(b). The function blocks DB.ON and DB.OFF are two separate instances of the  
 3 timer function block TON. DB.OFF is an instance of the SR flip-flop function block,  
 4 which is included in the standard. By connecting input and output ports, a diagram is  
 5 “wired together” from the components. As in the graphical representation of circuits,  
 6 the open circle at input port IN of function block DB.OFF indicates the negation of  
 7 a Boolean signal. The named instances of function blocks will usually be referred to  
 8 as function blocks also. The function block DEBOUNCE is composed from function  
 9 blocks predefined in the standard. Such a composite function block can itself be used  
 10 in further applications just as if it were one of the standard function blocks. This fea-  
 11 ture is useful for building an in-house or domain-specific collection of more abstract  
 12 function blocks.

13 The textual IEC 61131-3 language ST (structured text) is similar in appearance  
 14 to a structured programming language such as PASCAL. Fig. 2 shows an alternative  
 15 implementation of the body of DEBOUNCE in ST.

16 The second graphical language of the standard, SFC, can be regarded as an appli-  
 17 cation of Petri nets. Its language concepts include transitions, steps and actions. They  
 18 serve to co-ordinate the execution of function blocks that are regarded as asynchronous  
 19 sequential processes.

20 The role of SFC is illustrated by a small laboratory plant, which is depicted in  
 21 Fig. 3(a). The plant has been used previously as a case study for non-linear control  
 22 design methods [12] and a benchmark for the tool-aided analysis of discretely controlled  
 23 continuous systems [15]. The plant features two cylindrical tanks that are located at  
 24 different levels and are connected by a valve-controlled pipe. Two further pipes, which  
 25 are also equipped with two valves,  $V_0$  and  $V_2$ , control the flow of liquid at the inlet  
 26 and at the outlet, respectively. The liquid level in the second tank is measured by a  
 27 sensor  $L$  (see Fig. 3(a)).

28 A core safety requirement for this application is to avoid overflow in the coupled  
 29 tank system.

30 The SCF depicted in Fig. 3(b) controls the behavior of the system. It consists of five  
 31 steps  $s_0, \dots, s_4$ . The actions connected with the steps control the state of the valves: the  
 qualifiers  $S$  and  $R$  denote setting and resetting of an action, respectively. The transitions

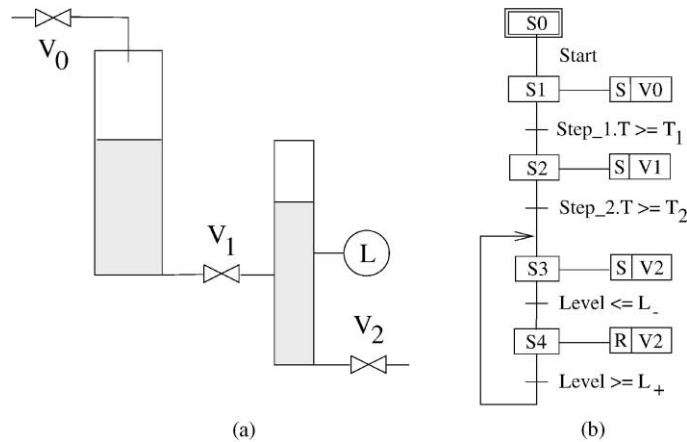


Fig. 3. Laboratory plant with SFC controller.

1 separating the steps are enabled by Boolean valued expressions representing conditions  
 2 on the state of the associated function block.

3 The encapsulation provided by function blocks together with their openness with  
 4 respect to the internal implementation furthers the reuse of function blocks in different  
 5 applications. Hence, it makes sense to develop component libraries. Examples include  
 6 the collection of standard function blocks of the IEC 61131-3, the corresponding Ger-  
 7 man standard [32], and the domain-specific library of function blocks used by a German  
 8 manufacturer of chemicals and drugs we have studied earlier. This in-house library con-  
 9 sists of about 70 function blocks that are sufficient to program many chemical process  
 10 automation tasks.

