# SHIELDS UP! DEFENDING SOFTWARE AGAINST SAFETY & SECURITY RELATED PROGRAMMING ERRORS

**Dr Darren Buttle**
ETAS GmbH

## ABSTRACT

Software in the modern car is astonishingly complex, comprising thousands of functions and millions of lines of code. Engineering that software to ensure that drivers and passengers remain safe is a herculean task. The key to managing complexity is abstraction – hiding unimportant details so that the problem at hand can be seen clearly, solved effectively and implemented efficiently. Unfortunately, the tool we use most often is the C language that does the opposite – exposing details so that software intent is obfuscated by workarounds for language deficiencies making development and maintenance unnecessarily hard.

In this paper we present ESDL, the language of the ETAS ASCET model-based software development tool as a better way to get to C. ESDL is a domain specific language that provides useful abstractions to make software clearer, faster to write, and easier to get right then uses automatic code generation for efficient implementation.

## INTRODUCTION

Everyone is aware of the challenges of building automotive software: there is an ever increasing demand to add new functions, those functions themselves are increasingly complex and time to market demands create huge pressure to increase development efficiency. Today's advances in vehicle technology are accelerating this trend. Advanced driver assistance systems (ADAS) are now sufficiently capable that the move to fully autonomous vehicles looks likely within a few vehicle generations, focusing more attention on the need to engineer *safe* software. And the recognition that vehicles, like every other device enabled by software, must support in field over-the-air updates significantly increases the need to engineer *secure* software

The average modern high-end car contains about 100 million lines of code [DoughtyWhite15]. But the way in which that software is engineered hasn't changed fundamentally since there were merely a few thousand lines of code. There are a few places where code generation has become standard, notably in powertrain development and for ECU's running AUTOSAR where the abstraction of the virtual function bus relies on code generation to create an RTE for optimized communication and scheduling infrastructure. However, in most areas software creation remains dominated by hand written C.

## PERILS OF C

C is the de facto programming language for embedded software for good reasons. C is simple, small, fast and portable. However, anyone who has developed software in C will be aware of its darker side: C's low level of abstraction often does not reflect the problem domain, and its encouragement to "program on the edge of safety" makes it astonishingly easy for errors to creep into programs and agonizingly difficult to find them. For many embedded micro-controllers however, C is often the only high level language choice available.

Safety standards like ISO26262 [ISO11] & IEC61508 [IEC10] and security standards like the Common Criteria [CC09] place stringent requirements on language selection and use. While software that satisfies these standards can be written in C, its use needs the adoption (and policing in the process) of language subsets.

"Safer C" subsets defined by guidelines such as those provided by MISRA [MIRA14], ISO [ISO12], CERT-C [Seacord14] help to avoid the worst abuses of C. They embody best practices and avoid language features that have proven difficult to use or understand, but don't help to remove problems that can creep into the code base like the ubiquitous "buffer overflow", numeric under/overflow, division by zero etc.

Preventing these classes of errors means that a second line of defense is needed, using either defensive coding to prevent the problems or static analysis to demonstrate that the problems do not occur. Defensive coding is hard to deploy manually for deeply embedded software. The additional code and runtime inherent in defensive coding means that it cannot be simply applied everywhere it is needed. Working out where checks can be safely removed, and maintaining this over multiple updates of the software, is error prone. Using static analysis offers the promise of zero overhead, correct

code, but practical application is non-trivial. The ambiguity inherent in the C language makes application of such techniques prone to false positives and this can have a detrimental impact on tool acceptance [Bessey10].

Using these common lines of defense does not however, guard against semantic problems in the code, for example changing into a gear that does not exist in a gearbox, or accidentally assigning a pressure to a temperature. C does not have enough "expressive power" to capture these important program properties in a short and precise way. C's weak type system, limited support for encapsulation (and thus systematic reuse) often results in users building mechanisms to emulate these features [Ernst02]. The result is that the real intent of the program becomes obfuscated, making it difficult to review and maintain, meaning engineers spend valuable time working around C's limitations that could be spent more productively building applications.

