Programming languages for use in safety-related applications

Wolfgang A. Halang a,∗, Janusz Zalewski b

a Fernuniversität, Faculty of Electrical and Computer Engineering, D-58084 Hagen, Germany
b Florida Gulf Coast University, Computer Science Program, Fort Myers, FL 33928, USA

Abstract

Programmable electronic systems are being used in almost all application sectors to perform non-safety and increasingly to perform safety functions as well. Although software-based solutions are usually superior to hardwired ones for reasons of efficiency and flexibility, there is a certain reluctance of the certification authorities when it comes to licensing computer-based systems which are classified as safety critical. Despite many attempts to overcome problems of software safety (IEC 61508, IEC 880, VDE 0801, IDS 00-55, RTCA/DO-178), up to now neither precise guidelines supporting the software development process are available, nor are there serious efforts being made to develop programming languages dedicated to the implementation of safety critical functions. To improve this unsatisfactory situation, i.e. to meet both economic and safety requirements, it is necessary to design appropriate language concepts with consequent regard to safety aspects. Accordingly, four subsets of a real time language suitable for the implementation of safety-related systems are proposed, whose definitions fulfill the respective requirements of the four safety integrity levels.

© 2003 Elsevier Ltd. All rights reserved.

Keywords: Safety-related systems; Safety requirements; Safety integrity levels; Real time programming languages; PEARL; Verification

1. Introduction

Computer-based solutions are permanently gaining significance in almost all application fields, not least because of the rapid progress in microelectronics enabling cheap and flexible implementations. Although it is quite natural that industry tends to benefit from these economic advantages, there is, however, a type of system designated as safety related for which this trend does not prevail. The reason for this is to be found in the growing awareness of safety hazards, and as a consequence of the demand for highly dependable systems. The latter holds true especially if malfunctions of the concerned systems may result in disastrous accidents, as it is characteristic for automation systems. Software-based solutions are still considered to be less trustworthy than conventional hardwired components, which is justified by the longer tradition of hardware and, therefore, many years of experience in the development of strategies to cope with corresponding failures. Actually, evaluating software with respect to dependability or safety aspects is a task which has been neglected for a long time. However, owing to various efforts during recent years, there exists a number of methods and guidelines belonging to different application fields and, among others, having the principal intention to define requirements for programming languages for use in safety-related systems (DIN, 1995; IEC, 1986, 1998; RTCA, 1992; MOD, 1991). The guidelines agree on the following general requirements:

• programming languages should be fully and unambiguously defined or shall possess a formally defined syntax, respectively,
• be user and problem oriented,
• be of a high level with preference to safety oriented standardised subsets,
• enable structured programming, restricting branches and loops, and avoiding jumps and computed branches,
• should support simple and uniform addressing techniques, and
• strong typing, i.e. enforce explicit declaration, initialisation and type conversions,
• but not comprise dynamic variables or other dynamic objects,
• language structures should be modular, interfaces between modules should be simple, uniform and fully defined, and
• procedures should be simply structured, having a minimum number of parameters, only; communication should exclusively be permitted via parameters; recursive procedure calls should be not allowed.

* Corresponding author. Tel.: +49-2331-987-372; fax: +49-2331-987-375.
E-mail addresses: wolfgang.halang@fernuni-hagen.de (W.A. Halang), zalewski@fgcu.edu (J. Zalewski).

1367-5788/03/$ – see front matter © 2003 Elsevier Ltd. All rights reserved.
doi:10.1016/S1367-5788(03)00005-1
Unfortunately, these guidelines are restricted to the description of shortcomings or desirable features with respect to existing languages. They are often rather imprecise, thus not offering substantial support during system development. Actually, no new language has been developed on the basis of these standards, nor have the mentioned guidelines so far motivated the re-design of existing languages for use in safety-related applications. Consequently, even though the process of verification is inevitable for any code segment implementing a safety-related function, no progress based on the cited standards is evident or is to be expected.

Indeed, because verification is the main prerequisite to enable the certification of larger software-based solutions, only serious improvements aiming to support the process of program verification will be a step in the right direction. The importance of this becomes even more evident when we consider the fact that for practicability reasons the workload involved in verifying safety-related software has to be restricted to a reasonable limit. The present state of the art implies, for example, that formal methods only apply for such verification processes when the code sequences are quite short.

