The Role of Static Program Analysis in Functional Safety

Roberto Bagnara, Abramo Bagnara, Patricia M. Hill

bugseng

http://bugseng.com

& Applied Formal Methods Laboratory
University of Parma, Italy

IWES 2021 — 6th Italian Workshop on Embedded Systems
Rome, December 10th, 2021

Outline I

Static vs Dynamic Program Analysis

Static Program Analysis

The automated analysis of computer software that is performed without actually executing programs

It comprises a number of (not disjoint) techniques, including:

- AST visits
- Control-flow analysis
- Data-flow analyis
- Model checking
- Abstract interpretation
- Model checking
- Symbolic execution
- Deductive methods
- ...

Static Analysis and ISO 26262

Static analysis can be used to comply with several of the objectives of ISO 26262:

- Part 6 "Product development at the software level"
- Part 9 "Automotive safety integrity level (ASIL)-oriented and safety-oriented analyses"

Static analyzers also have a role in achieving the objectives (and, in turn, will have to comply with the prescriptions) of

- Part 8 "Supporting processes", Section 11 "Confidence in the use of software tools"

Static Analysis and EN 50128

Static analysis is at the basis of several techniques/measures of EN 50128:

- Software Architecture (7.3)
- Software Design and Implementation (7.4)
- Verification and Testing (6.2 and 7.3)
- Common Cause Failure Analysis (D.9)

Static analyzers also have a role in achieving the objectives (and, in turn, will have to comply with the prescriptions) of

- Section 6.7 “Support tools and languages”

Table 1 — Topics to be covered by modelling and coding guidelines

Topics		ASIL				S.A.
		A	B	C	D	
1a	Enforcement of low complexity	++	++	++	++	✓
1b	Use of language subsets	++	++	++	++	✓
1c	Enforcement of strong typing	++	++	++	++	✓
1d	Use of defensive implementation techniques	+	+	++	++	✓
1e	Use of well-trusted design principles	+	+	++	++	✓
1f	Use of unambiguous graphical representation	+	++	++	++	–
1g	Use of style guides	+	++	++	++	✓
1h	Use of naming conventions	++	++	++	++	✓
1i	Concurrency aspects	+	+	+	+	–

Table 3 — Principles for software architectural design

Methods		ASIL A	ASIL B	ASIL C	ASIL D	S.A.
1a	Appropriate hierarchical structure of software components	++	++	++	++	✓
1b	Restricted size and complexity of software components	++	++	++	++	✓
1c	Restricted size of interfaces	+	+	+	++	✓
1d	Strong cohesion within each software component	+	++	++	++	✓
1e	Loose coupling between software components	+	++	++	++	✓
1f	Appropriate scheduling properties	++	++	++	++	–
1g	Restricted use of interrupts	+	+	+	++	–
1h	Appropriate spatial isolation of the software components	+	+	+	++	–
1i	Appropriate management of shared resources	++	++	++	++	✓

Table 6 — Design principles for software unit design and implementation

Methods		ASIL A	ASIL B	ASIL C	ASIL D	S.A.
1a	One entry and one exit point in subprograms and functions	++	++	++	++	✓
1b	No dynamic objects or variables, or else online test during their creation	+	++	++	++	✓
1c	Initialization of variables	++	++	++	++	✓
1d	No multiple use of variable names	++	++	++	++	✓
1e	Avoid global variables or else justify their usage	+	+	++	++	✓
1f	Limited use of pointers	+	++	++	++	✓
1g	No implicit type conversions	+	++	++	++	✓
1h	No hidden data flow or control flow	+	++	++	++	✓
1i	No unconditional jumps	++	++	++	++	✓
1j	No recursions	+	+	++	++	✓