## CODE GENERATION: A SAFER WAY TO C

What has proven successful in some domains, notably control systems, is "model driven development" where the abstraction is a graphical notation for describing data and/or control flow from which C code can be generated [Simulink, ASCET]. Automatic code generation to C has been shown to significantly reduce residual program anomalies (i.e. potential sources of bugs) in comparison to hand-written code [German08].

Programming tasks however, are not inherently suited to graphical description (this is why all programming languages are text based), but there is no reason why better abstraction cannot be provided in a textual notation.

Our approach was to develop a C-like textual modeling language that:

- increases the likelihood of writing a program that behaves as intended by allowing intent to be expressed with less noise than C;
- was amenable to "on-the-fly" static checking for errors during editing time;
- captured enough information to allow code generation to add defensive coding checks where required and, crucially in the deeply embedded domain, to elide those checks when they are not required; and
- can be automatically translated into MISRA compliant C code suitable for execution in very resource constrained, real-time environments. Typical domains of application are automotive electronics, process control, intelligent buildings etc.

The resulting language is Embedded Software Development Language (ESDL), and the associated tooling and code generation technology embedded in the ETAS ASCET product.

## A TOUR OF ESDL

ESDL has a look and feel that is familiar to existing C programmers: sharing a similar syntactic style, statement and expression language. Like C subsets, ESDL removes dangerous programming constructs but crucially makes this part of the language itself. This means that checks can be done on-the-fly in the IDE without need for auxiliary tooling.

The following sections highlight some key aspects of ESDL, and the classes of programming errors they help avoid.

SAFER SYNTAX

The syntax of ESDL avoids the well know and well documented "dangerous" constructs of C including:

- No optional braces for blocks: preventing statements accidentally being moved out of conditions (Apple's SSL/TLS bug is a recent example where an additional `goto` was not protected and a critical cryptography check was never executed);
- No global variables: eliminating problems of data consistency in multi-threaded applications.
- No use of statements as expressions: eliminating classic C errors like `if (a = 0) {...}`;
- No assignment to loop guards: removing the potential for accidentally creating an infinite loop;
- No pointers: to secure programs against arbitrary access to, and manipulation of, memory. ESDL programs can only declare and use static data;
- No labels & `goto`: ESDL has well-defined control flow that prevents programs that jumping to arbitrary locations.
- No unions: to simplify the ability to type-check ESDL programs
- No fall-through of case branches in switch statements (case expressions in ESDL can use ranges of values, eliminating the need for a fall through)

Unlike subsets, all of these features are integral to ESDL and checking is tightly integrated in the development environment.

SAFER NUMERIC TYPES

ESDL moves away from C's non-portable storage-centric primitive types (`int`, `char`, `short` etc.) to encourage developers to think about values and what they mean. The primitive numeric types of ESDL are `integer` and `real`. There is no implicit type conversion (except for over assignment).

New (sub-)types can be created that inherit the properties of their parent types, but constrain the range of the type to valid values. Ranges formalize assumptions about what values are expected. For example:

```
3  type Age is integer 0 .. 200;
4  // Enough for Jeanne Calment, the world's oldest person
5
6  type Temperature is real -273.15 .. 10000.0;
7  // Temperature cannot be less than absolute zero
```

Types with value ranges are a simple concept but provide powerful protection against semantic errors, offering a number of development benefits:

- More effective review: implicit assumptions about permitted values are explicit in the program text and thus easily visible for review or inspection processes.
- Easier maintenance: changes to a range in a single location are inherited by all elements of the type across the whole program.
- Automatic optimal memory size: the C type needed to store values of the ESDL type can be derived from its range. This ensures optimal memory allocation. Furthermore, there is no risk of introducing errors when the range needs to increase to hold more values, but a variable is not correctly resized.
- Automatic C casting: the complexity of safely using C casting to ensure correct calculation results are handled through code generation (following the guidance of MISRA-C:2004 Rule 10.x).
- Automatic generation of defensive code: declarative specification of the range enables code generation to automatically add defensive code to guarantee that variables never underflow or overflow.