### 11 3. Higher order logic for verification

The basic logic underlying our verification approach is higher order logic [3,6].  
 12 Several reasons motivated this choice:

- 13 (1) The means of abstraction and quantification over functions make this logic very  
 14 expressive and thus well suited to the concise description of complex theories. Evi-  
 15 dence of this fact is provided by the embedding of hardware description languages  
 16 [4] and the verification of floating point algorithms [11].
- 17 (2) HOL is a widely studied and well-understood logical system with a remarkably  
 18 small number of axioms and inference rules. Its expressiveness makes it possible  
 19 to use definitional extension as the principal method of theory development. Since  
 20 this method is conservative, logical inconsistencies can be practically ruled out.
- 21 (3) Automatic type inference systems for HOL make type annotations to a great extent  
 22 unnecessary. This shortens formulas and proofs because the information contained  
 23 in the typing is automatically inferred and propagated.

1 In comparison to alternatives such as Zermelo–Fränkel set theory, there are also a few  
disadvantages:

- 3 (1) The type discipline of HOL leads to a certain loss of flexibility, cf., e.g., [17]. This  
statement remains true despite the expressiveness of polymorphism and symbol  
5 overloading available in systems such as Isabelle/HOL.
- 7 (2) In comparison with first and second order logic, the implementation of the HOL  
type system is technically more demanding. In particular, the existence of type and  
function variables complicates unification, the basic method of equation solving  
9 [22]. In addition, most research in automated theorem proving has been performed  
in the area of first order theories.

11 For our purpose, the advantages of HOL outweigh these drawbacks. Its extendibility  
makes it unnecessary to introduce a special logic for the definition of the semantics  
13 of programs and specifications. Instead, HOL provides a logical core that can serve as  
the common semantic basis for a range of different formalisms.

15 Furthermore, it is essential that several reliable and efficient mechanical theorem  
proving assistants support the logic. Our system of choice is the object logic HOL of  
17 the generic theorem proving assistant Isabelle [26]. Like the HOL system [6], Isabelle  
builds on the functional programming language SML [20]. Noteworthy alternatives  
19 include the HOL system and the LISP-based PVS [29].

21 With regard to verification, the high degree of safety and reliability of a proof  
assistant are of paramount importance. In the Isabelle system, a number of measures  
are taken to achieve this aim:

- 23 (1) Theorems are elements of a special abstract SML data type `thm`. New elements of  
this type can only be formed by a small number of operations representing valid  
25 logical deductions or explicit axioms. If one assumes that the Isabelle implemen-  
tation of these basic operations is correct, then the static type checking of SML  
27 guarantees also the logical validity of all derivations.
- 29 (2) The preferred method for extending theories is definitional. This minimizes the  
danger of logical inconsistencies.
- 31 (3) Isabelle is an open and extendible system with a freely available source code. The  
source code is well structured and written in a functional programming language  
33 with only little use of imperative features. This renders it open to be scrutinized  
by independent researchers.

Isabelle has a comprehensive international user community. These combined factors  
35 have given Isabelle—like the HOL system—the reputation of an extremely trustworthy  
proof support system.

37 In addition to safety, a high degree of proof automation is essential to cope in a  
reasonable time with the many proof obligations arising during verification. The main  
39 tools of the Isabelle system in this respect are: (a) a simplifier based on term rewriting  
and (b) a proof search tool, called the classical reasoner. External decision procedures  
41 can be invoked from Isabelle using an oracle mechanism. The degree of automation

1 is sufficient for the definition of formal semantics and the verification of small-  
2 medium-sized function block applications.

#### 3 4. Embedding function blocks in HOL

4 The main motivation behind the formulation of higher order logic as used in the HOL  
5 system was the mechanical verification of hardware. Remarkable achievements in this  
6 area include the verification of an ATM network component [5] and of RISC pipeline  
7 conflicts [31]. In comparison, success of HOL in the area of software verification has  
8 been more tedious. Research has concentrated up to now on particular aspects of real  
9 programming languages such as the type safety of a Java Subset [23].