Summing up, up to now approaches aimed at improvements in the field of software safety obviously have not yielded results which enhance the confidence in software-based solutions. The certification authorities are, therefore, still reluctant to license exclusively software based safety-related systems and, consequently, system designers often prefer to employ conventional hardwired solutions despite their manifest disadvantages with respect to the mentioned economic aspects.

In an attempt to remedy this unsatisfactory situation, in this paper we introduce several programming language concepts which we think stand a better chance of enhancing trustworthiness and dependability of software based safety-related systems. After reviewing the state of the art in programming languages for safety-related systems, we discuss some inherent characteristics of software, focusing particularly on the differences with respect to the nature of software and hardware failures. Then, we present a basic principle which is decisive in the development of new language concepts for safety-related programming. The following section defines four increasingly restricted subsets of the real time programming language PEARL 90 (DIN, 1998), recommended for use in safety-related applications, and classifies these subsets according to their suitability with reference to the four discrete levels of increasing safety integrity requirements (SIL 1, . . . , SIL 4) introduced in the international standard IEC 61508-1 (IEC, 1998).

2. Programming languages and safety

There has been a number of studies in recent years, regarding the features of programming languages suitable or unsuitable for programming safety critical systems. For practical reasons we are mostly interested in languages already used on an industrial scale, rather than creating new ones with improved features, such as in Schlichting and Thomas (1995). General issues have been studied in several papers by Cullyer, Goodenough, and Wichmann (1991), Wichmann (1992, 1994) and in an report of the US Nuclear Regulatory Commission NRC (Hecht et al., 1996). They point to several important issues and questions to be asked when choosing a language for programming this kind of systems. The catalogue of issues includes:

- formally defined syntax,
- block structure,
- strong typing,
- wild jumps,
- memory overwrites (e.g. due to unchecked array bounds),
- memory exhaustion (e.g. due to memory leaks),
- dangling pointers,
- a model of integer and floating point arithmetic which is safe to deal with problems such as integer overflows and the approximate nature of floating point arithmetic (causing rounding errors, etc.),
- variable initialisation,
- potential assignments in Boolean expressions,
- exception handling,
- separate compilation (with type checking across modules),
- and
- temporal predictability, addressing such problems as the difficulty to determine the execution time of while loops.

One of the languages most commonly used in safety-related applications and, therefore, the most often evaluated one in this respect is C. It has been studied by several authors, including Hatton (1995) and Lindner (1998). In addition to the general issues listed above, a number of difficulties with C have been pointed out:

- very weak typing,
- severe problems with dynamic memory allocation and pointer arithmetic,
- logical expressions are defined, but a logical data type is not,
- potential ambiguities in if conditions,
- increment/decrement operators, etc.

As a result, at least one industry (automotive in Great Britain) produced a set of guidelines to standardise the use of C in safety-related applications, known as MISRA C (1998). This document includes over a hundred recommendations, named rules, that describe desirable and undesirable uses of various C constructs. For instance, one of the rules in the operators category reads as follows:

**Rule 35 (required).** Assignment operators shall not be used in expressions returning Boolean variables.

Ada is another language often used on an industrial scale in safety-related systems, mostly in military applications. A
3. Causes and nature of software failures

Although always implemented with the latest technology, the architecture of digital computers has remained essentially unchanged since the times of Konrad Zuse and John von Neumann in the 1940s. The prevailing architectural model matches the technical possibilities of that time. In other words, while the technology with which hardware is built and the capabilities of hardware have tremendously advanced within the last 60 years, all high level programming languages developed so far, and presently in use, reflect to different extents the way programs are executed by von Neumann machines, allowing maximum flexibility at the price of facilitating errors and being unsafe. As a consequence of this and due to the inflationary increase in the demand for functionality, the complexity of software is exploding exponentially. This, in turn, leads to error prone software development and results in a lack of software dependability.

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

When analysing malfunctions of computer based safety-related systems, it can be observed that often the reasons are not to be found in random hardware failures, i.e. failures occurring at random times or resulting from a variety of degradation mechanisms in the hardware. Rather, they result from special system states during operation, which had not been considered in the specification or implementation phases of the software. Naturally, programs are immaterial and, consequently, do not degrade in a similar way...
to hardware components. Software failures, which must be classified as systematic, are due to errors, mistakes or omissions in any development life cycle phase. Systematic failures include (IEC, 1998):

- failures due to errors in the safety requirements specification, and
- failures due to errors in the design, implementation, etc. of software.