The following example shows a simple ESDL program that uses a bounded integer called `My_Int`:

```
10 type My_Int is integer -123 .. 456;
11
12 class C
13 {
14     My_Int add;
15     My_Int sub;
16     My_Int mul;
17     My_Int div;
18
19     public void Calculate(My_Int in X, My_Int in Y)
20     {
21         add = X + Y;
22         sub = X - Y;
23         mul = X * Y;
24         div = X / Y;
25     }
26 }
```

The corresponding C code to ensure identical functionality without range violations, division by zero etc. is as follows:

```
226 void Pkg_C_TypeImplementation_Calculate (
227                 const struct Pkg_C_TypeImplementation * self,
228    /* IN    */ sint16                                  X,
229    /* IN    */ sint16                                  Y
230    )
231 {
232    /* temp. variables */
233    sint16 _t1sint16;
234    sint32 _t1sint32;
235
236    /* Calculate: line #1  */
237    _t1sint16 = X + Y;
238    add_VAL
239       = ((_t1sint16 >= -123) ? (((_t1sint16 <= 456) ? _t1sint16 : 456)) : -123);
240
241    /* Calculate: line #2  */
242    _t1sint16 = X - Y;
243    sub_VAL
244       = ((_t1sint16 >= -123) ? (((_t1sint16 <= 456) ? _t1sint16 : 456)) : -123);
245
246    /* Calculate: line #3  */
247    _t1sint32 = (sint32)X * Y;
248    mul_VAL
249       = (sint16)(((_t1sint32 >= -123) ? (((_t1sint32 <= 456) ? _t1sint32 : 456)) : -123));
250
251    /* Calculate: line #4  */
252    _t1sint16 = ((Y == 0) ? X : (X / Y));
253    div_VAL = ((_t1sint16 > -123) ? _t1sint16 : -123);
254 }
```

ESDL provides fixed-point types, making it easy to use the absolute calculation precision they offer (essential for safety related applications where the risk of floating-point calculation anomalies cannot be tolerated). C code is generated to manage the re-scaling, shifting and expression re-ordering automatically [Honekamp99]. The ESDL code more clearly expresses intent, without being polluted by underling implementation details.  For example, consider this ESDL fixed-point code:

```
1  package Pkg;
2
3  type A_Type is real -50.0 .. 50.0 delta 0.1;
4  type B_Type is real 10.0 .. 20.0  delta 0.4;
5  type C_Type is real -7.0 .. 49.6  delta 0.3;
6
7  static class D
8  {
9      public C_Type Calculate(A_Type in A, B_Type in B)
10     {
11         return (A*B);
12     }
13
14 }
```

Even for a relatively trivial calculation, the corresponding C implementation (considering C casting and trying to minimize precision loss) is complex:

```
228 sint16 Pkg_D_TypeImplementation_Calculate (
229    /* IN    */ sint16 A,
230    /* IN    */ uint8  B
231    )
232 {
233    /* temp. variables */
234    sint16 _t1sint16;
235    sint32 _t1sint32;
236
237    /* Calculate: line #1  */
238    _t1sint16 = A * (sint16)B;
239    _t1sint32 = (sint32)_t1sint16 * 2;
240    /* return with expr from Calculate: min=-23, max=165, hex=10/3phys+0, limit=(maxBitLength: true, assign: true), zero incl.=true  */
241    return (sint16)((((_t1sint32 >= -350) ? (((_t1sint32 <= 2480) ? _t1sint32 : 2480)) : -350)) / 15);
242 }
```

This type of code is difficult to maintain by hand and is best handled by automatic generation using tools like ETAS ASCET.

In addition the improved numeric type model of ESDL, generated C code is free from the following C runtime errors:

- Division by zero. ESDL defines division by zero to be the numerator of the division expression.
- Signed overflow of C base types (division of the minimum signed integer by -1). ESDL defines this to be the maximum value representable by the underlying C storage type.
- Under/Overflow of machine storage type. Potential violations are handled by “saturating” to the upper and lower limits of the machine type to provide fault tolerance.

