Generic Assurance Topics for Any Type of Programmable Content

Rob Ashmore and James Sharp

Dstl, Portsdown West, UK

Abstract

An ever-growing range of technologies can be used to implement programmable content. Examples include highly complex System-On-Chip designs, which use traditional electronics, as well as approaches based on quantum technologies and biological systems. Standards and guidance are required to support the safe use of these, and other, emerging programmable technologies. However, technology-specific guidance is challenging to produce, especially in a timely manner. To help bridge this gap, we propose a set of generic assurance topics, which are applicable to all types of programmable content, introducing considerations based on the assurance of both a program, and the associated substrate. The topics are initially introduced by considering multi-core processors. Their application to alternate technologies is illustrated by considering electronic hardware tailored for machine learning, quantum computing and computation using a bio-based substrate.

1 Introduction

1.1 Motivation

Safety-critical systems are making ever-increasing use of programmable content. This progress has been enabled by safety standards and guidance material. For example, RTCA/DO-178C (RTCA 2011a) addresses software, RTCA/DO-254 (RTCA 2000) addresses Complex Electronic Hardware (CEH) and both RTCA/DO-200 (RTCA 2015) and SCSC-127F (SCSC 2021) address data. Collectively, these examples cover three main aspects of programmability, specifically, software, hardware, and data. Whilst these examples are widely applicable, there is a growing recognition that technology-specific details are important. This is apparent, for example, in RTCA/DO-178C's supplements, e.g. RTCA/DO-331 (RTCA 2011b), which covers model-based development, and documents related to Multi-Core Processors (MCPs), e.g. CAST-32A (CAST 2016).

Programmable content technologies are developing rapidly. The associated need for specific, detailed information inevitably challenges the safety community. We could wait until the technology is well understood and key aspects have been codified as good practice, but this would mean standards significantly lag behind technology development. Another way of addressing the challenge would be to publish good practice rapidly, frequently updating it as new information becomes available. Neither of these approaches

is desirable; standards should not lag technology, nor should they change too often (Johnson 2016).

Consequently, we frame a generic set of topics that an applicant would be expected to discuss with a regulator. The details of those discussions would be specific to the technologies involved in a particular application. Such discussions may be similar to those associated with Certification Review Items (CRIs), which are used in the aviation domain. Equivalently, the topics we identify may be an appropriate structure for CRIs related to programmable content.

Over time, as experience is gained, information may be collected and summarised into a more-traditional form of codified good practice. Although feasible, we recognise this approach has limitations. In particular, both the applicant and regulator need a sufficiently detailed understanding of the relevant technologies: this is not easy to achieve.

1.2 Related Work

The use of a generic set of topics is not a new concept. For example, there are already the "4+1 Software Safety Assurance Principles" (Hawkins et al. 2013). In addition, the Federal Aviation Administration (FAA) and the National Aeronautics and Space Administration (NASA) have described a set of overarching properties (Holloway 2019).

Our work differs from the software safety assurance principles in that it explicitly considers the substrate on which the software is running. In addition, it differs from the overarching properties in that it only considers programmable content, rather than a whole aircraft or an entire system. Consequently, our scope is broader than that of the software safety assurance principles and narrower than that of the overarching properties. Crucially, this choice of scope allows us to direct an appropriate amount of attention to novel types of programmable content.

2 Structure of Our Approach

2.1 Concepts

A program instantiates user-observable behaviour, where a "user" might be, for example, a human or another system component. Although the implementation may be complicated, we suggest that a program's intended behaviour is sufficiently constrained to allow it to be captured in a set of requirements that are meaningful to a user. This is often because a program has been developed to satisfy the needs of a particular user (or group of users). For clarity, we note that our concept of a program encapsulates both software and data.

A substrate provides the physical environment in which the program executes. The same substrate is expected to be able to support many different programs (not necessarily executing simultaneously). Equivalently, from the view of any individual program, the substrate is over-specified; a single program will only use a portion of the substrate's capabilities. In addition, both the physical nature of the substrate and the typically complex interaction between substrate and program result in substrate-level requirements being more detailed and more numerous than user-level program requirements. Consequently, we suggest that, typically, it is not possible for all aspects of a substrate's behaviour to be captured by requirements.

Without a substrate, a program is merely theoretical. Although some claims might be made about a program's behaviour, e.g. number of steps associated with a worst-case execution, these inevitably make assumptions about the substrate. Consequently, safety assurance of a program cannot be completed in isolation. Conversely, without a program, a substrate delivers no user-observable functionality, so safety assurance of a substrate in isolation is, at best, incomplete. Hence, safety assurance arguments related to programmable content are about a program (or a collection of programs) executing on a given substrate. This observation reflects current practice, e.g. RTCA/DO-178C's consideration of the target computer (RTCA 2011a).

To simplify presentation, we often refer to a single program, with a single developer. In almost all cases, there will be multiple programs and multiple developers. For example, in a traditional implementation, the substrate would be some form of processor, whereas the Operating System (OS) and applications would be programs.

The preceding discussion may suggest a clear and obvious separation between program and substrate. As the example of processor microcode illustrates, however, this is not always the case. Fundamentally, for us, the program contains things that, by design, would be expected to vary between programs and hence are under developer control. By that interpretation, microcode would be part of the substrate. Regardless of where the dividing line is drawn for any specific application of a particular technology, it is important that nothing is left unconsidered. This is another reason why, as noted earlier, assurance is about a program executing on a substrate.

