Formal Verification of Software Architectural Constraints

Roberto Bagnara
University of Parma
Parma, Italy
Email: roberto.bagnara@unipr.it

Abramo Bagnara
BUGSENG
Parma, Italy
Email: abramo.bagnara@bugseng.com

Patricia M. Hill
BUGSENG
Parma, Italy
Email: patricia.hill@bugseng.com

***Abstract*—In the development of high-integrity software, all interactions between components must satisfy design constraints. Hierarchical levels must not be bypassed: if the design prescribes that software layer A cannot interact directly with layer C without the intermediation of layer B, this is something that must be verified. If components with different criticalities have to coexist on the same ECU, huge savings are possible if we can prove that lower-criticality components cannot interfere with higher-criticality ones. Effectiveness of monitoring safety mechanisms crucially depends on the independence between the monitored element and the monitor. In this paper, after recollecting the basic ideas and methodologies of software decomposition, we introduce the notions of independence and interference in ISO 26262 and other functional safety standards. We then show how architectural constraints at the software level can be formally verified by means of control and data flow static analyses. This technique allows for the formal verification of the design restricting all interactions between user-defined software elements: this includes dynamic, run-time dependencies such as read or write accesses to shared memory, function calls, passing and returning of data, as well as static dependencies due to header file inclusion and macro expansion. The formalization we present does not rely on complex logic formalisms and its implementation in the tool ECLAIR has been certified by TÜV SÜD.**

I. Introduction

It is well known that managing the complexity of large and/or critical software systems requires *decomposition*, that is, the system is divided into parts that are small enough to be reasoned upon, maintained and tested *in isolation*, i.e., independently from one another. At a very superficial level, software decomposition is advantageous in itself. For instance, it allows:

- organizing the program text into manageable chunks;
- separate compilation;
- decreasing some metrics related to program complexity.

These are nice to have, but the real advantage of software decomposition is when the interactions between the components are kept under strict control. In fact, the components of a system and their interactions can be modeled as a graph: if any component may interact with any other component, the complexity of the interaction may grow quadratically with the number of components. In extreme cases, failure to control the interactions among components can give rise to "spaghetti code": this pejorative expression is usually associated to the abuse of `goto`'s (the control-flow flavor of spaghetti code) and global variables (the data-flow flavor of spaghetti code), but attitudes like "every function may call any function" and "every function may read and/or write any variable" give rise to the same phenomena.

When software components have been defined, two metrics can help assessing the partitioning: *coupling* and *cohesion*.

Cohesion measures dependency among the parts of a component, so that

- cohesion is high when parts of the component cooperate to the same tasks and are strongly related;
- cohesion is low when the component happens to be a container for unrelated parts that do not cooperate or do so only marginally.

Coupling measures dependency among components, so that

- coupling is high when changing one component is likely to have an important impact on other components;
- coupling is low when changing one component does not affect other components or does so only marginally.

As a consequence, effective software decomposition aims at breaking down a program into components so as to achieving low coupling and high cohesion.

There are several software decomposition paradigms, such as procedural/algorithmic decomposition, abstract data types, and object-oriented decomposition. Note that these are orthogonal to the programming models: linguistic support helps, but it is not essential. In addition, there are several criteria for effective system decomposing, including:

Layering: Any system should be designed and built as a hierarchy of layers, where each layer uses only the services offered by the lower layers [3]. A layer is a component that provides services at a given level of abstraction: it depends only on lower layer and it has no knowledge of higher layers.

Partitioning: Vertically divide a system into several independent (or weakly-coupled) components. Partitions are divisions at the same level of abstraction.

Information hiding: Hide all what is likely to change [4].

Several criteria are often used at the same time, e.g., combining partitions and layers. Note also that there are variants whose suitability depends on the design goals. For instance, we have

