

Enhancing UML by Safety-Related Constructs

Shourong Lu

Shourong.Lu@FernUni-Hagen.de

Abstract: To enable the description of safety-related software, the Unified Modeling Language is provided with well proven constructs as found, e.g., in corresponding subsets of the real-time programming language PEARL and in Function Block Diagrams according to IEC 61131-3. These constructs are ordered in nested sets to fulfill the respective requirements of the four Safety Integrity Levels of IEC 61508. By adding to UML safety elements oriented at the prevailing safety standards, UML can be employed to design dependability structures for safety-critical systems. The safety elements are described in UML notations, and collected in a profile which can be incorporated into models of safety-related embedded real-time control systems.

1 Introduction

Industrial control systems are increasingly becoming more complex and software-intensive. Safety ought to be on top of the agenda right from the start, as it is an integral part of a system's design. Hazard and risk analyses need to be performed, the specific safety requirements identified, and systems as well as dependability structures certified against safety standards before being put into operation.

Over the years, several standards for safety-related applications appeared, such as IEC 880, RTCA DO-178B, or the British Defence Standards 00-55 and 00-56, which specify how software and systems should be developed in order to adhere to the requirements of specific industrial areas. The standard most often referred to for computerised systems is IEC 61508 [10], concerned with the safety of software-intensive systems. Here, Safety Integrity Levels (SILs) play an important rôle, to give a certain form to safety requirements and their implication on the development of software. Good design of software architectures can help to meet SIL-specific demands, but also requirements with respect to performance, reliability, portability or interoperability.

The Unified Modeling Language (UML) [12] offers an unprecedented opportunity to develop safety-critical systems. In UML, the paradigms of modern software development are integrated into a comprehensive and widely accepted visual language. The language can be used to construct software architecture specifications, often implementing safety-critical functions. Its built-in extensibility is a powerful feature of UML, providing mechanisms like stereotypes, tagged values and constraints with which the semantics of model elements can be customised and extended. Employing UML's genuine extension mechanisms, stereotypes can be newly defined to incorporate inherently safe elements into the

framework of UML, and collected in a profile of genuine patterns providing specifications of safety properties. Therefore, it is straightforward to consider UML for modeling and analysing safety features for critical systems.

Here, UML is extended by constructs oriented at the well designed and industry-proven facilities available in the “Process and Experiment Automation Realtime Language” PEARL [3, 4], which has been enhanced towards distributed systems and object-oriented design of applications having to meet severe safety requirements [8], but also at the high-level graphical language Function Block Diagram for programmable logic control as defined in the international standard IEC 61131-3 [9], aiming to exploit the ideas incorporated in these languages and the safety elements of the cited standards for the model-based and object-oriented development of safety-related embedded real-time control systems.

2 SIL-related Programming Language Structures

In the international standard IEC 61508 four distinct levels of safety integrity were introduced and assigned, respectively, to specify the safety requirements of safety-related systems, and in [5] evaluations were reported stating the suitability of programming languages and software verification methods for safety-related control application having to meet the requirements of the different SILs. For safety-related programming, language subsets ought to be used which comprise only constructs inherently safe and easy to understand, and leave all other features out. The advantage of this approach is that the bounds of a subset can be flexible, to allow for the use of some features in a limited and controlled way. Such uses must be carefully designed, controlled and justified, to ensure that the integrity of the software is not adversely affected.

To provide a common framework for textual programming, increasingly restrictive, nested subsets of PEARL were defined in [8] for each of the four Safety Integrity Levels, viz., HI-PEARL, Safe PEARL, Verifiable PEARL and Table PEARL. In the latter subset cause-effect tables controlling systems with SIL 4 requirements are formulated, which can be verified by visual inspection instead of employing formal proofs. Control software for SIL 3 systems is constructed graphically based on already proven function blocks, and the interconnection patterns are verified by inspection. In the subset Verifiable PEARL such graphical programs can be expressed textually. The subset Safe PEARL for SIL 2 enables formal program verification, and programs formulated in the subset HI-PEARL for the lowest SIL 1 can easily be analysed for schedulability, which is assured by eliminating some unsafe language constructs such as jumps and unbounded loops. Although not as easily as their SIL 2 counterparts, HI-PEARL programs are verifiable as well.

3 Incorporating PEARL Concepts into UML

The four safety-related subsets are shown in Fig. 1. Table PEARL provides just a single executable statement, which is used to formulate rules constituting cause-effect tables.