2.2 Program Level Assurance

We are concerned with providing assurance that programmable content behaves as expected by the user, which may be a human or another system component. Our focus is on showing that the programmable content meets the expectations of the user, rather than demonstrating that the user's expectations will result in a safe system. Nevertheless, assurance of programmable content behaviour should be a useful claim, within a wider assurance argument, that a system is acceptably safe. As noted previously, a program's intended behaviour is captured in a set of requirements. Consequently, assurance needs to protect against all the following situations (Figure 1):

1. Where there is a difference in understanding of the requirements between the user and the program developer: colloquially, "**requirements are misunderstood**".
2. Where the program's behaviour is a strict subset of the requirements: "**some expected behaviour is not present**".
3. Where the program's behaviour is a strict superset of the requirements: "**some unexpected behaviour is present**".

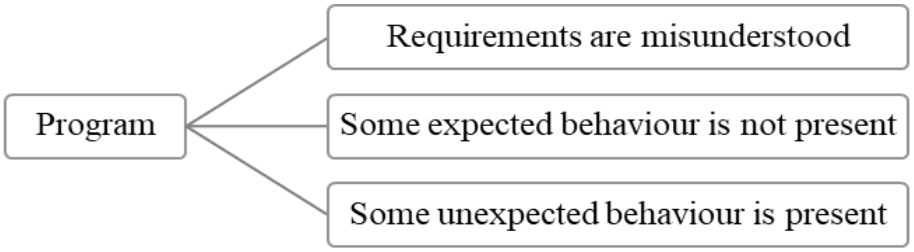

Figure 1 ~ Program Level Situations to Protect Against

It is helpful to reflect these behaviour-related outcomes in the more familiar context of traditional software.

In this case, protection against the first situation (**requirements are misunderstood**) is achieved by ensuring that requirements have particular characteristics, for example, accurate, complete, consistent, unambiguous and verifiable. Adopting a Requirements-Based Testing (RBT) approach, where the program and the tests are independently derived from the requirements, also helps protect against ambiguity in requirements.

RBT also protects against the second situation (**some expected behaviour is not present**). If the program's behaviour is verified against each requirement, then all expected behaviour should be present. This, of course, assumes that the verification methods are sound: control of the test environment and accuracy of tests cases are important considerations here.

Protection against the third situation (**some unexpected behaviour is present**) typically involves measuring code coverage using structural metrics, e.g. branch coverage, as part of an RBT endeavour. This approach is based on an implicit assumption that code structure (rather than, say, data) is the main way that different behaviours are instantiated. This approach also assumes that large-scale behaviour, as expressed in user-level requirements, can be decomposed to low-level behaviour, where coverage is typically measured. Robustness testing, e.g. checking for arithmetic overflow and exceeded frame times, also helps protect against some unexpected behaviour being present.

Formal methods (RTCA 2011c) provide an alternative approach to protecting against our three undesirable outcomes. In this approach, requirements are written in a formal language, guaranteeing an accurate, unambiguous, consistent (but not necessarily complete) description of intended behaviour. This description is refined, often through several stages, to formally demonstrate that the program instantiates the requirements. There have been several successful applications of formal methods; recent examples include a compiler (Leroy et al. 2016) and a microkernel (Klein et al. 2009).

Our three behaviour-related outcomes represent a slightly different way of considering program-level assurance. Although they cover similar themes, e.g. satisfaction of requirements, our outcomes provide an alternative perspective to the "4+1 Software Safety Assurance Principles" (Hawkins et al. 2013). Paradoxically, the widespread success of these principles, which summarise most current approaches to program-level safety, means they are not well-suited for our purpose. We believe they are, in many people's minds, wedded tightly to the development of traditional software, including the explicit, traceable, hierarchical decomposition of requirements. Consequently, they are not ideally suited for considering new types of programmable content (Ashmore and Lennon 2017). That said, our separation of program level and substrate level assurance means that the 4+1 Principles could be used for the former, with our topic areas being adopted for the latter. This might be an attractive approach when the program is largely traditional, with the novelty being in the substrate level.

In addition to the three behavioural outcomes noted above, there is also a need to consider protection against malicious intent. Different technologies may provide protection against certain types of malicious activity: the memory protection offered by the Rust programming language (Balasubramanian et al. 2017) is one example; the CHERI (Capability Hardware Enhanced RISC [Reduced Instruction Set Computer] Instructions) are another (Watson 2019). Most program developments are, however, subject to the same two vulnerabilities. Firstly, the insider threat, which may be partially protected against by review activities. Secondly, vulnerabilities inserted via the tool chain, e.g. (Thompson 2007), (Goodin 2017), and (Peisert et al. 2021), which may be partially (but only partially)

protected against by obtaining professional-quality tools from reputable supply chains. Given the common nature of these vulnerabilities, for reasons of brevity, malicious activity at the program level is not discussed in detail in this paper. Nevertheless, it remains a very important issue.

2.3 Substrate Level Assurance

We adopt an expectation-based approach for substrate-level assurance. In particular, we are concerned with demonstrating that the substrate behaves as expected by the program developer.