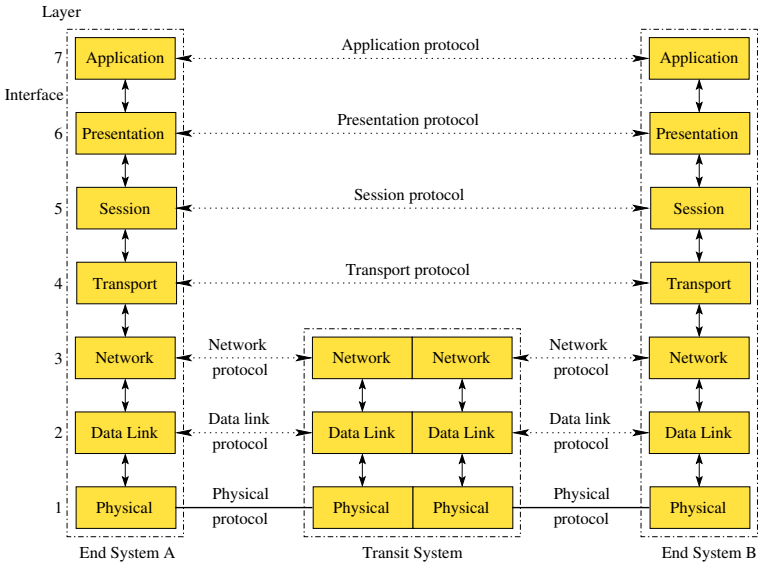

Fig. 1. Architecture model according to the ISO/OSI Reference Model [1] (image courtesy of Jürgen Foag [2])

- *opaque layering*: a layer may only refer to the layer immediately below (when the main design goals are maintainability and flexibility);
- *transparent layering*: a layer may refer to any layer below (when run-time efficiency is an additional design goal).

A classical example of an architecture model based on layering is represented by the ISO/OSI *Reference Model* [1]: this is depicted in Figure 1.

Another example is given by the *AUTOSAR Classic Platform*, which specifies a detailed layered software architecture with rather strict constraints [5]. The *AUTOSAR Adaptive Platform* is deliberately much less detailed and strict in order to provide more latitude to stack vendors for their solution design [6].

Proper decomposition (low coupling, high cohesion) is important for several reasons, some of which apply to all kinds of systems:

- easier design and development;
- easier component verification;
- easier system integration and verification;
- easier maintenance.

Safety and security concerns make things even more interesting, as unwanted or unknown dependencies or interference among components may easily turn into safety and/or security issues. Consider two software components A and B and an uncontrolled dependency of A from B:

- a failure in B might unexpectedly cause a failure in A (a safety issue);
- a vulnerability of B might unexpectedly turn into a vulnerability of A (a security issue).

For this reason, the theme of software decomposition goes, for safety- and/or security-critical systems, well beyond the need of managing complexity from the software design and engineering point of view. In fact, all functional safety and cybersecurity standards have (to a varying degree of precision and formality) requirements in terms of component independence, interference, isolation (the terminology changes passing from one standard to another).

In this paper we review, in Section II, the definitions and prescriptions of the ISO 26262 automotive functional safety standard related to partitioning and independence of components [7]. This standard is currently the one where such notions are detailed in the most comprehensive way. We then review, in Section III, how such notions are explicitly or implicitly referred to in other functional safety and cybersecurity standards. Section IV discusses how architectural constraints at the software level can be enforced by static analysis in C and C++. The way in which these ideas are implemented in the *ECLAIR Software Verification Platform*® is presented in Section V. Section VI concludes the paper.

II. Partitioning and Independence in ISO 26262

ISO 26262:2018 defines *freedom from interference* (FFI) as "absence of *cascading failures* between two or more *elements* that could lead to the violation of a *safety* requirement" [8, Clause 3.65]. Simply put, a *cascading failure* (CF) is a failure that causes an element to fail, which in turn causes a failure in another element [8, Clause 3.17], whereas a *common cause failure* (CCF) is the failure of two or more elements resulting directly from a single specific event (root cause) [8, Clause 3.18]. The union of CFs and CCFs gives what ISO 26262:2018 calls *dependent failures* (DFs), namely, failures that are not statistically independent [8, Clause 3.29].

The notion of DF comes into play in the definition of one aspect of *independence*: "absence of dependent failures between two or more elements that could lead to the violation of a safety requirement" [8, Clause 3.78].[1] As CFs are a subset

[1]This is the technical aspect of *independence*, the other aspect being the organizational one.

of DFs, FFI is instrumental in achieving independence. In turn, achievement of independence or freedom from interference between the software architectural elements can be required because of:

a. the application of an ASIL decomposition at the software level;
b. the implementation of software safety requirements;[2] or
c. the required coexistence of the software architectural elements [9, Annex E].

Concerning point c., criteria for the coexistence of elements are given in [10, Clause 6]. 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; (2) the guidance provided in [10, Clause 6] is used to determine whether sub-elements with different ASILs can coexist within the same element. Such guidance is based on the analysis of *interference* of each sub-element with other sub-elements: evidence has to be provided to the effect that 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.[3]

III. Partitioning and Independence in Other Standards

The notions and requirements that ISO 26262 spells out in rather detailed form have been part of the state-of-the-art in critical system engineering for decades. It is thus not surprising that we can find the same concepts and prescriptions, albeit in a less precise form, in all major standards, which will be reviewed in the next sections. Note that, in the quoted excerpts, emphasis is always ours and not of the quoted source.