Table A.4 — Software Design and Implementation

TECHNIQUE/MEASURE	SIL					S.A.
	0	1	2	3	4	
Formal Methods	-	R	R	HR	HR	–
Modelling	R	HR	HR	HR	HR	–
Structured methodology	R	HR	HR	HR	HR	✓
Modular Approach	HR	M	M	M	M	✓
Components	HR	HR	HR	HR	HR	✓
Design and Coding Standards	HR	HR	HR	M	M	✓
Analysable Programs	HR	HR	HR	HR	HR	✓
Strongly Typed Programming Language	R	HR	HR	HR	HR	✓
Structured Programming	R	HR	HR	HR	HR	✓
Programming Language	R	HR	HR	HR	HR	✓
Language Subset	-	-	-	HR	HR	✓
Object Oriented Programming	R	R	R	R	R	✓
Procedural programming	R	HR	HR	HR	HR	✓
Metaprogramming	R	R	R	R	R	✓

Table A.12 — Coding Standards

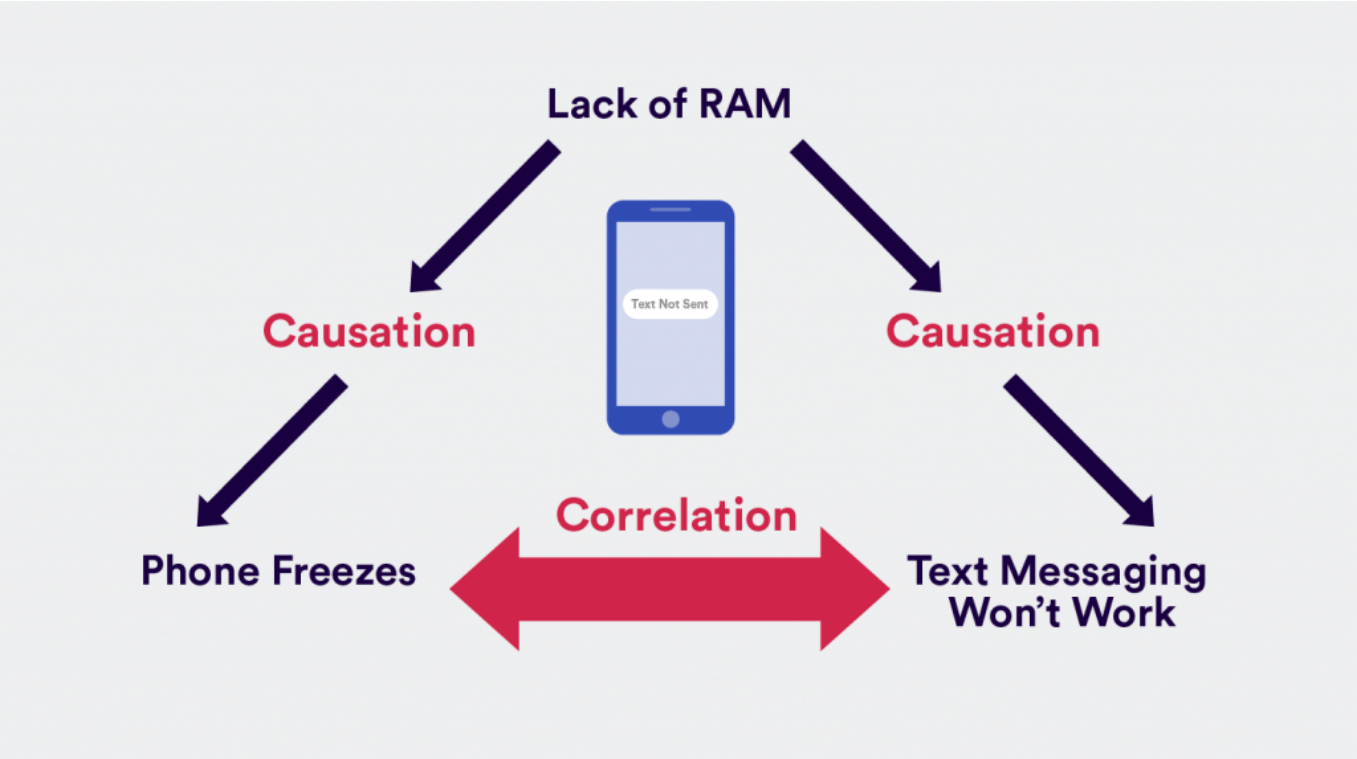
TECHNIQUE/MEASURE	SIL 0	1	2	3	4	S.A.
Coding Standard	HR	HR	HR	M	M	✓
Coding Style Guide	HR	HR	HR	HR	HR	✓
No Dynamic Objects	-	R	R	HR	HR	✓
No Dynamic Variables	-	R	R	HR	HR	✓
Limited Use of Pointers	-	R	R	R	R	✓
Limited Use of Recursion	-	R	R	HR	HR	✓
No Unconditional Jumps	-	HR	HR	HR	HR	✓
Limited size and complexity of Functions, Subroutines and Methods	HR	HR	HR	HR	HR	✓
Entry/Exit Point strategy for Functions,	R	HR	HR	HR	HR	✓
Limited number of subroutine parameters	R	R	R	R	R	✓
Limited use of Global Variables	HR	HR	HR	M	M	✓