As discussed previously, we contend that substrate-level behaviour cannot be meaningfully encapsulated (for a program developer) in a set of requirements. Inevitably, due to the *breadth* and *depth* that these requirements would have to cover, expectations only ever capture a subset.

The *breadth* issue arises because the substrate supports many different types of program. Consequently, an individual program will only use a fraction of the substrate's capability. A program developer will focus on those aspects of the substrate that appear relevant to their specific development. This understandable focusing inevitably produces a partial picture.

The *depth* issue arises because the substrate spans from a physical implementation to the level of abstraction used by the program developer: for traditional processors, the substrate spans from transistor-level properties of electrons to the Instruction Set Architecture (ISA). Across this scope, many aspects of physics can affect behaviour. For traditional electronics, single event upsets are an example that may arise naturally (Taber and Normand 1993), as are manufacturing and age-related issues (Dixit et al. 2021). There are also malicious attacks that exploit predictable outcomes of changes in the substrate's electronic environment, e.g. (Mutlu and Kim 2019) and (Murdock et al. 2020). Encapsulating a complete set of these effects in requirements seems, to us, an insurmountable challenge.

Typically, some form of on-target testing addresses common behaviour. Hence, the question of unexpected behaviour reduces to identification of potential edge cases. Ideally, these should be identified by a process that is rigorous, repeatable, and auditable. This is often achieved through the provision of some form of structure: the guidewords used in a hazard and operability study (HAZOP) are one example (Crawley and Tyler 2015). We propose the structure illustrated in Figure 2.

At the top level, this structure distinguishes between two classes. The first class covers cases where the program developer expects the substrate to behave in a manner that could be feasible but is not exhibited at the relevant time: colloquially, we term this a "**could be, but isn't**" (CBBI) behaviour. The second class covers cases where the developer expects a behaviour that "**could never be**" (CNB) delivered by the substrate. This distinction can be important because the former class (CBBI) can typically be controlled through configuration, whereas the latter class (CNB) cannot.

Note that the notion of expectation used here includes cases where the program developer expects something to happen, as well as cases where the program developer expects something not to happen. As such, it encapsulates situations where expected behaviour is absent, as well as situations where unexpected behaviour is present.

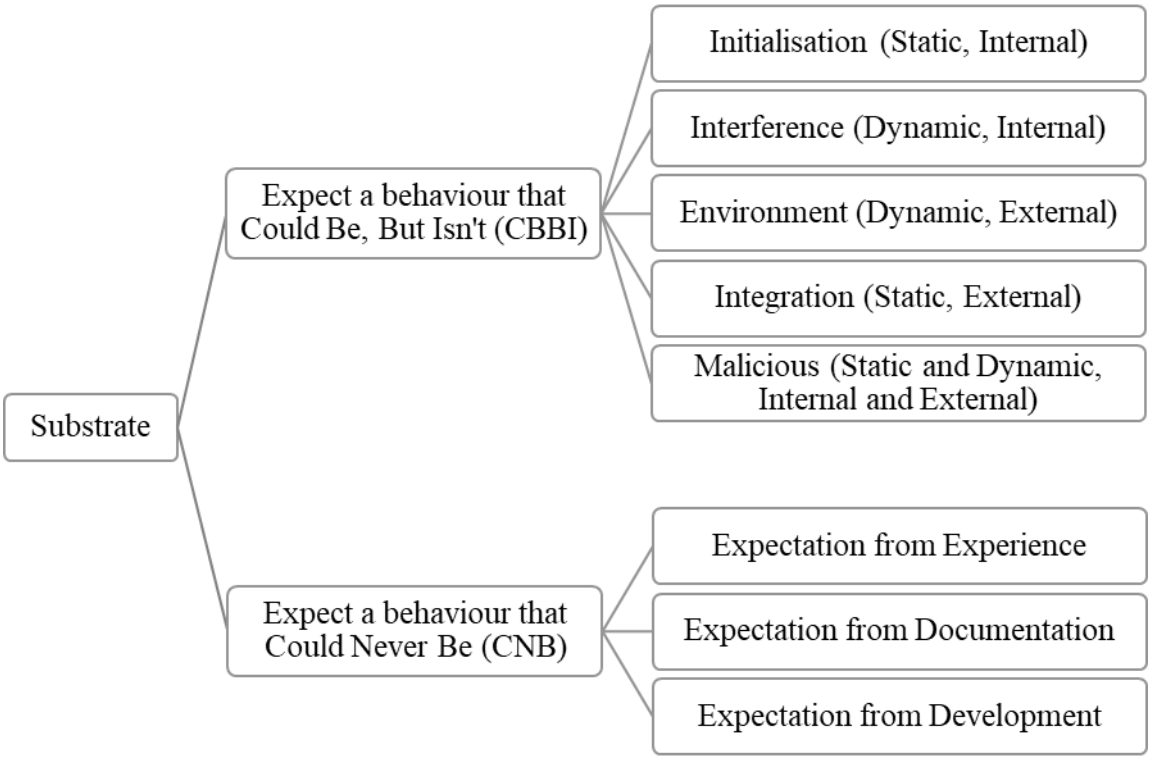

Figure 2 ~ Substrate Level Situations to Protect Against

Unexpected **CBBI** behaviours occur due to some influence on the substrate's behaviour. That influence may be **static,** i.e. remain fixed for the duration of program execution, or it may be **dynamic**. In addition, that influence may be **internal** to the substrate, or it may be **external**. The combination of these properties gives us four ways of identifying unexpected behaviour. These are illustrated in Table 1, with examples from the context of an MCP. These examples are provided solely to help illustrate our approach. As noted previously (in subsection 1.1), there are technology-specific standards for MCP, which are better suited to that specific technology than our more generic approach. Note that, as shown in Table 1, the "static, external" combination is addressed by (system) integration.