SAFER ENUMERATION TYPES

Enumerations are proper types (like they are in C++ and Java) and are declared as follows:

```
22 type Animal is {Dog,Cat,Fish};
23 type Color is {Red,Green,Blue};
24 type Fruit is {Apple,Pear,Mango};
```

ESDL ensures proper use of the abstraction each type creates and it is not possible to use enumeration literals as numerical values, or combine values from different enumerations. This stops programs combining enumerations incorrectly as illustrated by the following example:

```
44 Fruit My_Fruit;
45 My_Fruit = (Fruit.Apple + Color.Blue)/Animal.Fish;
```

Access to the value of the enumeration literal must be explicitly programmed, using the intrinsic `getValue()` function, another case of explicit type conversion to make the program clearer to the reader.

SAFER SEMANTICS

ESDL provides native support for units of measurement and conversions between them, using code generation to perform unit conversions (e.g. miles to kilometers). ESDL also provides semantic checking of program correctness in expressions, for example that a distance divided by a time is a speed as shown in the following example:

```
11 unit meters;  // Create meters "ex nihilo"
12 unit seconds; // Create seconds "ex nihilo"
13
14 unit centimeters = 0.01 * meters;
15 unit kilometers = 1000.0 * meters;
16 unit minutes = 60.0 * seconds;
17 unit hours = 60.0 * minutes;
18
19 unit kph = kilometers / hours;
```

SAFER DATA STRUCTURES

ESDL removes C's restriction that arrays are indexed from zero. An ESDL array can be indexed by any discrete scalar type, including negative numbers (e.g. to program a set of compensations around zero) and enumerations (e.g. to index by fault code). This means that the code is clearer and better reflects the data being manage and is more robust to change.

```
26 type Color is {Red,Green,Blue};
27 type Saturation is integer 0 .. 255;
28 type Pixel is [Color] of Saturation; // Array of RGB values
```

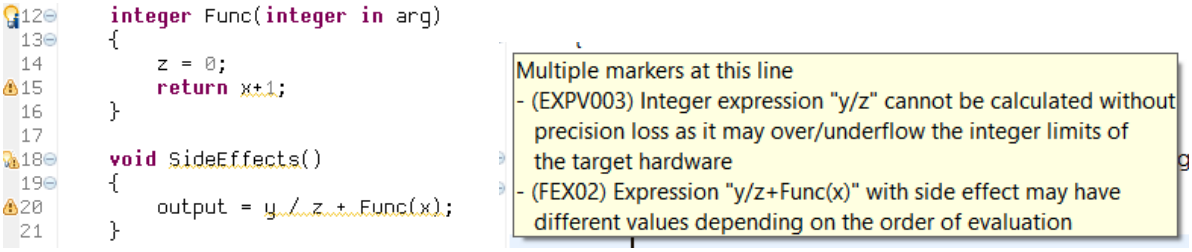
ESDL also eliminates the C problem of indexing outside the bounds of an array by automatically saturating out of bounds array accesses to the first or last array element appropriately. All ESDL arrays are statically sized and cannot be handled as pointers to memory, eliminating a huge class of "buffer overflow" errors commonly seen in software.

SAFER FUNCTIONS

ESDL abstracts how arguments are used in functions. Formal parameters are read-only by default, and can optionally specify read-write and mandatory assignment before read using a simple `in` (read only), `out` (must be assigned) and `in out` (read-write) model. This prevents incorrect use of parameter values inside a method or treating a value as an address (or vice versa).

```
47 void My_Function(integer in     ReadOnly,
48                  real    in out ReadWrite,
49                  boolean    out IsOkay)
50 {
51    boolean local;
52    local = IsOkay;    // Error! Cannot read before initialized
53    ReadOnly = 42;     // Error! Cannot write to read only value
54    ReadWrite += 42.0; // OK
55    IsOkay = false;    // OK
56 }
```

Calling functions can be problematic in C, as parameter aliasing and the presence of side effects makes it hard to establish, before compilation, what a function will do. ESDL removes ambiguity by forbidding parameter aliasing for writable parameters, making it illegal to pass the same `out` argument twice and therefore isolating users from the impact of the compiler vendors design choices in argument passing (and helping portability).
ETAS ASCET tooling also detects side effects. The following example shows a function `Func()` that has side effect of touching state variable z. When the function is used at Line 20, the potential impact of the side effected is detected:




SAFER STRUCTURE

It takes skill and discipline to write systematically re-usable C code. The language has little support for modularity and encapsulation. ESDL is object-based, using classes for encapsulation and information hiding. Class data and code is private by default and must be explicitly made public.