Thus, provision against these types of failures should already have been taken into account in the early phases of system development.

Software has to be valid and correct. The latter means that software has to fulfill a given problem specification, i.e. it must exclude systematic failures based on program design, coding, or the use of software tools. Correctness of software cannot be achieved by testing, reviews, audits, inspections, walkthroughs or other heuristic methods, because these inherently lack the rigor necessary to be able to detect all errors contained in a program. And, since by definition, correct software meets its specification, establishing software correctness alone is not sufficient. When a program does not behave as expected, often its very specification is incorrect, incomplete or inconsistent. Thus, it and the resulting software cannot be valid. Consequently, to produce proper software-based systems it is essential to invest high validation efforts already in the specification phase. Unfortunately, at this stage there are no usability checklists or measures for assessing whether a specification is valid.

4. Fostering safety by simplicity

Although sometimes supported by computerised tools, the proof of a program’s correctness depends for its success on human cognizance. In order to minimise the cognitive effort necessary to produce it, and to maximise its trustworthiness, it is essential to employ user oriented programming concepts. By their very nature, programming languages are machine-computer interfaces par excellence. This property is seldom recognised, possibly due to the programmers’ familiarity with the von Neumann architecture and their acquired comfort with machine oriented interfaces. Object oriented and visual higher level programming languages have been developed in parallel with, and reflecting the need for, graphical user interfaces for a wider population of end users. However, in order to support the process of system verification as far as possible, we need to go a step further and develop a new brand of “subject oriented” rather than “object oriented” programming languages to ensure that programming and non-programming system developers and certification licensors can be given a common ground at the interface.

The assertion that a program is apparently free of software faults is the major prerequisite for granting a safety licence by authorised certification institutions. Since everybody grasps a simple thing instantly and as intended, to enable verifiability of software, we have chosen simplicity as an appropriate, fundamental and particularly human oriented design concept. The ease with which simple systems are understood and their clear and distinct behaviour when executed is a pre-condition to achieve social consensus which constitutes the essence of verification. The fact that everybody understands an item under consideration immediately as intended increases the trustworthiness of the result. Thus, simplicity is a fundamental feature to increase the confidence in computer-based systems.

If varied pragmatic interpretation is possible or required due to vagueness or ambiguity of expression, an interpreted item is not simple. This leads to confusion as, unless specifically prevented, people seldom interpret the same thing in the same way. Conversely, we can conclude that an item of consideration is simple, if its meaning can be taken literally as given and requires no extra cognitive processing effort to arrive at. Applied to software, a program may be regarded as simple if it can be verified in a mechanised way. Such a mechanised software verification is a complete test, because it would not require intellectual understanding and can be carried out by a computer.

Irrespective of the fact that in reality strict causality is often questionable, people usually think in terms of actions or events and their results, which they tend to perceive as causes and effects. Therefore, cause–effect tables or decision tables appear to be highly appropriate as a form of software representation oriented at human cognition. This is further substantiated by the property that tables are visual means. If such diagrams of causes and effects are simple and follow accepted ergonomic standards, everybody will develop the same mental model of the operations of the formulated software. This cannot be guaranteed if text or words are used for programming since people tend to use their idiosyncratic experience to interpret them. Hence, visualisation is a condition sine qua non of interface design supporting shared understanding across a varied user population. As they have finite size, cause–effect tables also lend themselves to a complete test. Because of their more accurate fit to human cognition and because the results of complete tests are absolutely sure, programs thus verified may be permitted to be employed in automation systems which have to meet the requirements of Safety Integrity Level 4.

5. Safety integrity levels and inherently safe languages

Guided by the principle of simplicity, but relaxing the requirements gradually, we now define a sequence of programming language subsets for implementing software that has to fulfil the demands of Safety Integrity Level 4 through 1.

5.1. PEARL 90

The textual language PEARL 90 (DIN, 1998) is one of the very few genuine high level real time languages. Owing to
its clear concepts and its modular, block oriented structure, PEARL 90 is especially suitable for industrial process control applications and is also increasingly used in academia for teaching purposes. Its algorithmic part has approximately the same facilities as Pascal. Additionally, PEARL 90 provides the data types clock an duration and the following statements for task scheduling and control:

- activate for task activation,
- terminate for task termination,
- suspend for temporary blocking of a task,
- continue for releasing a suspended task,
- resume for the combination of suspend and continue, and
- prevent for annihilation of all scheduled future activations of a task.