Table 1 ~ Examples of CBBI Protections in an MCP Context

	Static	**Dynamic**
Internal	*Initialisation:* Ensuring the MCP configuration settings are as intended.	*Interference:* Mitigating the potential effects of shared resource use.
External	*Integration:* Introduction of a "safety net" (or "safety monitor") external to the MCP.	*Environment:* Protecting against single event upsets, for example, due to cosmic rays.

In addition to the four combinations shown in Table 1, we explicitly consider a fifth item, specifically, **malicious** activity. Examples from the perspective of an MCP are shown in Table 2. This demonstrates that malicious activity can be associated with all four of the previously identified combinations, which may suggest that a specific malicious item is unnecessary. We have considerable sympathy with that view, especially as we strongly favour greater integration between, historically separate, safety and security activities. However, on balance, we believe the explicit security-related focus provided by the

malicious item is beneficial. Note that we explicitly cover malicious activity for the substrate because it differs significantly between substrates. (At the level of detail considered in this paper, there is much more commonality in the types of malicious activity that are associated with the program level.)

Table 2 ~ Examples of Malicious Activity for an MCP

	Static	**Dynamic**
Internal	*Malicious:* Hardware-based spyware that operates continuously.	*Malicious:* Hardware-based Trojan that is triggered by specific conditions.
External	*Malicious:* Undervolting a processor, to break secure enclave protection.	*Malicious:* Generation of external electro-magnetic fields to induce faults.

For **CNB** behaviours, we seek to understand how the program developer may come to expect impossible ("**could never be**") substrate behaviour. We identify three potential sources for the mistaken expectation. Working from a more general perspective to a more specific one, these are:

- Expectations based on previous **experience**. For example, a developer used to working with single core processors may assume exclusive use (within a given time period) of peripherals. This might not be true for an MCP-based implementation.
- Expectations based on substrate **documentation**. These expectations could be mistaken, for example, because of errata or the presence of undocumented features (Domas 2017).
- Expectations based on **development** processes. Differences between the host, i.e. development, and target, i.e. operational, systems are a common source of mistaken expectations.

2.4 Outline Comparisons with Other Standards

We stress that our approach is not meant to replace existing, or emerging, technology-specific standards (or standards-like documents). Where technology-specific information is available, this should provide a more detailed description of relevant assurance topics (and how these may be addressed) than is possible for any generic approach. Technology-specific standards can describe both requirements and associated acceptable means of compliance. In contrast, our generic approach can only outline requirements, in the form of topic areas to be discussed.

As suggested above, our approach is intended to be used when no technology-specific standards are available. Since they satisfy different aims, a detailed comparison of our approach with existing technology-specific standards is not appropriate. Nevertheless, a top-level, outline comparison is considered beneficial, for two reasons. Firstly, it provides an additional description of our approach and how it may be used. Secondly, it provides confidence that the identified topic areas are both necessary and sufficient.

Consequently, our approach has been compared with CAST-32A (CAST 2016), which considers MCP, and the computation-level framework of SCSC-153A (SCSC 2020), which considers artificial intelligence implemented using machine learning techniques.

Details of these outline comparisons are in Appendix A (Table 6 and Table 7, respectively).

The mapping to CAST-32A illustrates that all but one of CAST-32A's objectives can be mapped to our generic topic areas. The exception is that CAST-32A explicitly asks for an accomplishment summary, whereas communication mechanisms are not explicitly detailed in our approach. This mapping also shows that each of our areas maps to at least one CAST-32A objective.

The mapping to the SCSC-153A computation level shows that all of SCSC-153A's objectives can be mapped to our generic topic areas. It also shows that each topic area maps to at least one objective. It is noteworthy that our substrate topic areas, of which there are eight, correspond to only two of the nineteen SCSC-153A objectives. This reflects the greater prominence given to the substrate in our work, and is deemed appropriate, since this is an area of significant technological development.

The simple nature of these mappings means it would be inappropriate to read too much into their results. Nevertheless, they provide some confidence that the topic areas identified in our work are both necessary and sufficient for discussing assurance-related issues associated with new programmable content technologies. Further confidence is provided by the examples discussed in the following section.

3 Example Application

3.1 Example Programmable Content Technologies

To illustrate our topic areas, we apply them to three program/substrate combinations:

- An algorithm developed using Machine Learning (ML), running on a large-scale, complex System-On-Chip (SOC);
- An algorithm running on a quantum computer; and
- An algorithm running on a bio-based substrate.

The first example represents a technology that is available commercially now. The second and third technologies represent items that are being actively researched and where the associated substrates have very different properties to silicon-based electronics. We suggest that this range of examples (together with the previous MCP-related commentary) illustrates the generic nature of our approach. We note, however, that many other types of substrate could have been considered, e.g. neuromorphic processors (Davies et al. 2018), and memory-based compute (Bearden et al. 2020). The large number of potential examples is a key motivation for this paper.