ISO 26262 Concept: Cascading Failure

A cascading failure (CF) is a failure that causes an element to fail, which in turn causes a failure in another element

(Image courtesy of Medium)

ISO 26262 and EN 50128 Concept: Common Cause Failure



A common cause failure (CCF) is the failure of two or more elements resulting directly from a single specific event (root cause)

(Image courtesy of CleverTap)

ISO 26262 Concepts: Dependent Failure

A dependent failures (DF) is either a cascading failure or a common cause failure (and nothing else)

The union of CFs and CCFs gives DFs

$$\text{DFs} = \text{CFs} \cup \text{CCFs}$$

In other words, failures A and B are DFs if they are not statistically independent:

$$P(A \cap B) \neq P(A) \cdot P(B)$$

ISO 26262 Concepts: Freedom From Interference and Independence

Freedom From Interference (FFI) is the absence of cascading failures (CFs) between two or more elements that could lead to the violation of a safety requirement

Independence is the absence of dependent failures (DFs) between two or more elements that could lead to the violation of a safety requirement

As CFs are a subset of DFs, FFI is instrumental in achieving independence

Note: No Mention of ASIL Yet!

Q: Why haven't we mention Automotive Safety Integrity Levels (ASILs) yet?

A: Because freedom from interference and independence are ASIL-independent concepts

ASILs come into play when considering the applications of these concepts in ISO 26262

Freedom From Interference and Independence: What For?

Achievement of independence or freedom from interference between the software architectural elements can be required because of:

1. the application of an ASIL decomposition at the software level

   independent elements with lower ASIL implementing an element addressing a safety goal with higher ASIL

2. the implementation of software safety requirements

   e.g., independence between the monitored element and the monitor

3. required coexistence of the software architectural elements

   e.g., functions implementing different architectural elements within the same program must not interfere with each other

Coexistence of Elements

ISO 26262 Part 9 gives criteria for coexistence of elements

When coexistence is required there are two options:

1. All coexisting sub-elements are developed in accordance to the highest ASIL applicable to the sub-elements: expensive!
2. Absence of interference between the sub-elements has to be demonstrated

Absence of interference means:

> there are no CFs from a sub-element with no ASIL assigned (QM), or a lower ASIL assigned, to a sub-element with a higher ASIL assigned, such that these CFs lead to the violation of a safety requirement of the element

In a sense, absence of interference is a weak form of FFI

Independence in EN 50128

EN 50128, Section 7.3.4.9

Where the software consists of components of different software safety integrity levels then all of the software components shall be treated as belonging to the highest of these levels unless there is evidence of independence between the higher software safety integrity level components and the lower software safety integrity level components. This evidence shall be recorded in the Software Architecture Specification.

Freedom From Interference in ISO 26262

Must be developed and evaluated taking into account faults concerning:

- timing and execution
- memory
- exchange of information

For ASIL D, software partitioning must be supported by dedicated hardware features or equivalent

An MPU is typically used for this purpose; however, this can only enforce partitioning of memory areas and SoC peripherals, other measures are required in order to ensure freedom of interference

How Can Static Analysis Help?