A. DO-178C

The section on "Architectural Considerations" [11, Section 2.4] requires the system safety assessment process to establish that "sufficient independence exists between software components." It then goes on by making it clear what the consequences are of being unable to prove independence:

> If partitioning and independence between software components cannot be demonstrated, the software components should be viewed as a single software component when assigning software levels (that is, all components are assigned the software level associated with the most severe failure condition to which the software can contribute).

Partitioning is defined in [11, Section 2.4.1]:

> Partitioning is a technique for providing isolation between software components to contain and/or isolate faults and potentially reduce the effort of the software verification process. Partitioning between software components may be achieved by allocating unique hardware resources to each component [...] Alternatively, partitioning provisions may be made to allow multiple software components to run on the same hardware platform. Regardless of the method, the following should be ensured for partitioned software components:
>
> a. A partitioned software component should not be allowed to contaminate another partitioned software component's code, input/output (I/O), or data storage areas.
> b. A partitioned software component should be allowed to consume shared processor resources only during its scheduled period of execution.
> c. [...] Failures of hardware unique to a partitioned software component should not cause adverse effects on other partitioned software components.
> d. [...]
>
> The software life cycle processes should address the partitioning design considerations. These include the extent and scope of interactions permitted between the partitioned components and whether the protection is implemented by hardware or by a combination of hardware and software.

The section on "Safety Monitoring" [11, Section 2.4.3] states:

> Safety monitoring is a means of protecting against specific failure conditions by directly monitoring a function for failures that would result in a failure condition. [...] there are three important attributes of the monitor that should be determined:
>
> a. Software level: [...]
> b. System fault coverage: [...]
> c. Independence of function and monitor: The monitor and protective mechanism are not rendered inoperative by the same failure that causes the failure condition.

B. IEC 61508

In IEC 61508, the strongest incentive to ensure independence can be found in Part 1 [12, 7.6.2.10]:

> **7.8.2.10** For an E/E/PE safety-related system that implements safety functions of different safety integrity levels, unless it can be shown there is sufficient independence of implementation between these particular safety functions, those parts of the safety-related hardware and software where there is insufficient independence of implementation shall be treated as belonging to the safety function with the highest safety integrity level. Therefore, the requirements applicable to the highest relevant safety integrity level shall apply to all those parts.

[2] E.g., to provide evidence for the effectiveness of monitoring safety mechanisms by showing independence between the monitored element and the monitor.

[3] It is important to realize that "absence of *interference*" and "*freedom from interference*" are distinct concepts in ISO 26262:2018. The latter concept does not depend on ASILs or lack thereof, so that "*freedom from interference*" implies "absence of *interference*," but not the other way around.

For software, the requirements on independence and non-interference between safety functions and non-safety functions are given in IEC 1508 Part 3 [13, 7.4.2.8, 7.4.2.9]:

> **7.4.2.8** Where the software is to implement both safety and non-safety functions, then all of the software shall be treated as safety-related, unless adequate design measures ensure that the failures of non-safety functions cannot adversely affect safety functions.
>
> **7.4.2.9** Where the software is to implement safety functions of different safety integrity levels, then all of the software shall be treated as belonging to the highest safety integrity level, unless adequate independence between the safety functions of the different safety integrity levels can be shown in the design. It shall be demonstrated either (1) that independence is achieved by both in the spatial and temporal domains, or (2) that any violation of independence is controlled. The justification for independence shall be documented.

C. IEC 62304

In IEC 62304 parlance, a *medical device* can be composed of difference subsystems, some of which are *software systems*. In turn a *software system* is composed of one or more *software items*, and each *software item* is composed of one or more *software units* or decomposable *software items*. *Software units* are not further decomposed for the purposes of testing or software configuration management.

IEC 62304, in Section "Software safety classification," broadly refers to the concept of "segregation" [14], [15, 4.3]:

> d) When a SOFTWARE SYSTEM is decomposed into SOFTWARE ITEMS, and when a SOFTWARE ITEM is decomposed into further SOFTWARE ITEMS, such SOFTWARE ITEMS shall inherit the software safety classification of the original SOFTWARE ITEM (or SOFTWARE SYSTEM) unless the MANUFACTURER documents a rationale for classification into a different software safety class [...] Such a rationale shall explain how the new SOFTWARE ITEMS are segregated so that they may be classified separately.