The three examples are considered in the following subsections. Each subsection begins with a brief overview of the program and substrate. This is followed by a discussion of some of the key assurance points. Note that these discussions are not intended to be complete. Their purpose is not to act as a ready-made argument that can be deployed for any use of the associated substrate. Instead, the discussions are intended to demonstrate the way our generic assurance topics highlight key assurance aspects.

3.2 An Algorithm Developed using ML, Running on a SOC

Overview: This example uses Commercial Off-The-Shelf (COTS) hardware in the context of an autonomous vehicle. More specifically, the substrate is an NVIDIA Jetson AGX Xavier SOC (NVIDIA Corporation 2021). This includes: a 512-core Graphical Processing Unit (GPU); an 8-core Central Processing Unit (CPU); two Deep Learning Accelerator (DLA) engines; a Very Long Instruction Word (VLIW) vision processor; Double Data Rate (DDR) 4 memory; and embedded Multi-Media Card (eMMC) storage.

We consider a program implemented as a Deep Neural Network (DNN), which performs object recognition on camera images. This program was developed using a combination of tools, including: the Jetson AGX Xavier Developer Kit; TensorRT; cuDNN; and TensorFlow. Training data was obtained from a fleet of camera-equipped vehicles. The program also includes the Board Support Package (BSP) and the OS, which in this case is Linux for Tegra (L4T).

Our considerations for this example draw heavily on (Ashmore and Sharp 2020), (Ashmore et al. 2021) and (SCSC 2020).

Program Level: For the purposes of this paper, it is assumed that the BSP and OS are largely traditional, so they are not considered further. In practice the BSP and OS are often an important part of the overall COTS package. As such, their influence on the behaviour of the substrate may require a thorough analysis.

ML is typically used to solve open problems, which do not have a complete set of accurate, consistent, and verifiable requirements. Indeed, if such a set of requirements is available then traditional software development techniques may be preferred (Salay and Czarnecki 2018).

When ML is used, protection against **requirements are misunderstood** may be achieved, at least in part, by:

- Ensuring training data is relevant to the object recognition task: for example, a training data set comprising German language road signs would not be relevant for an algorithm deployed on UK roads.
- Ensuring model behaviour is interpretable, including behaviour on a single input (local interpretability) and behaviour on classes of input (global interpretability).
- Ensuring independent verification activities provide suitable coverage of the inputs likely to be received during operational use.

Protection against **some expected behaviour is not present** may be achieved, at least in part, by:

- Ensuring measures of model performance adequately capture required behaviour. For example, some misclassifications may be more important than others: misclassifying a pedestrian as a road marking is likely to be worse than mistaking a 50 mph speed limit sign for a 30 mph limit. Consequently, a model that had similar misclassification rates, regardless of the classes involved, would not be demonstrating the expected behaviour.
- Ensuring test environments are sufficiently representative of the real world. This can be challenging, especially for open problems, where it is difficult to know which parts of reality need to be sufficiently represented.

Protection against **some unexpected behaviour is present** may be achieved, at least in part, by:

- Ensuring training data is suitably complete. For example, covering a sufficiently wide collection of classes, and sub-classes (Paterson and Calinescu 2019), of object types, observed in a sufficiently wide range of meteorological and illumination conditions.
- Ensuring testing on the target hardware covers, either exhaustively or via appropriate sampling, the training, test, and verification sets. Also, ensuring that this testing covers robustness issues, e.g. arithmetic overflow, numerical accuracy.
- Ensuring independent verification activities provide suitable coverage of inputs that may be received following failures elsewhere in the system, e.g. camera issues.

Substrate Level: The substrate is a COTS item. This means the programmable content developer is likely to, at best, have limited influence over its design features. It also means that the developer is likely to have only limited information about the SOC's implementation-level features. (A similar situation prevails for many modern processors, including MCPs, where detailed implementation-level feature information is unlikely to be available.)

From an **initialisation (CBBI - static, internal)** perspective, documenting and justifying the desired SOC configuration is important. This includes, but is not limited to, control of debug features and microcode updates.

From an **interference (CBBI - dynamic, internal)** perspective, runtime monitoring of the SOC's configuration is important (especially for software-configurable items). The potential effects of other software running alongside the program under test may also be important.

From an **environment (CBBI - dynamic, external)** perspective there may be a need to protect against the effect of active sensors on the autonomous vehicle, or other vehicles. For example, the combined effect of multiple, active sensors in a cluttered urban environment may need to be considered.

From an **integration (CBBI - static, external)** perspective, (system) integration activities providing for multiple processing channels, running on separate substrates, is an important consideration. For example, the object recognition channel (the subject of the current discussion) may be combined with a much simpler object detection channel. When multiple channels are used, the degree of independence needs to be carefully considered.

From a **malicious** perspective, the COTS nature of the substrate makes it extremely difficult to protect against the introduction of a hardware-based Trojan. Thinking about the standard cyber security triad of Confidentiality, Integrity and Availability (CIA), availability should be detectable and manageable by traditional safety measures, similar to a hardware failure of the SOC. Multiple processing channels should be used to detect loss of integrity. Confidentiality is very difficult to protect at the substrate-level. System-level architectural designs, which, from a confidentiality perspective, treat the SOC as an "untrusted box" may be an appropriate way of mitigating this risk.