Classes support variant initializers, allowing the creation of objects to be parameterized statically from a set of pre-defined values.

```
1  package Vehicle;
2
3  class My_Car
4  {
5
6      // Internal data is always private
7      integer cylinders = 4;
8      integer seats = 4;
9
10     // Variant initializers enable re-use and
11     // override default data values
12     public initializer Sportscar()
13     {
14         cylinders = 8;
15         seats = 2;
16     }
17
18     public initializer SUV()
19     {
20         seats = 6;
21         // Has 4 cylinders
22     }
23
24     // Methods are private by default…
25     void Private_Method()
26     {
27         // Some code…
28     }
29
30     // …and explicitly made public
31     public void Public_Method()
32     {
33     // Some code…
34
35     }
36 }
```

ESDL classes can be instantiated multiple times and can declare optional variant initializers. Variant initializers support product line engineering directly in the code base, allowing the creation of objects to be parameterized statically from a set of pre-defined values, for example:

```
38 static class My_Garage
39 {
40     My_Car   My_Normal_Car;
41     My_Car   My_Sports_Car = My_Car.Sportscar;
42     My_Car   My_SUV_Car    = My_Car.SUV;
43
44 }
```

SAFER STORAGE

ESDL programs are entirely statically resolvable: all objects that are required are known at code generation time and programs have bounded space and time requirements. ESDL has no runtime library; it relies only on the C runtime (i.e. the C startup code) for object creation and initialization. Consequently, ESDL has:

- No dynamic memory (no use of heap storage, no dynamic object creation, no need for garbage collection).
- No dynamic dispatching: the language is statically typed.
- No run-time exceptions: potential runtime errors are guarded against by defensive code generation.

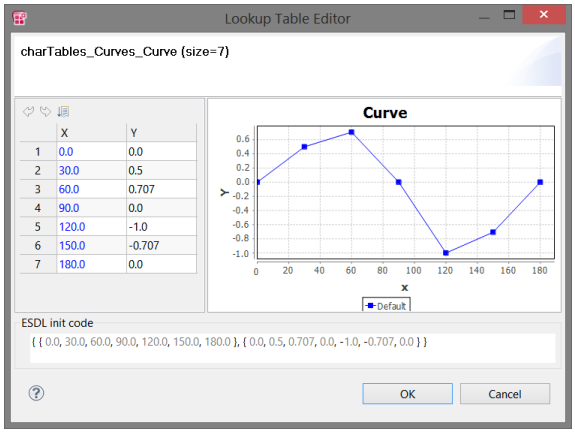
SAFER COMMUNICATION

ESDL provides *messages*, a thread-safe inter-process communication mechanism. Messages are declared at global scope and have one writer and multiple readers. Communication with messages uses state-based communication which, in a pre-emptive real-time environment (such as OSEK or AUTOSAR OS), can be highly optimized [Polenda96]. Code generation ensures that a thread sees a temporally consistent “snapshot” of all the messages it reads throughout its execution

SAFER ABSTRACTIONS

ESDL provides a series of domain-specific abstractions tailored to building automotive software:

- **Maps, curves** and **axes** to allow complex, computationally intensive functions to be implemented as look-up tables with interpolation between sample points. Maps and curves can share axes, allowing the computationally intensive axis search result to be re-used multiple times in simpler interpolation calculations to improve performance. Tool support in ETAS ASCET provides simple tabular definition and visualization:




- **System constants** for compile-time configurable values for building data and code variation points into the program. Unlike C pre-processor variables, system constants are integrated simply in the normal program (they behave like normal `const` data) and can define permitted values (or rely on a type).

```
10 sysconst integer CYLINDERS in 1 .. 4,6,8,12 = 4;
11
12 type CylinderTemperature is array [1 .. CYLINDERS] of Temperature;
```

All ESDL system constants are generated in C with compile-time protection against incorrect value definition.

```
#ifdef CYLINDERS
/**< check validity of variation constant CYLINDERS */
#if (CYLINDERS < 1 || CYLINDERS > 4) && CYLINDERS != 6 && CYLINDERS != 8 && CYLINDERS != 12
#error Value of <CYLINDERS> is not one of the allowed values <1 .. 4,6,8,12>
#endif /* (CYLINDERS < 1 || CYLINDERS > 4) && CYLINDERS != 6 && CYLINDERS != 8 && CYLINDERS != 12 */

#define COMPONENT_IMPL_CYLINDERS   CYLINDERS
#else
#define COMPONENT_IMPL_CYLINDERS   4
#endif /* CYLINDERS */
```

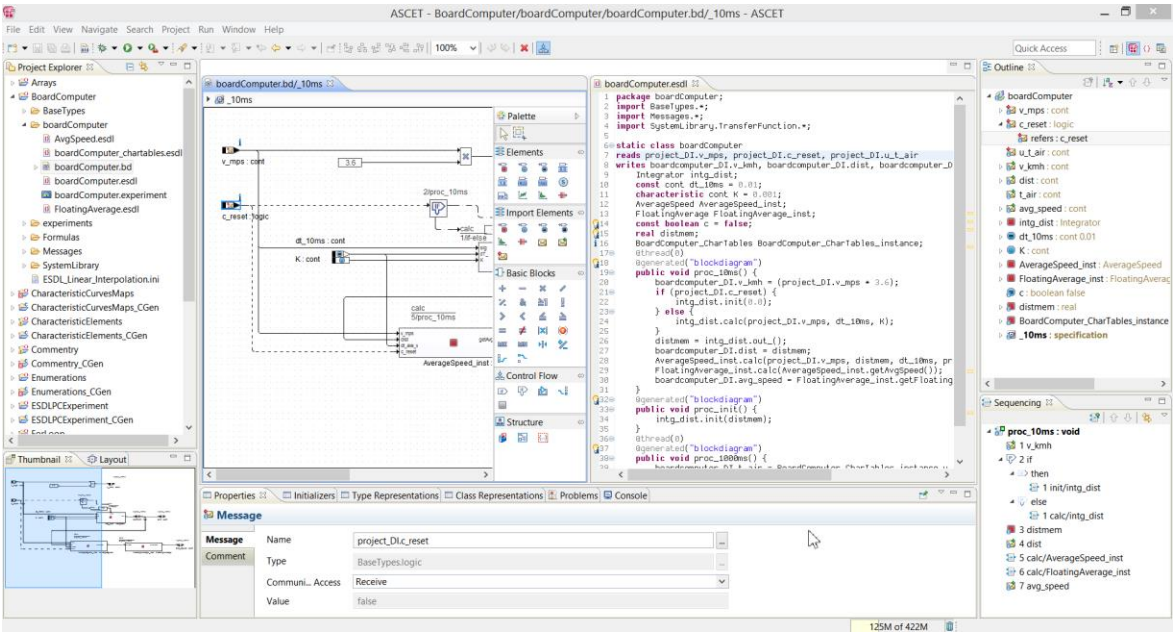
- **Dependent characteristics** to enable relationships between values to captured, Through ASAM MCD-2 MC (aka. ASAP2 or A2L) generation, a calibration system like ETAS INCA can then ensure that changes to the base characteristic is reflected across all dependent characteristics. **Virtual measurements** are also provided to allow the model to make a measurement available to simplify the calibration process without needing to allocated space in ECU memory.

```
    const real PI = 3.142;
    characteristic real Radius = 0.0;

    characteristic real Diameter <- Radius * 2.0;
    // Diameter can be updated automatically when Radius is updated

    virtual real Circumference <- Radius * 2.0 * PI;
    // Make a virtual measurement available
```

## TOOL SUPPORT

The latest version of the ETAS ASCET model-based software development tool places ESDL at its core. ETAS ASCET brings an integrated graphical and textual modelling tool to the Eclipse platform. Using Eclipse allows development teams to leverage the power of a huge eco-system in an open tool integration platform.