This concept is further elaborated in [14], [15, B.4.3]:

> The software ARCHITECTURE should promote segregation of software items that are required for safe operation and should describe the methods used to ensure effective segregation of those SOFTWARE ITEMS. Segregation is not restricted to physical (processor or memory partition) separation but includes any mechanism that prevents one SOFTWARE ITEM from negatively affecting another. The adequacy of a segregation is determined based on the RISKS involved and the rationale which is required to be documented.

D. EN 50128

EN 50128 (railways) spells out the rationale for independence as follows [16, 7.3.4.9]:

> 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.

In the subsequent clause, one can find an important incentive in a formal specification of the software architecture [16, 7.3.4.10]:

> 7.3.4.10 The Software Architecture Specification shall describe the strategy for the software development to the extent required by the software safety integrity level. The Software Architecture Specification shall be expressed and structured in such a way that it is
>
> a) complete, consistent, clear, precise, unequivocal, verifiable, testable, maintainable and feasible,
> b) traceable back to the Software Requirements Specification.

E. IEC 60335

In IEC 60335 (household and similar electrical appliances), Annex R of Part 1 "Software evaluation" (normative), "independent monitoring" is cited as an example of "additional fault/error detection means" in Clause R.2.2 [17]. Section R.3.2.2 "Software architecture" has the following requirements:

> **R.3.2.2.1** The specification of the software architecture shall include the following aspects:
>
> – *techniques and measures to control software faults/errors (refer to R.2.2);*
> – interactions between hardware and software;
> – *partitioning into modules and their allocation to the specified safety functions;*
> – *hierarchy and call structure of the modules (control flow);*
> – interrupt handling;
> – *data flow and restrictions on data access;*
> – architecture and storage of data;
> – time-based dependencies of sequences and data.
>
> **R.3.2.2.2** The architecture specification shall be validated against the specification of the software safety requirements by static analysis.

Such requirements have their counterparts in the section about "Module design and coding" [17, R.3.2.3].

F. ISO/SAE 21434

ISO/SAE 21434, in its Annex E (informative), introduces the notion of *Cybersecurity Assurance Level* (CAL), which

plays the role of *Safety Integrity Level* and similar concepts in functional safety standards [18, Annex E]. CALs are qualitative expressions of the level of cybersecurity assurance required: the higher the CAL, the higher is the rigor with which cybersecurity activities are performed. ISO/SAE 21434 does not prescribe how many CALs there are and how they are used to drive the cybersecurity engineering: this is left to individual organizations and the norm only provides a classification scheme along with examples. The example given in [18, Annex E] defines four CALs, from CAL 1 to CAL 4, which are assigned depending on two attributes of the relevant threat scenarios: their maximum impact (from low to high: *negligible*, *moderate*, *major* or *severe*) and the attack vector (from less to more risky: *physical*, *local*, *adjacent* or *network*).[4] Of course, threat scenarios that admit network attack vectors are the more risky ones, and higher CALs correspond to higher risk. What is important to note, though, is that the risk value is dynamic and depends also on external factors. For example, when a vulnerability is discovered in the field and becomes public, risk becomes higher; when, subsequently, the vulnerability is fixed or other countermeasures have been applied, risk drops.

Annex E of ISO/SAE 21434 stipulates that

> In product development, if cybersecurity requirements are allocated to components, and *isolation from other components cannot be confirmed*, then the components can be developed in accordance with *the highest CAL* for those cybersecurity requirements.

Among the requirements of ISO/SAE 21434, it is worth mentioning

> **[RC-10-06]** Established and trusted design and implementation principles should be applied to avoid or minimize the introduction of weaknesses.
>
> NOTE 7 Examples of design principles for architectural design for cybersecurity are given in NIST Special Publication 800-160 Vol. 1 *[19]*, appendix F.1.

In turn, Appendix F of [19], which extensively cites the rest of the same document, discusses, among many other things, the importance of decomposition, layering, isolation and non-interference. In particular, Appendix F.1.2 of [19] "Design Considerations" states (underlined expressions are references to Appendix E of the same document):

- **Composition**
  Trustworthiness judgments are compositional. They must align with how the set of composed elements provides a system capability. The way that the system is composed from its system elements must include the design principles of Compositional Trustworthiness and, to the extent practical, Structured Decomposition and Composition.
- [...]
- **Failure Propagation**
  All systems fail at some point. When a failure occurs, another failure scenario or the creation of a new failure scenario should not be triggered or invoked (Protective Failure). Designing without single points of failure (Redundancy) — including not having common mode failures (Diversity) — can help isolate system element failures while providing the required system capabilities. Additionally, the response to failure should not lead to loss or other failures (Protective Recovery).