From an **experience (CNB)** perspective, a developer may be unfamiliar with the combination of features available on the SOC. In particular, the combination of CPU, GPU and DLA may mean that the developer's program is executed on an unexpected part of the substrate. This risk may be exacerbated if different SOC workloads result in different execution patterns.

From a **documentation (CNB)** perspective, the possibility of undocumented features and document errata should be considered. The latter may be protected against by careful monitoring of information provided by the substrate manufacturer. Where specific features of the substrate are of particular importance, a "trust but verify" approach may be appropriate.

From a **development (CNB)** perspective, the SOC chosen for this example may be sufficiently powerful to support both development and operational use, thus removing host-target differences. Alternatively, these differences may be important, if development is conducted on a GPU, but operational inference is conducted on a DLA. One way of mitigating this concern would be to conduct all post-training evaluation on the intended target hardware.

3.3 An Algorithm Running on a Quantum Computer

Overview: We begin by noting that, if it is successfully commercialised (Dyakonov 2019), quantum computing is most likely to be applied as a co-processing technology, i.e. alongside traditional computers.

To illustrate the utility of quantum computing, it is helpful to consider different categories of problem (or application). For presentational reasons, we are deliberately imprecise: a more precise description is provided in (Gheorghiu et al. 2019). As shown in Table 3, we identify three separate classes, depending on whether a problem can easily be solved, and whether a solution can easily be checked, using traditional computers.

Table 3 ~ Computation Classes Relevant to Quantum Computation

Using Traditional Computers ...	**Class 1**	**Class 2**	**Class 3**
... Easily Solved?	Yes	No	No
... Easily Checked?	Yes	Yes	No

For problems in Class 1, there is no need to invoke the additional complexity of a quantum computer. For problems in Class 2, provided the implications of a failed check can be handled safely, a significant amount of the assurance burden may be borne by using traditional computing to check a solution and, as such, assurance of the quantum portion may be less of a concern. Hence, problems in Class 1 and Class 2 are excluded from the current discussion.

To the best of our knowledge, the existence of Class 3 problems has not been proven. However, there are problems that credibly can be claimed as being in this class. One example requires calculation of a path that traverses a network (or graph) of a particular form (Childs et al. 2003).

A key aspect of a quantum computer is the notion of a "quantum bit", or qubit. Unlike a traditional bit, which holds a single value (either 0 or 1), a qubit holds a superposition of both 0 and 1; equivalently, a qubit holds a probability distribution over the space *{0, 1}*. Entanglement can be used to link qubits, so that the superposition of n qubits describes a probability distribution over a space containing 2n items. When a measurement is taken, the superposition collapses to a single item. The likelihood of receiving a particular measurement matches the associated probability encoded in the superposition.

Over recent years, there has been significant progress on quantum computing. For example, there is an open-source system, which includes a compiler, simulator and emulator (Steiger et al. 2018). This system includes a backend that links to cloud-based access to quantum computers, provided by IBM (IBM 2021). Other ways of interacting with these quantum computers are also available (IBM n.d.).

A simplified, but indicative, quantum computer system stack, based on (Fu et al. 2016), is illustrated in Table 4. This distinguishes program-level considerations and substrate-level considerations.

Table 4 ~ Simplified Quantum Computer System Stack

Stack	**Program / Substrate**
Quantum Algorithm	Program
Programming Paradigm / Language	
Compiler	
Quantum Instruction Set Architecture	Substrate
Quantum Execution / Error Correction	
Quantum Chip	

Given our current level of understanding, this example focuses on issues to be discussed, rather than on potential solutions to these issues.

Program Level: Our chosen example involves calculating a traversal path for graphs of a certain form. When considering **requirements are misunderstood**, it could be whether the program should calculate any path, or whether the shortest path should be calculated. As highlighted previously, quantum computing is inherently stochastic. Hence, requirements relating to the probability of obtaining the desired result (including whether results from sequential runs are statistically independent) may be misunderstood.

In terms of **some expected behaviour is not present**, test results may be influenced by inappropriate control of the test environment. This may be a particular concern if multiple tests are executed sequentially.

The program being considered is relatively simple in intent and, as such, offers little opportunity for cases where **some unexpected behaviour is present**. One possible example is different classes of network may be much easier to analyse. In some circumstances, significant changes in compute timing, for different classes of input, may be additional and unwanted behaviour.

Substrate Level: Quantum computation is an immature domain. Consequently, a specific example substrate is not discussed. Instead, we consider the quantum substrate in more general terms than that of the ML SOC.

From the perspective of **initialisation (CBBI - static, internal)**, we are concerned with the way that initial qubit values are established, as well as the way that qubit entanglements are created.

In terms of **interference (CBBI - dynamic, internal)**, decoherence and noise are significant challenges to the physical implementation of a quantum computer. These effects make it difficult to maintain specific values in, and entanglement between, qubits. Quantum Error Correction (QEC) methods are used to help mitigate these effects, at the expense of using a larger number of qubits.

With regards to **environment (CBBI - dynamic, external)**, current quantum computers require tight control of the environment, in particular to reduce the amount of environmental noise.