10 The foundation of our verification framework is a HOL embedding of a subset of  
11 structured text (ST). The technical details of this embedding can be found in [33]. It is  
12 a relatively deep embedding, which means that the syntax of function blocks and the  
13 assignment of semantics are represented explicitly in HOL. Semantics are defined via  
14 evaluation functions for the four different syntactical categories, namely expressions,  
15 statements, functions and function blocks. As a result, every function block is associated  
16 with a deterministic, but not necessarily finite Mealy automaton in HOL. Time is treated  
17 as an input variable. Like all other input variables, its value stays constant in each  
18 round. This fits well with the paradigm of reactive systems, which produce responses  
19 instantaneously.

20 The HOL terms that describe the semantics of function blocks are initially cluttered  
21 with occurrences of the evaluation functions. In a term rewriting process, which resem-  
22 bles a symbolic evaluation, these occurrences can be eliminated. This process can be  
23 largely automated. It yields HOL terms that resemble simulations of ST function blocks  
24 viewed as functional programs. In this form, the automata are suitable for verification.

25 An important aspect of our semantics is compositionality. This means that the transi-  
26 tion function of the automaton belonging to a composed block is a composition of  
27 the transition functions of the automata belonging to the components. Thus proven  
28 properties of the components can be reused. Furthermore, by abstracting over compo-  
29 nent properties, it is possible to prove properties of composed function blocks without  
30 reference to the concrete implementation of the components.

31 In addition to ST, our verification framework also deals with subsets of the two  
32 graphical IEC 61131-3 languages SFC and FBD. This is based on interpretations of  
33 these two formalisms in ST. The result is in both cases a formal semantics that is se-  
34 quential and deterministic. We will sketch the interpretation of function block diagrams  
35 below. For a semantic interpretation of SFC we refer to [33].

##### 36 4.1. Interpretation of function block diagrams in ST

37 The connection of function block inputs with outputs in a diagram induces a depen-  
38 dency relation on its components: a function block  $A$  depends on a function block  $B$

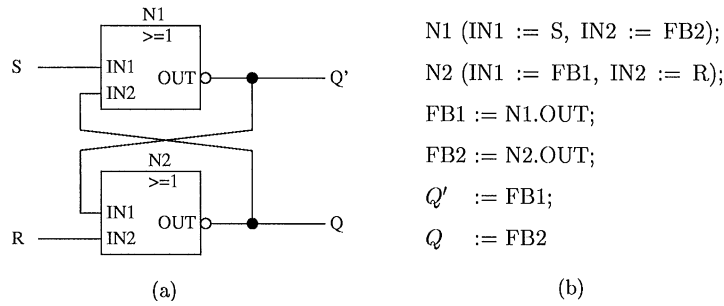


Fig. 4. Function block with feedback loop.

1 provided that at least one input port of  $A$  is connected directly or indirectly to some  
 2 output port of  $B$ . The relation is a partial order as long as the diagram does not contain  
 3 feedback loops as shown in Fig. 4(a). In the latter case, we require the user to specify  
 4 feedback variables for connections. This has the effect of a unit delay on the connec-  
 5 tions involved, i.e., the input port always receives the output value from the previous  
 6 round. In the definition of the dependency relation for function block diagrams with  
 7 feedback variables, such delayed connections are disregarded. This eliminates cycles  
 8 and ensures that the dependency relation is a partial ordering.

9 The essential step in the interpretation of a function block diagram in ST is a  
 10 serialization of the function block executions per round. The only requirement placed  
 11 on this serialization is its compatibility with the dependency ordering. This rule does not  
 12 specify the relative execution order of independent function blocks. For example, in the  
 13 function block DEBOUNCE, both DB\_ON and DB\_OFF have to be executed before  
 14 DB\_FF, but nothing is said about their relative execution order. Since we disallow  
 15 global variables in our verification framework, this under-specification does not lead to  
 16 non-determinism; in addition, the resulting semantics of a function block is not affected  
 17 by the choice of execution sequence as long as it is compatible with the dependency  
 18 ordering.