IV. Enforcing Architectural Constraints by Static Analysis in C and C++

Enforcing architectural constraints may require, depending on the nature of the constraints, different techniques and implementations at different levels: at the source code level, at the level of the operating system or hypervisor, at the hardware level. For instance, freedom from interference in ISO 26262 must be developed and evaluated taking into account faults concerning [9, Annex D]:

- timing and execution, including blocking of execution, deadlocks, livelocks, incorrect allocation of execution time, and incorrect synchronization between software elements;
- memory, including corruption of content, inconsistent data (e.g., because of data races), stack overflow and underflow, read or write access to memory allocated to another software element;
- exchange of information, including, among many other instances, communication via global or file-scope variables and via memory-mapped I/O registers, the incorrect addressing of information and so on.

ISO 26262 explicitly mentions static analysis among the verification methods that are appropriate for demonstrating freedom from interference.

Static analysis is not always sufficient, though. For instance, in ISO 26262 and for ASIL D, software partitioning must be supported by dedicated hardware features or equivalent means [9, Clause 7.4.9]. An MPU is typically used for this purpose; however, as this can only enforce partitioning of memory areas and system-on-chip peripherals, other measures are required in order to fully ensure freedom of interference and, again, static analysis is by far the best technique for anything that is observable at the source code level.

So, how can static analysis help? First, observe that compliance to the MISRA guidelines [20], [21], [22] reduces the risk of crashes and of execution blocking due to unexpected excessive loop iterations (in the *timing and execution* category of ISO 26262). MISRA compliance also provides protection against stack and buffer overflow (in the *memory* category of ISO 26262). With the forthcoming publication of the *Amendment 4* to MISRA C:2012, which completes

[4] Thread scenarios of *negligible* impact are assigned no CAL in the example. This does not mean that no cybersecurity activities are in order, only that the level of rigor required may not justify the adoption of the requirements and guidelines of ISO/SAE 21434.

MISRA C:2012 coverage of C11 and C18 features, guidance will be extended to cover also deadlocks, some forms of incorrect synchronization, and data races.

Secondly, and this is actually the main point of this paper, static analysis is also instrumental in checking system decomposition by tracking run-time as well as compile-time dependencies.

A. Constrain Dynamic/Run-time Dependencies

Static analysis can reliably detect calls to C/C++ functions and C++ methods. Of course, when such calls happen via pointers, the check cannot be exact in all cases, but it can err on the safe side. That is, when the analysis cannot exclude that a call via pointer violates an architectural constraint, it will flag the potential call site. In this case, developers will have to investigate further but, nonetheless:

- this is still orders of magnitude better than checking architectural constraints by peer review alone;
- given that calls via pointers are infrequent, as is the address-taking of functions and methods, the phenomenon is rare.

Static analysis is also the perfect technique to track read and/or write access to scalar and aggregate variables. It is of course very important that reads and writes are tracked separately: from the safety point of view, reading a variable is generally less critical than writing to it, as the latter may cause data corruption. From the security point of view, though, unwanted reading may cause unintended information disclosure. Moreover, when it comes to embedded systems, reading memory-mapped I/O registers can trigger all kinds of side-effects just as when writing to them.

For any static analysis tool meant to assist in checking the compliance with architectural constraints, the ability to track accesses to individual structure fields is very important. Think about a network stack that defines a packet data structure: in general, not all layers should be granted read/write access to all fields. For example, if the transport layer uses sequence numbers to allow the receiving end to deliver the packets in the order in which they were transmitted by the other end, the network and data link layers should certainly not write to the sequence number field.

Let us consider the simplified scenario described in [23]. When focusing on a particular safety requirement, the C functions constituting the Linux Kernel can be partitioned in two classes: *Safety-Related* (SR), and *Not Safety-Related* (NSR). If we only consider functions, the corresponding safety architectural constraint is that SR functions may only call other SR functions. In fact, if function `f()` is safety-related, hence it must be developed and tested according to safety-adequate standards, and `f()` calls `g()`, then also `g()` must be developed and tested according to such standards. Roughly speaking, a reliable function cannot call an unreliable one, for otherwise neither would be reliable.

Tracking functions is not sufficient, as the Linux Kernel internal state is represented by many persistent data structures pointed to by global and file-scope variables. The enforcing of architectural constraints on their access is instrumental in ensuring that corruption of such data structures cannot take place.

B. Constrain Static/Compile-time Dependencies

Module dependencies can also manifest themselves at compile-time and so, a fortiori, static analysis is the ideal tool to keep them under control. They can concern header files, where the relevant action is their inclusion via `#include` directives. By following the principle of single responsibility (always a good software engineering practice that has various benefits), the content of header files is minimized which, in turn, can lead to strong guarantees of independence: if a module does not even see the declaration of a variable or a field, surely it cannot access it.