Matching FBD, the constructs of the textual language Verifiable PEARL allow for parameter passing and procedure invocation, only. Safe PEARL is an inherently safe language subset restricted to the executable statements procedure call, assignment, conditional selection, and loop with a compile-time bounded number of iterations. Finally, the subset HI-PEARL meeting the requirements of SIL 1 was derived from PEARL by eliminating all unsafe language constructs which may lead to undeterministic behaviour.

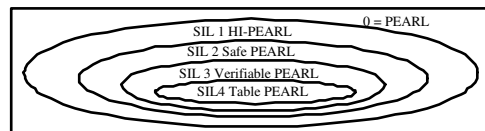


Table PEARL

```
<module> ::= Module [<var-decl>]* [<rule>]* MODEND ;
<var-decl> ::= DCL <name> : <type>;
<rule> ::= IF <bool-expression>
  THEN
    <variable> := <expression>
  FIN
```

Verifiable PEARL

```
<module> ::= MODULE [( <name> )]
  [( <problem> )] MODEND

<problem> ::= PROBLEM
  [<proce-spec>]* [<var-decl>]* [<task-decl >];
<proce-spec> ::= SPC <name> : PROC <lst-of-par> <return-type>;
  [GLOBAL] ;
<var-decl> ::= DCL <name> : <type>;
<task-decl> ::= <name> : Task[GLOBAL]; <proc-stmt>+ END
<proc-stmt> ::= <proc-name> <lst-of-par>
<lst-of-par> ::= (<par> [<par>]*)
<par> ::= <const> | <var-name>
```

Context PearlInterface

```
inv: self.feature -> forAll (f1
  (f.parameter -> size >= 1) &
  (f.operation = parameter transportation))
```

Context PearlPort

```
inv: self.owned. Port () implies
  (self.required -> size () = 1) &
  (self.provided -> size() = 1)
```

PEARL Collection:

```
Context PearlCollection
inv: self.baseClass = Component
self.ownedElement. IsInstantiable = true
self.ownedElement.contents -> forAll (ml
  m.OcIsKindOf( Module) &
  m.Port.isKindOf( PearlPort))
```

HI-PEARL Module for SIL1

```
Context PearlModule inv:
self.baseClass = Classifier &
self.ownedElement -> not exit (unsafe behaviors)
```

Safe PEARL

```
<module> ::= MODULE [( <name> )]
  [<mod-extension>; [<interface-spec>]*
  [<system-part>] [<problem-part>] MODEND ;

<mod-extension> ::= EXTEND (<name>) [ (<name>)+]
<interface-spec> ::= INTERFACE (<name>) <interface-decl>*
<interface-decl> ::= <interface-var-decl> | <interface-proc-decl>
<interface-var-decl> ::= SPC <name> : <type> READ;
<interface-proc-decl> ::= SPC <name> : PROC <lst-of-par> <return-type>

<system-part> ::= SYSTEM; <name> : <name>;

<problem-part> ::= PROBLEM; [<declarations>]+
<declarations> ::= <var-decl> | <proc-decl>
<var-decl> ::= DCL <name> : <type>;
<proc-decl> ::= <name> : PROC <lst-of-par> <return-type> <body> EN
<lst-of-par> ::= (<par-decl>)* [<par-decl>]*
<par-decl> ::= <name> : type
<body> ::= [<var-decl>]+ [<statement-seq>]
<statement-seq> ::= [<statements>]*
<statement> ::= (<statement-seq>) <assignment-statement>
  | <conditional-statement> | <while-statement>
<assignment-statement> ::= <variable> := <expression>
<conditional-statement> ::= IF <bool-expression>
  THEN <statement-seq>
  [ELSE <statement-seq>] FIN
<while-statement> ::= WHILE <bool-expression>
  REPEAT <statement-seq> END
```

Table PEARL Module for SIL 4:

```
Context PearlModule inv:
self.baseClass = Classifier &
self.ownedElement -> forAll ( E1
  If (boolean -expression) Then expression1 Else expression2)
```

Verifiable PEARL Module for SIL 3:

```
Context PearlModule inv:
self.baseClass = Classifier &
self.ownedElement -> forAll( E1 E = procedure &
  (exist (p: Parameter |
    p.name = name, p.type = type, p.value = value)
    -> size >= 1) & self.hasNoInterfaces() )
```

