Acta Polytechnica CTU Proceedings **11**:1–5, 2017
doi:10.14311/APP.2017.11.0001

available online at http://ojs.cvut.cz/ojs/index.php/app

COMPARISON OF LANGUAGE SUBSET AND LANGUAGE EXTENSION BY SAFE RELATED INFORMATION APPROACH TO SAFE SYSTEM DEVELOPMENT

Tomáš Brandejský*, Vít Fábera

Department of Applied Informatics in Transportation, Faculty of Transportation Sciences, CTU in Prague
* corresponding author: brandejsky@fd.cvut.cz

Abstract. Presented contribution is dedicated to discussion of two different approaches into increase of programming language safety. They are language subset and extension of original safety mechanisms. As examples we used MISRA C/C++ subset and SPARK language builded on the base of ADA language. In the last chapters we discuss novel approaches based on application of programmable hardware which is described in VHDL language, which is also modification of ADA language. Especially SPARK and VHDL languages represents novel approaches to safe system development which are now discussed in relation to new Railway SW projects.

Keywords: MISRA, ADA, SPARK, VHDL, programmable hardware, software safety.

1. Introduction

Nowadays we can observe two opposite approaches to modification of programming languages to be more suitable to safe system development. The first is the reduction of chosen programming language expressions to safe subset, like MISRA C/C++ [1, 2] and this approach is also supported by today standards of safe software development like ISO61508 [3] or EN50128 [4]. In the last few years, novel and partially opposite development is expanded. It is based on extension of original language by additional explanations containing description of such aspects of software, that are not described enough by original language. Typical example is SPARK [5] language precisiating original ADA one [6] even though it was originally designed as language specialized to safe control system development.

2. MiSRA C/C++

MISRA C (Motor Industry Software Reliability Association) C language subset was developed in 1998, latter it was updated and also there was developed analogical support for C++ (MISRA C++). MISRA C/C++ defines itself as guidelines. It defines language subset in the form of rules partially suggesting good practice solutions, frequently warning programmers away of wrong application of the language. Especially valuable of MISRA guideline are parts explaining why was each rule formed, explaining what is the subject of possible future problem. Applying of this rules allows to write robust code with decreased probability of malfunction.

Now it is possible to apply compilers and source code checkers implementing MISRA guidelines to allow automatic static analysis of code. Even, there exists free software tool clang-misracpp2008 [7] but the development of this tool probably stopped before two years. From the practical viewpoint, MISRA guidelines checker implemented in compiler is also very interesting. It is even better than standalone checker because checking in compiler runs in each code making and does not require additional checking operation step. Many companies producing compilers for development of embedded systems based especially on single chip micro-controllers, for example Keil software company which is widely used by railway control system developers community in our country.

There exists effort to implement MISRA rules into GNU C/C++ compilers because these compilers are free, proven in use and supports many processors and micro-controllers. One of the first works in this areas was [8], where authors describe its checker for GNU C which is written in Ciao Prolog language into which MISRA rules they were transformed. Ciao Prolog is similar to GNU Prolog and is also free and compatible with GNU one but it seems to be more advanced than GNU Prolog in its design. This project probably is not active now.

There was also European project within the frame of ITEA3 call for development of GCC fork called GGCC [9] with aim to implement MISRA rule checking into GCC compiler. This project probably newer started or produced no reasonable result. Thus it is possible to conclude that now there is no applicable support of MISRA for GNU C compilers and that GNU compilers development effort now focuses rather to support of parallel (super-)computers, like implementation of ACC extension of standard C/C++ languages.

MISRA C specification is not public domain and this fact might also cause small interest of GNU community to implement this subset.

There exist also similar subset called High Integrity C++ [10] by Programming Research, but also this specification is not public domain as well as checking tools.

3. ADA language

Origin of ADA language falls into falls into 80's of previous century. This language was developed for US Army applications as safe, strong typing programming language supporting parallelism, inter-process safe communication and defensive programming style preventing programmers errors and decreasing the need of testing. This Pascal (or Algol) like language was adopted in avionics and nuclear industry and it is referred by safe software system development standards [3, 4].

Because the language was designed with respect to extremely large scale control system design operated by US Ministry of Defence, it was designed to eliminate maximum of programmers errors, ability to detect large percentage of them during compilation of code and with ability to operate in distributed environment with respect to possible communication errors and system component faults.