19 A chosen execution order can be translated directly to a sequence of ST func-  
 20 tion block invocations. This is followed by updates of feedback and output variables.  
 21 Fig. 4(b) exemplifies this for the case of the SR flip-flop. Here, feedback variables  
 22 FB1 and FB2 have been introduced for the connections from N1.OUT to N2.IN1 and  
 23 N2.OUT to N1.IN2.

## 5. The verification approach

25 The deep embedding of PLC programming languages in HOL provides a formal  
 26 semantics. Furthermore, the semantics given above are operational. Function blocks  
 27 can thus be evaluated symbolically using a term rewriting tool. Requirements on the  
 28 behavior of function blocks can be translated to HOL predicates and proven formally.

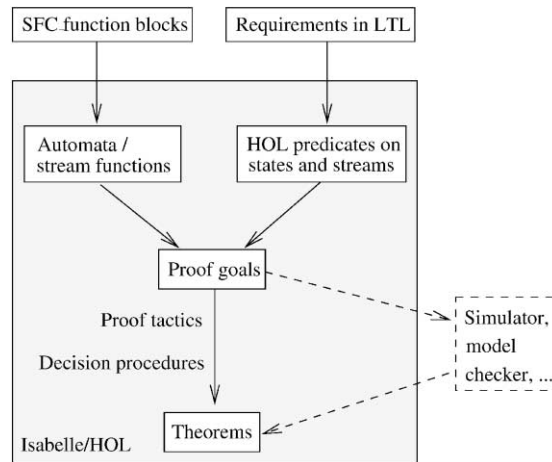


Fig. 5. Function block verification process.

1 Fig. 5 shows the verification process for SFC function blocks using linear time temporal logic (LTL [19]) as a specification language.

3 One of the strong points of the HOL based approach is its openness with respect  
 5 to possible extensions. An addition of further programming or specification language  
 7 constructs is unproblematic as long as it does not affect the underlying semantic model  
 9 of the language parts already embedded. The same remark holds true for the modeling  
 11 of machine or environment aspects, which might be necessary for the verification of  
 13 more complex systems. To put this statement more generally HOL serves as logical  
 15 glue that connects different programming and specification formalism and allows their  
 17 integration and analysis within one framework.

11 It might be interesting to investigate the integration of the Step system developed  
 13 by Zohar Manna's group in Stanford with the framework used here. Especially their  
 15 combination of model checking and theorem proving might be advantageous for the  
 17 application domain considered in this contribution.

15 In relatively small examples such as the verification of a liquid container controller  
 17 presented in [16], the standard Isabelle/HOL proof tools are sufficient. Because specifi-  
 19 cations are mapped to predicates on streams, the basic proof principle is induction over  
 21 natural numbers. In the induction step, the validity of a statement in round  $(n + 1)$  has  
 23 to be derived from its validity in round  $n$ . Induction is also essential for the proof of  
 auxiliary algebraic equalities and inequalities and the verification of iterated structures  
 such as a generic adder. Other frequent proof techniques are case distinctions, algebraic  
 simplifications and arithmetic estimations. Isabelle's classical reasoner has been very  
 useful for the automation of these kinds of proofs.

23 For more complex applications, a higher degree of proof automation is essential. This  
 25 starts off with the automated translation of function blocks into Isabelle theories. Tactics  
 specially adapted to programming or specification language constructs should be tried

1 automatically or offered interactively to the user for selection and parameterization.  
2 Relevant automated proof procedures include the symbolic model checking of finite  
3 state systems [2] and algorithms for the establishment of program invariants [9,30].