Safe PEARL Module for SIL 2:

```
Context PearlModule inv:
self.baseClass = Classifier &
self.ownedElement -> forAll( E1 E = procedure &
  (exist (p: Parameter |
    p.name = name, p.type = type, p.value = value) -> size >= 1) &
  self.Interfaces.isKindOf( PearlInterface))
```

Figure 1: PEARL subsets and corresponding stereotypes

The main element dealing with safety issues in the above-mentioned subsets of PEARL is the module. Used to describe software architectures, modules are, according to the

different requirements of the SILs, constructed with rules, procedures, interfaces, various variables and conditions. A PEARL program is structured by grouping modules into collections. The collections are either statically distributed or dynamically allocated to system nodes; they form the elements for dynamic reconfiguration. Collections communicate by point-to-point message exchange on the basis of the port concept, only. Messages are sent to ports, or received from ports, which are known only locally in their own collections. Thus, ports form the input and output interfaces of collections.

The module of the PEARL subsets is closely matched by the Part concept of UML 2.0. Corresponding to each SIL we create a stereotype named `PearlModule`, using `Classifier` as its base class. Constraints allow to specify claims about architectures, their components, and their expected execution behaviour. Formulated as OCL expressions, these constraints can be pre- or post-conditions, but also invariants. Thus, we can also describe the rules of Table PEARL with OCL expressions. The parameter passing of Verifiable PEARL is mapped to the UML feature tagged values. To match Safe PEARL's and HI-PEARL's interface services, we employ both the offered and the required interfaces as known in UML 2.0. The corresponding stereotypes are defined in Fig. 1.

4 Incorporating Function Blocks into UML

Defined within the standard IEC 61131-3, Function Block Diagrams (FBD) are signal-flow and design-object oriented, and used in many safety-related control applications up to SIL 3. The basic language elements in FBD are instances of functions and function blocks which, for the purpose of program composition, are interconnected with connection lines between their inputs and outputs, respectively. Functions and function blocks represent high-level application-oriented and re-usable language elements. Before being released, all functions and function blocks contained in a specific library are verified employing appropriate, usually formal methods. Safety-licensing needs to be carried out only once for a certain application area, after a suitable set of function blocks has been identified.

The definition of a function block (FB) is simply a type declaration, like a class is the definition of an object type. A program can contain several instances of the same FB, each one independent of the others as their internal data are allocated in separate memory areas. Moreover, the same FB can be used in several different programs, reducing the effort for code re-writing. This means that libraries of user-defined reliable software components can be compiled of field-proven functions blocks. An instance of a function block consists of input, output and in-output variables as well as algorithms. Function blocks communicate with each other by assignments of data values to interface variables, only. A characteristic feature of FBs is the separation of external interfaces and internal implementation.

To make the concept of function blocks according to IEC 61131-3 available in UML, we use the class stereotyped Component architecture of UML 2.0 [13] to define FB stereotypes. Hereby, we can model the internal structure of FBs with Parts and Connectors, and model the variable-interfaces of FBs with Ports and Interfaces. Data flow can be specified by connecting in-ports and out-ports. Out-ports contain the result of a computation

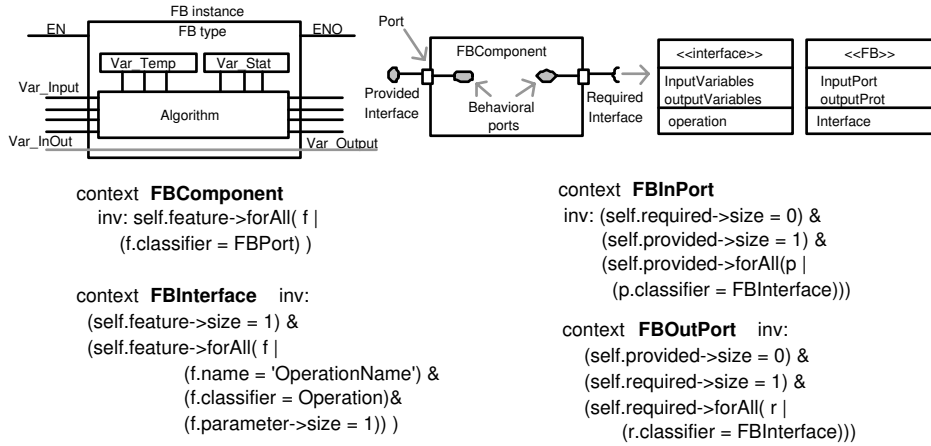


Figure 2: Stereotypes of Function Block

based on the input and the current state of the function block. An FB is defined as a UML component stereotype as depicted in Fig. 2. The ports in the UML component are direct counterparts to the interface variables of FBs. The stereotypes **FBInPort** and **FBOutPort** are inheriting from **FBPort**, **FBOutPort** requires one **FBInterface** and **FBInPort** provides one **FBInterface**. The stereotype **FBInterface** is needed to model input and output variables of FBs; it may contain operations. Using OCL, the stereotypes could be expressed in UML notations as shown in Fig. 2.