One example of architectural constraints that can be enforced in this way is the one whereby the only module that can directly interface with hardware is the *Hardware Abstraction Layer* (HAL). This is easy to implement if, e.g., all the header files that allow direct interaction with the hardware are, say, in the `hardware/` folder.

Hardware-related header files give us the opportunity of explaining one of the reasons why the ability of tracking macro expansion is so important. The header files provided by hardware manufacturers are generally not partitioned in a way that reflects criticality. Consider, for example the header file partially reproduced in Figure 2. The *sysinfo* block, the one starting at address `SYSINFO_BASE`, is read-only and contains system information, such as the Chip ID: this is not very critical. The *watchdog* block, starting at address `WATCHDOG_BASE`, is much more critical, as mishandling watchdogs can defeat their whole purpose [24], [25]. The ability to track macro expansion at the highest level of granularity allows precise control of those hardware features different parts of the HAL will have access to.

C. The Importance of Early and Continuous Enforcement of Architectural Constraints

Once the system has been decomposed and the allowed interactions between components have been defined, how can you check that the implementation complies with this aspect of the design? Checking by peer review is time consuming and error prone and, of course the effort is multiplied by the number of times the check needs to be redone. Checking only at the end of the project is asking for trouble, as any non-compliance found at that stage might require extensive reworking.

It is important to observe that violations of architectural constraints are often not very visible: a programmer might, in good faith, call a function from where it should not be called or write to a variable that should not be written there. Chances are that nobody will notice this until much later in the project when the cost of remediation is very high.

This, in addition to the strong soundness guarantees it provides, is the great advantage of checking software architectural constraints by static analysis: little time spent in encoding the

```
/**
 * Copyright (c) 2021 Raspberry Pi (Trading) Ltd.
 *
 * SPDX-License-Identifier: BSD-3-Clause
 */
#ifndef _ADDRESSMAP_H_
#define _ADDRESSMAP_H_

#include "hardware/platform_defs.h"

// Register address offsets for atomic RMW aliases
#define REG_ALIAS_RW_BITS  (0x0u << 12u)
#define REG_ALIAS_XOR_BITS (0x1u << 12u)
#define REG_ALIAS_SET_BITS (0x2u << 12u)
#define REG_ALIAS_CLR_BITS (0x3u << 12u)

Omitted parts...
#define SYSINFO_BASE _u(0x40000000)
#define SYSCFG_BASE _u(0x40004000)
#define CLOCKS_BASE _u(0x40008000)
#define RESETS_BASE _u(0x4000c000)
Omitted parts...
#define BUSCTRL_BASE _u(0x40030000)
#define UART0_BASE _u(0x40034000)
#define UART1_BASE _u(0x40038000)
Omitted parts...
#define TIMER_BASE _u(0x40054000)
#define WATCHDOG_BASE _u(0x40058000)
#define RTC_BASE _u(0x4005c000)
#define ROSC_BASE _u(0x40060000)
#define VREG_AND_CHIP_RESET_BASE _u(0x40064000)
#define TBMAN_BASE _u(0x4006c000)
#define DMA_BASE _u(0x50000000)
Omitted parts...
```

Fig. 2. Partial contents of `pico-sdk/src/rp2040/hardware_regs/include/hardware/regs/addressmap.h`, cloned from https://github.com/raspberrypi/pico-sdk.git on February 24, 2023

decomposition into a tool configuration is repaid in spades by allowing the check to to be performed in a completely automatic and reliable way, possibly as part of continuous integration processes.

V. The ECLAIR `B.PROJORG` Service

The ideas presented in this paper have been implemented in the *ECLAIR Software Verification Platform*® under the service named `B.PROJORG`, a mnemonic name for *PROJect ORGanization checker*. Despite the low-key name, which is meant not to scare away developers of non-critical systems, the service has been certified by TÜV SÜD Rail GmbH for safety-related development in compliance with ISO 26262 (up to ASIL D), IEC 61508 (up to SIL 4), EN 50128 (up to SIL 4), IEC 62304 (up to Class C), and ISO 25119 (up to SRL 3).