## 6. The challenge of complex dynamics

5 Until now, the use of theorem prover-based tools has been restricted to the veri-  
6 fication of systems with relatively simple continuous dynamics. This is partly due to  
7 the fact that the treatment of more complex systems would require extensive analy-  
8 sis libraries for real or complex numbers. As the pioneering work in [10] shows, this  
9 is a comprehensive task. Even with such libraries, a complete analytic verification of  
10 systems such as the two-tank laboratory plant sketched in Fig. 3 seems a daunting task.

11 Besides providing formal models of controllers and abstractions of plant properties,  
12 one useful role for deductive proof tools in this area might be the validation of inter-  
13 polation and extrapolation properties. These guarantee that nothing unexpected happens  
14 for parameter combinations that have not been explicitly covered during simulation or  
15 model checking. This validates intuitive worst-case reasoning and increases the trust-  
16 worthiness of verification results.

## 17 7. Towards an industrial strength tool

18 The theorem prover-based verification technique presented in Section 5 requires spe-  
19 cial skills from the quality assurance personnel because the proof assistant relies on  
20 sophisticated user guidance. These skills cannot be expected from engineers in the field.

21 Conversely, people with skills in formal specification and verification techniques  
22 normally lack the domain expertise needed to understand functional and safety re-  
23 quirements that are often not made explicit and, if so, are usually presented in an  
24 incomplete, ambiguous and informal manner. In the course of our work with the IEC  
25 standard, its German counterpart, and the in-house standard and function block library  
26 of a manufacturer in chemical industry, we spent days and weeks in reading through  
27 these documents and many hours talking to domain experts to fully understand the  
28 requirements.

29 Hence, to make the verification approach the presented work in automation practice,  
30 we need to find effective means to solve the following three tasks:

- 31 (1) comprehensive elicitation of functional, safety, and—if appropriate—timing re-  
32 quirements;
- 33 (2) formalization of these requirements in a suitable logic; and
- 34 (3) correctness proof.

35 The set of standard function blocks that are typically used in specific control domains  
36 ranges between 50 and a few hundreds and the complexity of the majority of function

1 blocks maintained in the domain library is relatively low. As these library components  
2 are verified only once but used many times, the effort to have these tasks performed  
3 by computer theoreticians is acceptable.

4 However, for handling individual control applications, which are composed of net-  
5 works of function blocks, we need to wrap an open verification environment with a  
6 front-end that is usable by domain experts. This verification environment may use a  
7 theorem prover as its backbone and comprise other tools such as model checkers, sim-  
8 ulators or computer algebra systems. The interface to the front-end must be capable of  
9 eliciting enough facts about critical application requirements through a series of com-  
10 munication interactions with domain experts such that formal requirement statements  
11 can be derived. Such dialogs need to know about the terminology of the field, they  
12 may rely on a collection of known requirements typical for that domain, and they may  
13 exploit proven properties of function blocks connected to the application interface and  
14 the inner “wiring” of the application program to conduct that dialog. It may also exploit  
15 paraphrasing capabilities to verify the adequacy of formalized requirement statements  
16 acquired in earlier communications. The work on knowledge intensive software engi-  
17 neering tools conducted by Rich and Waters (cf., e.g. [28]) might provide prototype  
18 solutions for the engineering environment sketched here.

19 To facilitate the verification task, it is also important to find proof patterns and  
20 reusable proof strategies to automate recurring verification steps. In this respect, the  
21 integration of automatic model checking procedures such as pioneered by Shankar for  
22 PVS seems particularly promising.

23 To come up with usable solutions, a close co-operation with interested vendors, users  
24 and evaluators for PLC controllers in safety critical fields is urgently needed.

25 In [16,33] we have used higher order and linear time logic to specify the functionality  
26 and critical properties of function blocks. In [21] a master student at the University of  
27 the Witwatersrand, Johannesburg, has recently documented the function block interfaces  
28 of the German FB standard [32] using Parnas’ table specification technique [25], Z,  
29 and classical pre-/post-conditions. A field study, which elaborates whether either of  
30 these alternative documentation techniques is easier to read and write for engineers, is  
31 still to be seen.