5 Related work

Constructing programming language subsets for highly safety-critical applications has been proposed for many years. Examples for this are SPARK [1, 14], a safe subset of the programming language Ada, or subsets of Java and Pascal, and even C can be used in some safety-related systems [2]. The subsets are usually enforced by coding standards, and have been chosen to exclude certain weaknesses of their respective base languages. As mentioned above, of PEARL four nested subsets have been defined to enhance the safety of real-time control software.

Earlier, we have explored to extend UML with PEARL-oriented constructs for modeling real-time systems [11]. In this paper, we incorporate subsets of programming concepts oriented at the PEARL subsets and the function block concept into UML for modeling safety-related systems as well. The integration of function blocks into UML with a Function Block Adapter has first been introduced in [6]; it was oriented at IEC 61131-3. Then, the Function Block Adapter has further been extended to adapt also function block types as defined in other languages [7]. Here, we define a function block stereotype as a safe subset to be integrated into UML for safety-related application.

6 Conclusion

With the help of a profile, UML can be adapted to application areas for which standard UML is not specific enough. In this paper, a set of UML stereotypes is introduced providing the functionality of safe subsets needed to design safety architectures. These safety-related stereotypes can be assigned to UML elements. Oriented at the safety facilities found in the prevailing safety standards, the stereotypes can be incorporated into models of safety-related embedded real-time control systems. Their advantage is to allow for high-level descriptions during early phases of the development process. This work will be carried further to build a complete UML safety profile matching the requirements of the four Safety Integrity Levels.

References

- [1] B.A. Carré and T.J. Jennings: SPARK-The SPADE Ada Kernel Programming. Department of Electronics and Computer Science, University of Southampton, 1988.
- [2] W.J. Cullyer, S.J. Goodenough and B.A. Wichmann: The Choice of Programming Languages for use in Safety-Critical Systems. *Software Engineering Journal*. March 1991.
- [3] DIN 66253, Teil 3: *Mehrrchner-PEARL*. Berlin-Köln: Beuth Verlag 1989.
- [4] DIN 66253-2: *Programmiersprache PEARL90*. Berlin-Köln: Beuth Verlag 1998.
- [5] W.A. Halang and A.H. Frigeri: Methods and Languages for Safety Related Real Time Programming. *Computer Safety, Reliability and Security*, W.D. Ehrenberger (Ed.), pp 196 – 208, LNCS 1516, Berlin-Heidelberg-New York: Springer-Verlag 1998.
- [6] T. Heverhagen and R. Tracht: Integrating UML-RealTime and IEC 61131-3 with Function Block Adapters, Proc. *ISORC 2001*, IEEE Computer Society, 395–402, 2001.
- [7] T. Heverhagen and R. Tracht: A Profile for Integrating Function Blocks into the Unified Modeling Language. Proc. *Specification and Validation of UML Models for Real Time Embedded Systems*, SVERTS 2003.
- [8] W.A. Halang and J. Zalewski: Programming Languages for Use in Safety-Related Applications. *Annual Reviews in Control* 27, 1, 39 – 45, 2003.
- [9] International Standard IEC 61131-3: *Programmable Controllers, Part 3: Programming Languages*. Geneva: International Electrotechnical Commission 1992.
- [10] International Standard IEC 61508: *Functional Safety of Electrical/Electronic/ Programmable Electronic Safety-related Systems*. Geneva: International Electrotechnical Commission 1998.
- [11] S. Lu: An PEARL orientierte Spezifikation verteilter eingebetteter Systeme mit UML-Stereotypen, Proc. *Workshop PEARL 2003*, Berlin-Heidelberg: Springer-Verlag 2003.
- [12] Object Management Group: *Unified Modeling Language: Specification*. V1.4, 2001.
- [13] Object Management Group: *Unified Modeling Language: Superstructure*. OMG document ptc/2003-08-02, 2003.
- [14] I.C. Pyle: *Developing Safety System – A Guide Using Ada*. Prentice Hall, 1991.