The language was equipped by specific typing mechanism supporting defensive programming to support ability of error detection by compiler. Defensive programming is related to strong typing, which means, that magnitudes between variables of different types cannot be shared without explicit conversion. Because mainstream programming languages like C offer limited list of types (integer, real, character), the use of type control is able to discover only limited number of programmer errors when such languages are used. On the opposite side, ADA language offers ways how to define new data types e.g. by constraining of definition domain of sample type (e.g. `type secondfloornumbers is range 200..253;` defines data type which magnitudes are constrained to really existing scale of numbers and allows to compiler to identify each impossible magnitude caused by programmer or communication error, e.g. 0). Because strong typing might be exhausting, when it is required in situations when it brings to benefit, in ADA it is possible to define subtypes, which magnitudes are transformed automatically without explicit type conversions between original type and all its subtypes.

All data structures use closing clause unambiguously defining its end to prevent mistakes caused by wrong understanding of nested structures in the time of development, testing and maintenance. On the end of function and procedure definition it is supported its name stating.

For many years, in mainstream programming languages the support of concurrent programming was extraordinary. ADA language contains it from its origin in 70's. ADA allows not only define processes, but it contains strong support of safe inter-process communication and process synchronization called rendezvous. Each process has defined so called entries to accept messages sent by other processes and is able to send messages to entries of other ones. But not only. It is possible to choose, which messages will be accepted in given state of process and it is possible to define reactions to unexpected situations, especially to fact, that given message was not received for given time. Each process might define its local variables and separate itself from others.

Due to its real time orientation, ADA language supports exception handling to solve unexpected situation without program fail. There is present concept of modular programming which applied templates (modules without precisely defined types, operators, procedures and functions which might be precisiated later in the time of final program linking. It allows e.g. to write sorting algorithm library without knowledge of sorted data type. Such library si then applicable to any data type implementing assignment and comparison operators.

Latter ADA adopted object oriented programming concept but it is not widely used as e.g. C/C++, Java and some other languages, especially due to high price of commercial compilers and need of specific education of programmers. Similarly like for C/C++, also for ADA language there was effort to develop safe subset to increase its (high) safety. Also standards [3, 4] refers this subset but in reality nothing such exists. The improving of ADA language was moving different way and tends to SPARK language, especially due to impossibility do make such simple programmers errors as they are solved by MISRA for C/C++. Significant part of such errors can not occur in standard ADA and thus its improvement is much difficult, because even original (full scale) ADA operates on different (much higher) level of safety than e.g. C.

There exist support of ADA language based on GNU compiler collection, but it means that there are supported especially versatile operating systems like Linux and common purpose processors. There is also interesting, that compiler is not named ADA compiler, but by acronym GNAT. It is caused by licensing, when ADA compiler term can be used only for compiler satisfying large (and thus expensive) verification, which is out of scope of GNU community.

SPARK language is not pure subset. It also extends ADA code by additional information to eliminate potential ambiguities and insecurities hidden in ADA source codes. The main changes to standard ADA language are the following:

- The use of access types and allocators is not permitted (term access in ADA denotes pointer – more frequent term known from the other programming languages, its misuse might cause significant errors hard to discover by testing).
- All expressions (including function calls) are free of side-effects. This improvement was reached by incorporating of additional description of use of variables and symbols visible from neighborhood to prevent e.g. mistaking of global and local variable of the same name.
- Aliasing of names is not permitted in general but the renaming of entities is permitted as there is a static

relationship between the two names. In analysis all names introduced by a renaming declaration are replaced by the name of the renamed entity. This replacement is applied recursively when there are multiple renames of an entity.

- The goto statement is not permitted.
- The use of controlled types is not currently permitted.
- Tasks and protected objects are permitted only if the Ravenscar profile (or the Extended Ravenscar profile) is specified. Ravenscar profile denotes set of pragmas influencing compiler work to prevent some possibly unsafe construction in ADA code. Ravenscar profile is available in standard ADA too, in SPARK is is not option but he standard.
- Raising and handling of exceptions is not currently permitted (exceptions can be included in a program but proof must be used to show that they cannot be raised).