To see what a configuration for this service looks like, let us reconsider the example of [23] that has two classes of C functions: `SR` (Safety-Related) and `NSR` (Not Safety-Related); so we specify that we are only interested in entities of kind function:

```
-config=B.PROJORG,
  all_component_entities+="kind(function)"
```

As `SR` and `NSR` may not constitute a partition of the system, we put all other functions in a component called "`unknown`":

```
-config=B.PROJORG,component_entities+=
  {SR,content,"name(fun1)"},
  {SR,content,"name(fun2)"},
  {NSR,content,"name(fun3)"},
  {NSR,content,"name(fun4)"},
  {"unknown",content,"any()"}
```

So functions named `fun1` and `fun2` go into component `SR`, functions named `fun3` and `fun4` go into component `NSR`,

and all other functions go into component `unknown`. The safety constraint is that `SR` functions may only call other `SR` functions. As calls are always allowed between functions in the same component, we only need to specify that calls from `NSR` functions to `SR` and `unknown` functions are allowed as well as calls from `unknown` functions to `NSR` and `SR` functions:

```
-config=B.PROJORG,component_allows+=
  "from(NSR)&&to(SR||unknown)"
-config=B.PROJORG,component_allows+=
  "from(unknown)&&to(SR||NSR)"
```

All calls that are not intra-component and are not explicitly allowed are reported as violations: nothing can escape.

For a slightly more complex example, let us reconsider the ISO/OSI Reference Model in simplified form. An application software program, the `APPLICATION` component, 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`. The architectural constraint that has to be enforced is that 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, which is clearly not wanted. In order to add interest, we want to allow an exception: the `APPLICATION` component may call the `link_status()` function in `DATA_LINK`. The rationale of this exception is that becoming aware of the temporary impossibility to communicate only via timeouts is incompatible with the application goals.

For this example, we 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, so that the `NETWORK` and `TRANSPORT` components, as well as the non-exceptional part of the `DATA_LINK` component can be defined implicitly by means of their header file: `nl.h`, `tl.h` and `dl.h`, respectively:

```
-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\\(.*$"}
```

The exceptional part of the `DATA_LINK` component, denoted by `DATA_LINK/Except`,[5] is constituted by the single function named `link_status`: as we are in C, there is no need to specify the function argument types. Similarly, the only entity in the `APPLICATION` component is the `main()` function.

In addition to checking dynamic/run-time dependency, we want to check static/compile-time dependency, such as header file inclusion. To do this, we first need 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)"}
```

Now that the compile-time and run-time aspects of all components have been defined, we simplify our life by introducing a relation between components called `over` (an arbitrary identifier). We define it so that A `over` B means A is directly above B in our ISO/OSI simplified model, where we do not have the *presentation* and *session* layers:

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

Remembering that all intra-component actions are implicitly allowed, specifying the compliant interactions is now trivial. 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 is granted permission to 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)"
```

It can be seen that `B.PROJORG`, thanks to the expressiveness of its configuration language, is able to capture all sorts of of software architectural constraints. Yet, when the constraints and the components are simple, the configuration is correspondingly simple and natural. Moreover, the configuration of `B.PROJORG`, as any other ECLAIR configuration, can be rendered in plain English by service `B.EXPLAIN`, which has

[5] Observe that names of components are completely in the hand of the users: for `B.PROJORG` they are simply strings that cannot contain whitespace. Here we see the use of the '`/`' character to spell out a subcomponent of `DATA_LINK` that has peculiar properties

been designed to facilitate peer reviewing of configurations also by those unfamiliar with the configuration language.

VI. Conclusion

Proper software decomposition into components and subcomponents is a crucial element of the design of any software or software-controlled system. Decomposing and restricting the possible interaction between components is the only way to effectively dominate complexity in spite of the constant growth in the size of software artifacts.

When it comes to the design of critical systems, the enforcement of (software) architectural constraints is even more important as interactions between components can cause safety and security issues of any kind. It is thus no surprise that all safety and security standards have related prescriptions, though at different levels of detail and rigor.

In this paper, after a refresher on the topic of software decomposition in the general theory and practice of software engineering, we reviewed the contents of major functional safety and security standards on the purpose and use of software architectural constraints in the design and implementation of critical systems. We have then shown how the ability to formally and strictly control the interactions of software components by static analysis is instrumental in gathering the required evidence in a reliable way and, most importantly, after the initial configuration, in a completely automatic way.

For concreteness of exposition, we have shown how these ideas are captured in `B.PROJORG`, a service based on the *ECLAIR Software Verification Platform*®, whose configuration language is formal, expressive and yet rather simple to use.

If the software architectural constraints defined in the design phase are captured into a tool configuration, and the automatic check is performed often, possibly in the context of a CI/CD system, and starting from the early coding phases, any deviation will be promptly reported with the obvious advantages in terms of development time and costs.