ETAS ASCET uses ESDL to store all the semantic information about models – including that of graphical block diagram models. Block diagram modelling is just another way to create ESDL – one that is suited to engineers with an electrical or mechanical engineering background. The multi-paradigm approach within a single modeling platform enables engineers from different disciplines to build solutions quickly and effectively. The ETAS ASCET development tooling is shown below.




## CONCLUSION

No language can *guarantee* better code, but removing the obvious pitfalls and generating code that is otherwise hard to write increases the likelihood of programs behaving as intended. There is always a trade-off in development between the

discipline exercised to prevent the introduction of errors and the effort required to detect (and then fix) those that are introduced. ESDL encourages more discipline that traditional C but in return reduces coding effort by:

- Limiting the introduction of design errors
- Eliminating random coding errors through use of code generation
- Systematically including defensive programming checks

These require adoption of ESDL, a C-like language to capture the additional design information: implicit assumptions about what the program might do are formalized as explicit parts of the source code. Typically this information appears as type constraints that need only appear once, improving overall code readability and making the code base more robust to change. This is particularly helpful when addressing review & inspection obligations arising from safety and security standards like ISO26262 and IEC61508.

## REFERENCES


1. [Bessey10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak & Dawson Engler, *"A Few Billion Lines of Code Later"*, Communications of the ACM, Vol. 53, No. 2, February 2010.
2. [CC09] *"Common Criteria for Information Technology Security Evaluation"*, Version 3.1, Revision 3, July 2009.
3. [Ernst02] Ernst el al, *"An Empirical Analysis of C Preprocessor Use"*, IEEE Transactions On Software Engineering, Vol. 28, No. 12, December 2002
4. [German03] Andy German, *"Software Static Code Analysis Lessons Learned"*, CrossTalk Magazine: The Journal of Defense Software Engineering, pp13-17, November 2003.
5. [Honekamp99] U.Honekamp, J.Reidel, K.Werther, T.Zurawka & T.Beck, *"Component-node-network: three levels of optimized code generation with ASCET-SD"*, Proceedings of the 1999 IEEE International Symposium on Computer Aided Control System Design, pp243-248, 1999.
6. [IEC10] IEC 61508 Functional safety of electrical / electronic / programmable electronic safety-related systems, March 2010.
7. [DoughtyWhite15] Pearl Doughty-White & Miriam Quick, *"Codebases"*, www.informationisbeautiful.net/visualizations/millions-lines-of-code, September 2015.
8. [ISO11] ISO TC22/SC3, *"ISO 26262: Road Vehicles - Functional Safety"*, November 2011.
9. [ISO12] ISO WG 23/N 0389, *"Information Technology — Programming Languages — Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use"*, ISO/IEC TR 24772, Edition 2, January 2012.
10. [MIRA13] MIRA Limited, *"MISRA:C-2012 Guidelines for the use of the C language in critical systems"*, March 2013.
11. [Polenda96] S.Polenda, *"Optimizing Interprocess Communication for Embedded Real-Time Systems"*, Proceedings of the 17th IEEE Real-Time Systems Symposium, Washington DC, 1996.
12. [Seacord14] R.C.Seacord, *"The CERT C Coding Standard: 98 Rules for Developing Safe, Reliable and Secure Systems"*, Second Edition, SEI Series in Software Engineering, Addison-Wesley, 2014


## DEFINITIONS, ACRONYMS, ABBREVIATIONS

ASAM: Association for Standardisation of Automation and Measuring Systems

ESDL: Embedded Software Development Language

MISRA: Motor Industry Standards and Reliability Association