## 8. Conclusion

33 The main body of this paper has presented a theorem prover-based interactive veri-  
34 fication technique that supports compositional correctness and safety proofs of PLC  
35 programs expressed in the IEC 61131-3 languages FBD, SFC and ST. These custom-  
36 designed languages have been widely accepted by engineers constructing software-based  
37 industrial control systems.

38 We have particularly focused on safety-related application domains in which soft-  
39 ware faults and defects must be avoided to reduce the risk for damages to persons  
40 or property. These high dependability requirements justify the extra effort needed for

1 the explicit modeling of a function block's or control program's externally observable  
2 behavior, the specification of safety constraints and their formal verification. We have  
3 used higher order logic predicates and LTL operators on input and output streams for  
4 this purpose. For larger applications it is essential that the verification technique be  
5 modular. This allows the reuse of verified properties of components in the verification  
6 of composite functions blocks and control applications.

7 We have also argued that an inherent problem of our approach is the degree of  
8 detailed knowledge about the working of the proof assistant tools and the formal rep-  
9 resentation of programs and requirements. This kind of expert knowledge cannot be  
10 expected from automation engineers in the field. Therefore, we stressed the need to  
11 find ways to automate recurring development steps and generate a great deal of the  
12 supporting Isabelle theories automatically from specifications and programs to be veri-  
13 fied. An ergonomic design of a development environment matching the work processes  
14 in development laboratories and licensing authorities is the next step to be taken in  
15 close co-operation with interested automation industries.

## References

- 17 [1] R. Alur, T. Henzinger, Reactive modules, in: Proc. 11th Annu. IEEE Symp. on Logic in Computer  
18 Science, IEEE Computer Society Press, Silver Spring, MD, 27–30, 1996, pp. 207–218.
- 19 [2] R. Alur, T. Henzinger, P.-H. Ho, Automatic symbolic verification of embedded systems, *IEEE Trans.*  
20 *Software Eng.* 22 (3) (1996) 181–201.
- 21 [3] P. Andrews, *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof*,  
22 Academic Press, New York, 1986.
- 23 [4] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, J.V. Tassel, Experience with embedding  
24 hardware description languages in HOL, in: V. Stavridou, T. Melham, R. Boute (Eds.), *Theorem Provers*  
25 *in Circuit Design*, Proc. IFIP TC10/WG 10.2 Internat. Conf. Nijmegen, June 1992, North-Holland,  
26 Amsterdam, 1992.
- 27 [5] P. Curzon, The formal verification of the fairisle ATM switching element, Technical Report 329,  
28 Computer Laboratory, University of Cambridge, 1994.
- 29 [6] M. Gordon, T. Melham, *Introduction to HOL*, Cambridge University Press, Cambridge, 1993.
- 30 [7] W.A. Halang, B.J. Krämer, L. Trybus, Exploiting a graphical programming paradigm to facilitate  
31 rigorous verification of embedded software, *Comput. J.* 38 (4) (1995) 301–309.
- 32 [8] N. Halbwachs, *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers,  
33 Dordrecht, 1993.
- 34 [9] N. Halbwachs, Y. Proy, P. Roumanoff, Verification of real-time systems using linear relation analysis,  
35 *Formal Methods System Des.* 11 (2) (1997) 157–185.
- 36 [10] J. Harrison, Theorem proving with the real numbers, Ph.D. Thesis, Computer Laboratory, University of  
37 Cambridge, 1996.
- 38 [11] J. Harrison, Floating point verification in HOL Light: the exponential function, Technical Report 428,  
39 Computer Laboratory, University of Cambridge, 1997.
- 40 [12] T. Heckenthaler, S. Engel, Approximately time-optimal control of a two-tank system, *IEEE Control*  
41 *Systems Mag* 14 (3) (1994).
- 42 [13] IEC International Standard 1131-3, *Programmable Controllers*, part 3: *Programming Languages*, 1993.
- 43 [14] International Electrotechnical Commission, On-line at <http://www.iec.ch>.
- 44 [15] S. Kowalewski, M. Fritz, H. Graf, S. S. J. Preußig, O. Stursberg, H. Treseler, A case study in tool-aided  
45 analysis of discretely controlled continuous systems: the two tanks problem, in: Proc. 5th Internat.  
46 Workshop on Hybrid Systems (HSV), Notre Dame, USA, September 1997.
- 47 [16] B.J. Krämer, N. Völker, A highly dependable computer architecture for safety-critical control  
48 applications, *Real-Time Systems J* 13 (1997) 237–251.