Compliance to the MISRA guidelines reduces the risk of execution blocking due to unexpected excessive loop iterations (timing and execution) as well as of stack overflow (memory)

Static analysis is also instrumental in checking system decomposition!

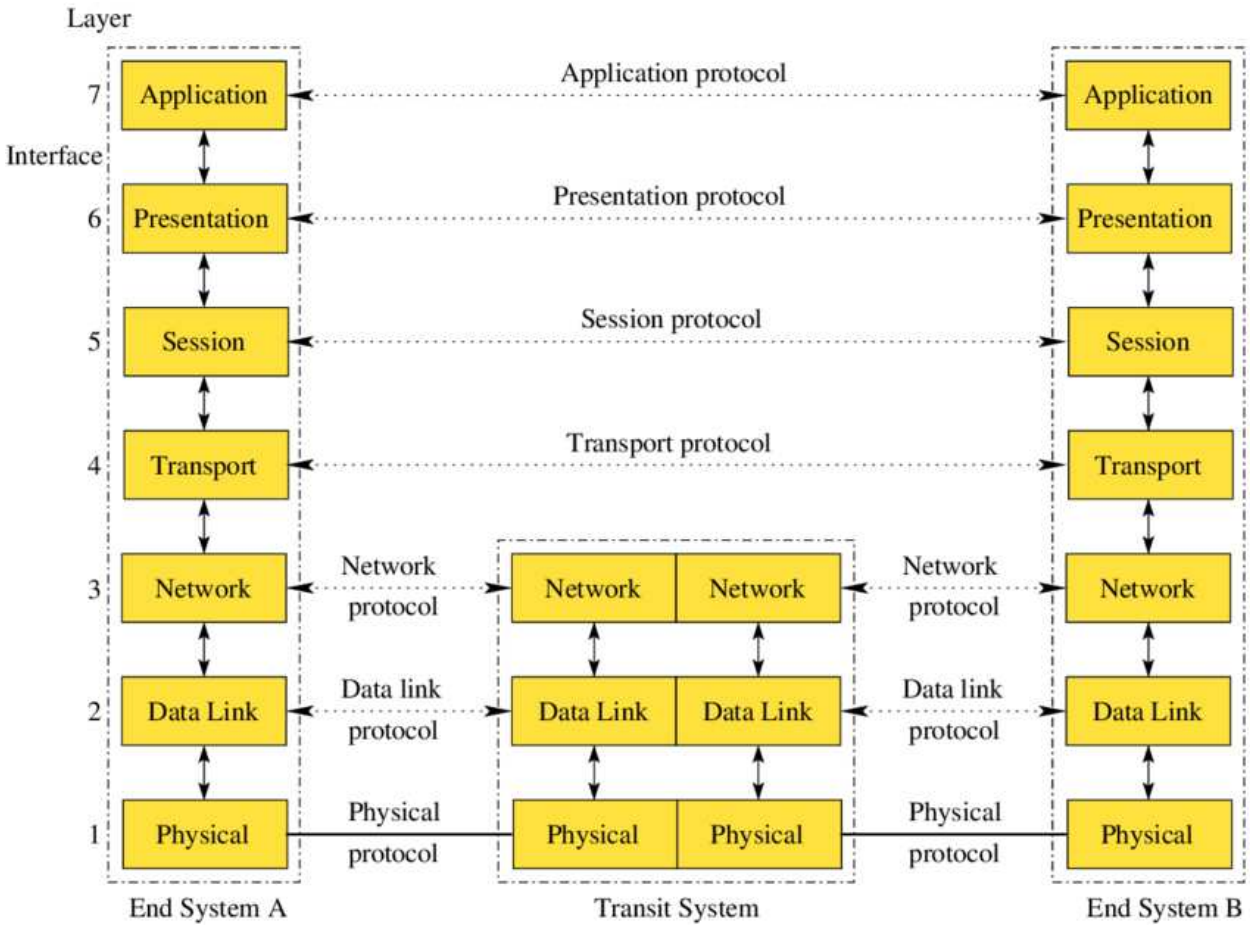
1. Checking dynamic/run-time dependencies, by tracking:

   functions/methods actions: call
   pointer variables actions: pointee read or write
   other variables actions: read or write
   structure fields actions: read or write

2. Checking static/compile-time dependencies, by tracking:

   files actions: include
   macros actions: expand (with finer control on the expanded macro)

ISO "Open System Interconnect" Model



(Image courtesy of Jürgen Foag)

Checking System Decomposition: How? When?