References

[1] ISO/IEC, *ISO/IEC 7498-1:1994: Open Systems Interconnection — Basic Reference Model: The Basic Model*. Geneva, Switzerland: ISO/IEC, Jun. 1996.

[2] J. Foag, "Speculative protocol processing for high-speed packet forwarding," Ph.D. dissertation, Technical University Munich, Germany, 2004.

[3] E. W. Dijkstra, "The structure of the "THE"-multiprogramming system," *Communications of the ACM*, vol. 11, no. 5, pp. 341–346, 1968.

[4] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[5] AUTOSAR, "Layered software architecture," AUTOSAR, Tech. Rep. 53, Nov. 2022, Classic Platform, Release R22-11.

[6] ——, "Explanation of adaptive platform software architecture," AUTOSAR, Tech. Rep. 982, Nov. 2022, Adaptive Platform, Release R22-11.

[7] ISO, *ISO 26262:2018: Road Vehicles — Functional Safety*. Geneva, Switzerland: ISO, Dec. 2018.

[8] ——, *ISO 26262:2018: Road Vehicles — Functional Safety — Part 1: Vocabulary*. Geneva, Switzerland: ISO, Dec. 2018.

[9] ——, *ISO 26262:2018: Road Vehicles — Functional Safety — Part 6: Product development at the software level*. Geneva, Switzerland: ISO, Dec. 2018.

[10] ——, *ISO 26262:2018: Road Vehicles — Functional Safety — Part 9: Automotive safety integrity level (ASIL)-oriented and safety-oriented analyses*. Geneva, Switzerland: ISO, Dec. 2018.

[11] RTCA, SC-205, *DO-178C: Software Considerations in Airborne Systems and Equipment Certification*. RTCA, Dec. 2011.

[12] IEC, *IEC 61508-1:2010: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems — Part 1: General Requirements*. Geneva, Switzerland: IEC, Apr. 2010.

[13] ——, *IEC 61508-1:2010: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems — Part 3: Software Requirements*. Geneva, Switzerland: IEC, Apr. 2010.

[14] ——, *IEC 62304:2006: Medical device software — Software life cycle processes*. Geneva, Switzerland: IEC, May 2006.

[15] ——, *IEC 62304:2006/Amd 1:2015: Medical device software — Software life cycle processes — Amendment 1*. Geneva, Switzerland: IEC, Jun. 2015.

[16] CENELEC, *EN 50128:2011: Railway applications — Communication, signalling and processing systems - Software for railway control and protection systems*. CENELEC, Jun. 2011.

[17] IEC, *IEC 60335-1:2020: Household and Similar Electrical Appliances — Safety — Part 1: General Requirements*. Geneva, Switzerland: IEC, Sep. 2020.

[18] ISO/SAE, *ISO/SAE 21434:2021: Road Vehicles — Cybersecurity Engineering*. Geneva, Switzerland / Warrendale, PA, USA: ISO/SAE, Aug. 2021.

[19] R. Ross, M. Winstead, and M. McEvilley, "Engineering trustworthy secure systems," NIST, Special Publication 800-160v1r1, Nov. 2022.

[20] MISRA, *MISRA-C:2012 — Guidelines for the use of the C language critical systems*. Nuneaton, Warwickshire CV10 0TU, UK: HORIBA MIRA Limited, Feb. 2019, third edition, first revision.

[21] ——, *MISRA C:2012 Amendment 2 — Updates for ISO/IEC 9899:2011 Core functionality*. Nuneaton, Warwickshire CV10 0TU, UK: HORIBA MIRA Limited, Feb. 2020.

[22] ——, *MISRA C:2012 Amendment 3 — Updates for ISO/IEC 9899:2011/2018 Phase 2 — New C11/C18 features*. Norwich, Norfolk NR3 1RU, UK: The MISRA Consortium Limited, Oct. 2022.

[23] E. Gurvitz. (2021, Sep.) A report on Kernel FFI (Freedom From Interference) and some philosophical musings. Enabling Linux In Safety Applications (ELISA) Project, Architecture Working Group. Blog post, last accessed on February 24, 2023. [Online]. Available: https://elisa.tech/blog/2021/09/01/architecture-working-group/

[24] D. Brown, "Solving the software safety paradox," *Embedded Systems Programming*, vol. 11, pp. 44–53, 1998.

[25] P. Koopman. (2014, Sep.) A case study of Toyota unintended acceleration and software safety. Carnegie Mellon University. Presentation slides, last accessed on February 24, 2023. [Online]. Available: https://users.ece.cmu.edu/~koopman/toyota/koopman-09-18-2014_toyota_slides.pdf