The above statements are coupled to time related or event driven scheduling conditions. The synchronisation mechanism and the protection of shared data in the language are based upon a concept of generalised semaphores. Exception handling is founded on internal signals in an unstructured manner. PEARL 90 defines a generic input/output concept in the form of so-called DATIONS (DATA tATIONs), which serve the specification of device topology, data access, control information etc.

### 5.2. Enhancing safety

In the following sections, we offer four nested subsets of PEARL aimed at enhancing the safety of real time control software, and to be viewed as potential candidates for consideration in this domain. These subsets are associated with the safety integrity levels as follows:

<table>
<thead>
<tr>
<th>SIL</th>
<th>PEARL subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIL 1</td>
<td>HI-PEARL</td>
</tr>
<tr>
<td>SIL 2</td>
<td>Safe PEARL</td>
</tr>
<tr>
<td>SIL 3</td>
<td>Verifiable PEARL</td>
</tr>
<tr>
<td>SIL 4</td>
<td>Table PEARL</td>
</tr>
</tbody>
</table>

Although the subsets are traditional procedural languages, they are simple and subject oriented. Oriented at the absolutely necessary, they comprise the simplest and most well understood language constructs, only. The four PEARL 90 subsets are becoming progressively restrictive towards higher safety integrity levels. The assignment of the subsets to the different safety integrity levels results from the simplicity and clarity of the verification methods available for the programming paradigms on which the single languages are based. As outlined above, the simplicity of the software verification process directly correlates to the trustworthiness of computerised systems. Basically, the language subsets differ in some "unsafe" language features that are gradually prohibited. The main advantage of this approach is that one does not have to learn a different language for each safety integrity level, and that the compiler can help to verify if the programs respect certain safety requirements. The use of language subsets to implement safety critical systems also allows to mix code of different levels of safety integrity, which enables a seamless connection between safety critical and uncritical parts of a system.

### 5.3. Table PEARL

The subset Table PEARL defined below consists of only one executable statement allowing the formulation of the rules which constitute cause–effect tables. As was already pointed out in Section 4, software represented in this form can be verified with the highest certainty. Therefore, it is licensable for applications having to meet Safety Integrity Level 4. For the formal definition of this and the other subsets we use an extension of the classical Backus Naur form (EBNF), brackets [ ] to denote optional syntactical expressions, as well as the symbols * and + to denote repetition of the correspondingly marked expressions at least zero times or once, respectively.

#### Table PEARL

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨module⟩</td>
<td>MODULE [(var-decl)<em>]</em></td>
</tr>
<tr>
<td>⟨var-decl⟩</td>
<td>DCL (name) : (type);</td>
</tr>
<tr>
<td>⟨rule⟩</td>
<td>IF [bool-expression] THEN (variable) := (expression) FI N</td>
</tr>
</tbody>
</table>

### 5.4. Verifiable PEARL

The derivative Verifiable PEARL defined in the following table comprises just those language constructs which are necessary to program in a textual form the interconnection patterns of Function Block Diagrams, as defined in IEC 61131-3 (IEC, 1992), in other words, parameter passing and procedure invocation. Corresponding programs are rather easy to verify with rigorous methods of high trustworthiness (Halang, Frigeri, Lichtenecker, Steinmann, & Wendland, 1998). Hence, they can be certified for Safety Integrity Level 3.

#### Verifiable PEARL

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨module⟩</td>
<td>MODULE [(name)]</td>
</tr>
<tr>
<td>⟨problem⟩</td>
<td>⟨problem⟩ MODEND;</td>
</tr>
<tr>
<td>⟨proc-spec⟩</td>
<td>⟨var-decl⟩)/(proc-spec)/+;</td>
</tr>
<tr>
<td>⟨proc-stmt⟩</td>
<td>SFC (name) : PROC [bit-of-par]</td>
</tr>
<tr>
<td>⟨bit-of-par⟩</td>
<td>DCL (name) : (type);</td>
</tr>
<tr>
<td>⟨par⟩</td>
<td>⟨par⟩/(par)/+;</td>
</tr>
<tr>
<td>⟨const⟩</td>
<td>⟨const⟩[⟨var-name⟩]</td>
</tr>
</tbody>
</table>
5.5. Safe PEARL