Once you have decomposed your system and designed the allowed interactions between components, how do you check that the implementation complies to this aspect of the design?

By hand: time consuming, error prone. . . , all multiplied by the number of times you need to redo the check

- checking only at the end is asking for trouble

By static analysis: some little time to encode the decomposition into a configuration, then the check is completely automatic and reliable

- can be part of your continuous integration processes

Example: Simplified OSI Model

An application software program needs to use the services of a network stack

The network stack has three components corresponding to OSI layers, identified by `DATA_LINK`, `NETWORK` and `TRANSPORT`

We want to make sure the network layers are not bypassed

E.g. if the `DATA_LINK` component is accessed bypassing the `NETWORK` component, then a packet that cannot be routed may be built

In order to add interest, we want to allow an exception: the `APPLICATION` component may call the `link_status()` function in `DATA_LINK`

OSI Example: Component Entities

In this example, we only consider variables and functions with external linkage in user code (not system code):

```
-config=B.PROJORG,all_component_entities+=
  "linkage(external)
     &&kind(var||function)
     &&all_decl(loc(top(file(kind(user||main_file)))))"
```

In order to save typing, we exploit the fact that we used a proper (MISRA compliant) header file discipline:

```
-config=B.PROJORG,component_entities+=
  {"DATA_LINK/Except",content,"^link_status\\(.*$"},
  {DATA_LINK,content,"any_decl(loc(top(file(^dl\\.h$))))"}
  {NETWORK,content,"any_decl(loc(top(file(^nl\\.h$))))"},
  {TRANSPORT,content,"any_decl(loc(top(file(^tl\\.h$))))"}
  {APPLICATION,content,"^main\\(.*$"}
```

OSI Example: Component Files

In addition to checking dynamic/run-time dependency, we may check static/compile-time dependency, such as header file inclusion

To do this, we need first to assign source files to components:

```
-config=B.PROJORG,component_files+=
    {"DATA_LINK/Except","^dl\\.h$"},
    {DATA_LINK,"^dl\\.c$"},
    {NETWORK,"^nl\\.[ch]$"},
    {TRANSPORT,"^tl\\.[ch]$"},
    {APPLICATION,"^main\\.c$"},
    {"","kind(main_file||user)"}
```

OSI Example: The over Relation

We define a relation between components called over, meaning directly above in the OSI model:

```
-config=B.PROJORG,component_relation+=
    {APPLICATION, over, TRANSPORT},
    {TRANSPORT, over, NETWORK},
    {NETWORK, over, DATA_LINK}
```

OSI Example: Permissions

Entities in component A may include and call entities in component B if A is over B:

```
-config=B.PROJORG,component_allows+=
    "rel(over)&&action(include||call)"
```

As an exception, the `APPLICATION` component may call the `link_status()` function and include `DATA_LINK`'s header file:

```
-config=B.PROJORG,component_allows+=
    "from(APPLICATION)
       &&to(DATA_LINK/Except)&&action(include||call)"
```

Conclusion

Static analysis plays an important role in achieving the objectives of ISO 26262 and EN 50128

We gave a brief (approximate and incomplete) introduction to the ISO 26262 notions dependence, freedom from interference and absence of interference

We have shown how the ability to strictly control the interactions of software components by static analysis is instrumental in gathering the required evidence in a reliable way

We did not have time to highlight other important applications of static analysis in the context of ISO 26262 and EN 50128

The End


Questions?
roberto.bagnara@unipr.it
info@bugseng.com
bugseng