MISRA C/C++ especially eliminates confusing structures which might be understand differently by compiler, programmer and structures which are differently understand by different compilers, especially due to vague language definition (it is especially problem of C language, C++ in its last version was significantly improved, precisiated). For example if there is used condition ((x==A)&&(y==B)), there is not defined in language specification if the sub-expression (x==A) is evaluated as the first or the second. It is no problem until we do modification of this condition to form ((x()==A)&&(y()==B)). In this new condition x and y are not static objects (variables), but they are function calls and functions might contain so-called side effects, in this case they might operate the same global variable and thus the results of calls of functions x() and y() might be influenced by call order.

This subset has also many rules eliminating confusing structures causing misunderstandings by programmers and testers. MISRA subsets contain tens such confusing structures. E.g. it is possible to mention constants starting with zero (they are understood by compilers as octal numbers by definition), problem of nested structures if-else which might not contain else branch and no terminal symbol stopping this structure is defined in the language and so on.

4. SPARK

SPARK language introduces explicit description of use of global objects (and commonly objects defined on higher level of visibility) called dependencies. The use of these additional descriptions is known from design by contract methodology, which was on the origin of such languages as Eiffel [11]. This methodology prescribes that software designers should define formal, precise and verifiable interface specifications for software components, which extend the ordinary definition of abstract data types with preconditions, postconditions and invariants. Preconditions are conditions which must be satisfied to enter the following section. Opposite postcondition must be satisfied on the end of the block and the invariant must be valid inside the block.

5. VHDL

VHDL (VHSIC Hardware Description Language) [12] is now frequently used in programming of field-programmable gate arrays. It describes structure and/or behaviour of digital hardware. VHDL is similar to ADA language both in concept and syntax (parallel processes, subtypes, ...). It was primarily proposed for simulation, now the VHDL description is an input for synthesis tool instead of schematic diagram to produce final design. The interesting question is, if the similarity of VHDL and ADA languages cannot be used to description of both SW a FPGA implementation of systems with high requirements to Reliability, Applicability, Maintenance and Safety. Simulation of system described by VHDL language is not straightforward due to need to respect signal and not process oriented nature of FPGA. This nature caused presence of sensitivity list in VHDL code. In VHDL, the proces statement has an part called *sensitivity list*. The syntax is as follows:

```
[process_label:]process[(sensitivity_list)]
declarative_part
begin
statement_part
end process[process_label];
```

Example:

```
ANDGate1: process (A,B)
begin
Y <= A and B after 5 ns;
end process;
```

The sensitivity list consists of input signal identifiers separated by commas. Sensitivity list is important at the moment of the simulation. During the initialization phase of simulation, the process runs and it assigned initial values of signals and variables (signal Y in our case). Then the process is suspended. If the any signal listed in the sensitivity list is changed, the process is resumed and executed. The sensivity list after the process keyword is syntactically optional, but we can really say that is mandatory. The sensitivity list placed after the process keyword can moved at the end ot the process section and written like the *wait on* statement:

```
ANDGate1: process
begin
Y <= A and B after 5 ns;
wait on A, B;
end process;
```

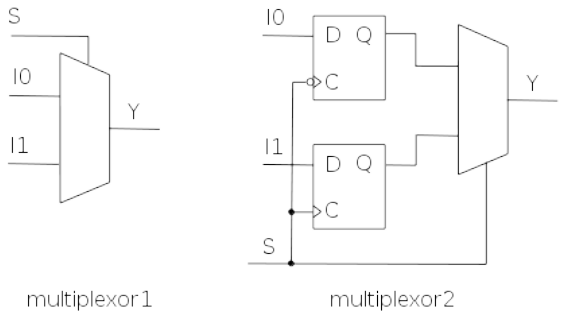

Figure 1. Corresponding schematic diagrams to VHDL code related to simulation.

The process having sensitivity list after process keyword must contain **no** *wait on* statement. If some signal is missing in the list, it affects naturaly the simulation. For example, signal B is missing:

```
ANDGate1: process (A)
begin
Y <= A and B after 5 ns;
end process;
```

Let consider the set of simulus: both signals are set to 0 at time 0ns, A signal is changed to 1 at 10ns, B signal is changed to 1 at 20 ns. The result of simulation in Isim (a part of XILINX ISE Design Pack) is constant zero signal Y, because the process in not activated at the time 20 ns due to missing signal B in the sensitivity list. While process ANDGate1 really describes combinational AND, ANDGate2 process has sequential behavior.