- 1 [17] L. Lamport, L. Paulson, Should your specification language be typed? Technical Report 425, Computer  
Laboratory, University of Cambridge, May 1997.
- 1 [18] N.G. Leveson, C.L. Turner, An investigation of the Thereac-25 accidents, *IEEE Comput.* 26 (7) (1993)  
18–41.
- 3 [19] Z. Manna, A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer,  
Berlin, 1992.
- 5 [20] R. Milner, M. Tofte, R. Harper, D. MacQueen, *The Definition of Standard ML (revised)*, MIT Press,  
Cambridge, MA, 1997.
- 7 [21] S. Morton, Specifying VDI/VDE 3696 function blocks, MS Thesis, Department of Computer Science,  
University of the Witwatersrand, Johannesburg, 1998.
- 9 [22] T. Nipkow, Higher-order unification, polymorphism, and subsorts, in: *Proc. 2nd Internat. Workshop  
Conditional and Typed Rewriting Systems, Lecture Notes in Computer Science*, Vol. 516, Springer,  
11 Berlin, 1991.
- [23] T. Nipkow, D. von Oheimb, *Java/ight* is type-safe—definitely, in: *Proc. 25th ACM Symp. Principles of  
13 Programming Languages*, ACM Press, New York, 1998.
- [24] OMG, CORBA/IIOP 2.2 Specification, 1998, On-line available via <http://www.omg.org>.
- 15 [25] D. Parnas, J. Madey, M. Iglewski, Formal documentation of well-structured programs, Technical  
Report CLR 259, Communications Research Laboratory, Faculty of Engineering, Mc Master University,  
17 Hamilton, September 1992.
- [26] L. Paulson, in: *Isabelle: A Generic Theorem Prover, Lecture Notes in Computer Science*, Vol. 828,  
19 Springer, Berlin, 1994.
- [27] W. Reisig, in: *Petri Nets, EATCS Monographs on Theoretical Computer Science*, Vol. 4, Springer,  
21 Berlin, 1995.
- [28] C. Rich, R. Waters, Knowledge intensive software engineering tools, *IEEE Trans. Software Eng.* 4 (5)  
23 (1992) 424–430.
- [29] J. Rushby, D. Stringer-Calvert, A less elementary tutorial for the PVS specification and verification  
25 system, Technical Report SRI-CSL-95-10, Computer Science Laboratory, SRI International, Menlo Park,  
CA, 1995.
- 27 [30] H. Saïdi, The invariant checker: automated deductive verification of reactive systems, in: O. Grumberg  
(Ed.), *Computer-Aided Verification, CAV '97, Lecture Notes in Computer Science*, Vol. 1254, Springer,  
29 Berlin, 1997.
- [31] S. Tahar, R. Kumar, Formal specification and verification techniques for RISC pipeline conflicts,  
31 *Comput. J.* 38 (2) (1995) 111–120.
- [32] VDI/VDE Richtlinie 3696, Herstellerneutrale Konfigurierung von Prozessleitsystemem; Allgemeines zur  
33 herstellerneutralen Konfigurierung 1995 (in German).
- [33] N. Völker, Ein Rahmen zur Verifikation von SPS-Funktionsbausteinen in HOL, Ph.D. Thesis,  
35 FernUniversität Hagen, Shaker Verlag, 1998 (in German).