From the perspective of **integration (CBBI - static, external)**, the way that information is passed from a traditional computing resource to the quantum computer is of interest. Additionally, the number of times a quantum algorithm is repeated is another factor. Further, most QEC approaches rely on integration between a control mechanism implemented on a traditional computer and a quantum computer.

Malicious intervention into quantum cryptographic protocols is well-studied (Padmavathi et al. 2016). The potential vulnerabilities of quantum computers appear to be less well investigated. One obvious topic is an effective denial of service attack, caused by increasing the noise in the environment.

Quantum computer programs are based on a very different paradigm to traditional programs. For example, all steps in a quantum program need to be reversible, qubits cannot be copied, and no loops are permitted. (If necessary, loops can be achieved by repeated interaction between traditional and quantum computers, with the former handling the loop construct.) There are also cases where it is appropriate to use an undefined qubit in a calculation. These differences mean a developer may have unrealistic expectations based on previous **experience (CNB)**.

Inaccurate **documentation (CNB)** may create unrealistic expectations. One significant area involves understanding gate-level reliability, which depends on the QEC scheme employed, as well as the quantum computer's tolerance to external noise.

Unrealistic expectations may arise as part of **development (CNB)** due to inaccuracies in simulators and emulators used to debug quantum algorithms. Some of these, especially those relating to the final, pre-measurement superposition distribution, may be difficult to identify.

3.4 An Algorithm Running on a Bio-Based Substrate

Overview: This example considers molecular-based techniques, which exploit Chemical Reaction Networks (CRNs), implemented via Domain Strand Displacement (DSD) (Badelt et al. 2017). A brief introduction to CRNs is available in (Ashmore 2020).

A chemical reaction, in which reactants produce products at a given rate, is expressed in the general form:

$$Reactants \xrightarrow{Rate} Products \qquad \text{(Equation 1)}$$

Both *reactants* and *products* are more generally referred to as *species*. A CRN is a collection of related reactions.

In order to practically implement a CRN, fuel species are typically needed, e.g. to catalyse reactions, and waste species are often produced. Hence, a more complete description is:

$$Reactants + Fuel \xrightarrow{Rate} Products + Waste \qquad \text{(Equation 2)}$$

A number of implementations of CRNs have been demonstrated, including neural networks and solutions to the 3-SAT satisfiability problem (Winfree 2019).

For the purposes of our current discussion, we consider Equation 1 to be at the program level and Equation 2 to be at the substrate level; that level also includes, for example, the physical vessel in which the reactions occur.

Within this section, we assume the CRN is implemented in a bulk, well-mixed system rather than, for example, using a surface-based implementation (which, from our perspective, would be a different type of substrate). Where we need a specific example, we consider a combined CRN watchdog-oscillator system, where the watchdog can reset a failed oscillator (Ellis et al. 2019). However, much of the following focuses on generic CRN-related issues, rather than specific details of this example. This discussion is based on (Ellis et al. 2019), (Lutz et al. 2012) and (Johnson et al. 2019).

Program Level: In terms of **requirements are misunderstood**, CRNs are inherently stochastic, which needs to be appropriately reflected in the requirements. Requirements may also include implicit (and, consequently, misunderstood) assumptions regarding the fate, or future utility, of intermediate products. CRN behaviour is memoryless, stochastic, and asynchronous, so CRNs can be represented as Continuous Time Markov Chains (CTMCs). They are thus amenable to model checking, using tools like PRISM (Kwiatkowska et al. 2011), which can help detect ambiguous requirements.

Model checking can also provide a way of protecting against **some expected behaviour is not present**. CRNs also consume physical resources. In the case of our specific example, this may limit the number of times a broken oscillator may be restarted. Conversely, a user might expect unlimited restarts.

In CRNs, the concentration of a species, i.e. the amount present in the bulk mixture, can be viewed as analogous to a real-valued variable. More strictly, since concentrations cannot be negative, two species are needed to represent a real-valued variable: one for positive values and one for negative values. Including a set of reactions to cover negative values, when a variable can only take positive values, would be an example of **some unexpected behaviour is present** at the program level. Another example of unexpected behaviour could be including reactions to "tidy up" intermediary species, when not specified to do so in the requirements.

Substrate Level: In terms of **initialisation (CBBI - static, internal)**, key issues include the provision of sufficient quantities of reactant and fuel species. In addition, vessels used to contain the CRN implementation should be demonstrably free from contamination.

Concerning **interference (CBBI - dynamic, internal)**, in a bulk mixture all possible reactions occur simultaneously. This can make it challenging to schedule discrete steps within a program. Steps may be approximated with very different reaction rates (earlier steps having faster rates). Additionally, the products of a first step may be the reactants of a second step: this will ensure some of the first step completes before the second step begins; it will not ensure that the first step fully completes before the second step begins. Another approach involves using separate physical containers for each step: this obviously complicates the physical implementation (and integration). Another interference issue is unintended reactions, especially reactions that involve waste species.

From the perspective of the **environment (CBBI - dynamic, external)**, chemical reaction rates are sensitive to environmental conditions, like temperature and pressure. If these are not as intended, then the actual reaction rates may be very different to those anticipated. Additionally, the change in reaction rate may not be uniform across all reactions in a CRN, making sequencing of program steps even more challenging.