Only in very rare cases all features of high level programming languages are necessary to formulate the functionalities required in control engineering. Therefore, in the following table we define an inherently safe language subset restricted to those executable statements which are really necessary and, thus, indispensable, viz., procedure calls, assignments, conditional selections, and loops with bounded numbers of iterations. The correctness of programs formulated with this subset can be proven with adequate tool support by, for instance, higher order logic, which is a typified variant of Church's higher predicate logic. Therefore, this subset qualifies for usage under the conditions of Safety Integrity Level 2. The attribute READ within the declaration of an interface variable ensures that other modules are only permitted to read the declared variable but not to write it. Variables, procedures and types are just names within this language subset. Expressions contain the usual Boolean and arithmetical operations.

<table>
<thead>
<tr>
<th>Safe PEARL</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Language Subset" /></td>
</tr>
</tbody>
</table>

6. HI-PEARL

The subset H(igh)l(ntegrity)-PEARL defined in Halang and Stoyenko (1993) preserves the obvious advantages of the original language, but is much more suitable for implementing safety-related software, especially with respect to schedulability, since it is verifiable with respect to its functionality and timeliness. As a consequence, HI-PEARL constitutes a programming language feasible for implementing systems classified as safety related on SIL 1.

For formulating critical regions, the structured LOCK mechanism, with a time-out clause, is introduced and the concept of semaphores is removed. The time-out clause ensures that the waiting time before entering a critical region is limited. In case that the lock cannot be carried through before this time limit is exceeded, an alternative action takes place. The mentioned time-out clause and the lock's execution time limit enable the determination of an upper time execution limit for each lock statement at compile-time. To ensure that no synchronisation or input/output operation takes arbitrarily long—in case multiple processes claim a shared resource—shared objects may only be referenced within the framework of a LOCK statement. To prevent the formation of arbitrarily long cascades of exceptions and interrupts, with each exception specification a constant value is associated, bounding the number of occurrences and the time for handling the exception within one single activation of the process provoking it. The minimum time period between two interrupt signals on one line equivalent to the maximum frequency of the interrupt signal has to be stated in the interrupt’s specification. To disable iterations that prohibit an analysis of schedulability of programs and, hence, lead to undeterministic behaviour WHILE and REPEAT loops are removed. Instead, iteration statements have to explicitly specify their execution times and the maximum number of repetitions. The GOTO statement which is unsuitable for structured programming as well as directly or indirectly recursive procedure calls are banned.
7. Conclusion

Despite the existence of standards like IEC 61508-1 or DIN V VDE 0801, software-based systems are still considered to be less trustworthy than conventional hardware components, which is justified in part by the longer tradition of hardware engineering and many years of experience in the development of strategies for coping with corresponding failures. This situation is due to

- the standards being unclear about the characteristics of environments for safety critical programming,
- no programming language (and related programming environment) based on these guidelines being available.

To overcome this situation, a dedicated subset of PEARL 90 has been proposed for each of the four safety integrity levels defined in IEC 61508-1, following the principle of simplicity: the more safety critical a system is, the more simple the related control software needs to be. Accordingly, on the highest Safety Integrity Level (SIL 4) the safety/integrity of a “software” system can be verified just by looking at a Cause Effect Table, i.e. without employing formal proofs. Control software for SIL 3 systems can be graphically constructed, based on already proven function blocks. This will simplify the process of software development as compared to textual languages and still keep the task of safety proofs relatively easy. Programs written in the subset for SIL 2 have the main advantage of being formally verifiable. Finally, the subset HI-PEARL for the lowest Safety Integrity Level (SIL 1) can be easily analysed for schedulability, which is assured by eliminating some “unsafe” language constructs such as GOTO, WHILE and REPEAT. Although they are not as simple as the subset for SIL 2, HI-PEARL programs are verifiable as well.

However, much more important than the technical details mentioned above is the realisation of the important role the human component plays in the process of engineering safe computerised systems, which can and will lead to future research. In this paper we have drawn attention to the fact that programming languages are best perceived as human–computer interfaces which are subject to ergonomic principles that need to be followed in design of end user interfaces. We suggested, therefore, that higher level programming languages ought to be “subject” (i.e. human user) rather than “object” (i.e. system) oriented. We have identified simplicity as a fundamental design principle for programming languages, fostering human cognitive performance and, thus, safety.

References