We show one more example of code and corresponding schematic diagrams to simulation (Fig. 1).

```
multiplexor1: process (S, I0, I1)
begin
if S = '0' then Y <= I0;
else Y <= I1;
end if;
end process;

multiplexor2: process (S)
begin
if S = '0' then Y <= I0;
else Y <= I1;
end if;
end process;
```

In simulation, the output Y of multiplexor2 process changes only if event on S occurs. It means that output Y follows the value of I0 at the moment of falling edge on S and value of I1 at the moment of rising edge on S.

6. Synthesis

Processes ANDGate2 and multiplexor2 have sequential behaviour during the simulation with ISim. How process these codes synthesis tools XILINX ISE 12.4? If a signal is missing in the sensitivity list, tools writes a warning like *"I0 should be on the sensitivity list of the process"*. If signal Y is listed in the sensivity and it Y an output of the entity, the VHDL compiler writes an error *"y with mode 'out' cannot be read"*. If Y is an internal signal, it can state in the sensitivity list without warning or error. But all variants of code lead to the same RTL (Register Transfer Level) – combinational logic – AND Gate and multiplexor. Syntesis tool decides if it uses combinational logic or flip-flop according to typical VHDL code structures (`if clk'event and clk = '1' then` for flip-flops).

So, if we forget to put a signal in the sensitivity list and if we ignore warnings, the simulation of the code and the behavioral of the target design can be different. The reason is: to put all input signal in the sensitivity list and or use a variant of the sensitivity list with keyword `all`, which was added into new specification of the language. The gcc compiler has `-wall` switch, that cause to stop compiling also if a warning occurs. Switch -wall is recomended in C language if we create a safety software. Synthesis tools (XILINX ISE) have usually no switch like `-wall`, Vivaldo Design Suite from XILINX makes possible to change severity from warning to errors.

As seen, VHDL, although declared as language with safety properties, has also some weak points. A question is: Wouldn't it be suitable to extend VDHL compilers in the sense of -wall switch and to add a variable list like in SPARC, espetially, if VHDL is speculated as a language for railway systems? By the way, rules for safety design using VHDL was presented in [13].

7. Discussion

VHDL language uses control structures and programming methodology analogical to original ADA language. It is the subject of future research, if it is possible to modify it in the way analogical to SPARK to increase its safety and analysis if such extension could bring significant increase to safety due to influence of hardware (gate array) to it.

8. Conclusion

The presented paper brings analysis of impact of two basic methods to increase language safety to cover safety requirements of systems working on high safety integrity levels, e.g. on railways. It is not only focused to comparison of ADA and SPARK languages but it is illustrated on problems bring by implementation of VHDL simulator designed with respect to describe safe system by single language without need to distinguish if this system will be implement by HW or SW.

References

[1] Guidelines for the use of the c language in critical systems, March 2013. ISBN 978-1-906400-10-1 (paperback), ISBN 978-1-906400-11-8 (PDF).

[2] Guidelines for the use of the c++ language in critical systems, June 2008. ISBN 978-906400-03-3 (paperback), ISBN 978-906400-04-0 (PDF).

[3] Iec 61508 – electronic functional safety package.

[4] En50128: Railway applications – communications, signaling and processing systems – software for railway control and protection systems, 2011.

[5] Iso/iec 8652:2012(e). ada reference manual. Copyright ©1992, 1993, 1994, 1995.

[6] Spark 2014 reference manual. Copyright ©2013-2016.

[7] "https://github.com/rettichschnidi/clang-misracpp2008".

[8] G. Marpons. A coding rule conformance checker integrated into gcc [online]. "babel.ls.fi.upm.es/ gmarpons/pubs/PROLE08Codingrules.pdf".

[9] Ggcc reseny mandrivou v ramci itea3.

[10] "http://www.codingstandard.com/section/index/".

[11] Eiffel language [online]. "http://www.ecma-international.org/publications/standards/Ecma-367.htm".

[12] Ieee standard vhdl language reference manual, 2000.

[13] T.Musil. *Návrh metodiky pro vývoj a verifikaci bezpečných algoritmů implementovaných v dynamicky rekonfigurovatelných FPGA (Methodology for design and verification of algorithms with demands for safety on dynamically reconfigurable FPGA), doctoral thesis.* Czech Technical University, 2015.