In terms of **integration (CBBI - static, external)**, a bio-based approach needs a mechanism for communicating the results of the computation. This is often achieved via molecules with different levels of luminescence. Using this type of approach to "read" the result of the computation is part of integration. For ongoing, continuous programs, e.g. monitors, integration may also involve ensuring there is a sufficient flow of fuel and reactants; removal of waste may also be a factor.

To the best of our knowledge, there has been little work on **malicious** interference with CRNs. Consequently, we merely highlight two theoretical possibilities. Firstly, we note that additional species could be inserted so that the program's behaviour changes after a certain number of oscillations (or, perhaps, a certain number of oscillator resets). Secondly, we note that the concentration of waste species could provide an adversary with valuable information, potentially undermining confidentiality.

From an **experience (CNB)** perspective, most programmers used to traditional electronics may be challenged by the notion that all possible reactions can occur simultaneously. The significant time taken for processing to complete (typically measured in hours) may be another relevant factor.

With regards to **documentation (CNB)**, the mapping from an abstract CRN, e.g. Equation 1, to an implementation CRN, e.g. Equation 2, means that, in essence, each substrate is program-specific. In traditional terms, the substrate is more like an Application-Specific Integrated Circuit (ASIC) than a general-purpose CPU. Hence, documentation relates to the way the relevant species are created and their resulting properties, e.g. amount of species fragments, or other unintended items.

In terms of **development (CNB)**, there are two main ways of analysing bulk mixtures. CTMCs account for stochastic behaviour and are amenable to model checking. Ordinary Differential Equations (ODEs) represent expected, average behaviour. This combination of methods may help protect against incorrect expectations. In addition, bisimulation approaches can provide confidence that an implementation CRN accurately reflects the intent of an abstract CRN (Johnson et al. 2019).

4 Summary

Table 5, overleaf, provides a very brief summary of the main points raised in the preceding examples. Depending on the nature of the example, these points may be mechanisms for providing assurance evidence, or issues that should be addressed by assurance evidence. For completeness, this table also includes a line on malicious activity at the program level.

The table demonstrates the potential applicability of our generic assurance topics to a wide range of programmable content. Whilst this is encouraging, more work is needed before these topics, broken down into program and application substrate, can become a formal recommendation. Completion of a number of specific worked examples, with sufficient detail to provide compelling assurance arguments, are an important part of that work.

Finally, motivated by the creation of specialised SOCs for ML tasks, and the broadening of computation away from traditional transistor-based architectures, we have highlighted the need for a set of generic assurance topics. These should enable a consistent approach to assuring elements within the ever-broadening domain of programmable content.

Table 5 ~ Summary of Generic Assurance Topics and Chosen Examples

		Developed Using ML, Running on a SOC	**Algorithm Running on a Quantum Computer**	**Algorithm Running on a Bio-Based Substrate**
Program	**Requirements are misunderstood**	Relevant training data; interpretable behaviour; independent verification	Requirements relating to probability of correct result; independence of sequential runs	Reflect stochastic nature; intermediate products; probabilistic model checking
	Some expected behaviour is not present	Adequate measures of model performance; sufficiently representative test environment	Control of the test environment, e.g. sequential tests	Probabilistic model checking; consumption of physical resources
	Some unnecessary behaviour is present	Sufficient testing (including robustness); independent verification	Significant, input-dependent changes in timing	Inclusion of species for negative values; tidy up of intermediate species
	Malicious activity	Insider threat; tool-based vulnerabilities	Insider threat; tool-based vulnerabilities	Insider threat; tool-based vulnerabilities

		Developed Using ML, Running on a SOC	Algorithm Running on a Quantum Computer	Algorithm Running on a Bio-Based Substrate
Substrate	**Initialisation (CBBI - static, internal)**	Document and justify SOC configuration	Initial qubit values and entanglements	Sufficient quantities of species; vessels free from contamination
	Interference (CBBI - dynamic, internal)	Runtime monitoring; effects of co-hosted software	Decoherence; noise; QEC	Sequencing steps; unintended reactions
	Environment (CBBI - dynamic, external)	Active sensors on own, and other vehicles	Noise	Effects on reaction rates
	Integration (CBBI - static, external)	Multiple processing channels	Passing information from traditional to quantum; number of re-runs	Reading the results; input flow of species; removal of waste
	Malicious activity	Hardware-based Trojans; multiple processing channels; untrusted "closed box" architecture	Denial of service by increasing environment noise	Insertion of additional species; concentration of waste species
	Experience (CNB)	Combination of processing features	Reversible; no qubit copying; no loops	Simultaneous reactions; long processing time
	Documentation (CNB)	Undocumented features; document errata; trust but verify	Gate-level reliability; QEC scheme	Way species are created
	Development (CNB)	Host-target differences; train on target	Inaccuracies in simulators and emulators	CTMCs; ODEs; bisimulation

Disclaimers

Acknowledgments

The authors gratefully acknowledge the comments provided by the anonymous reviewers. These led to a significantly improved paper.

References

Ashmore, R. (2020). Urban Maths - Best of Both Worlds. *Mathematics Today, June 2020.* Retrieved from: https://ima.org.uk/14267/urban-maths-the-best-of-both-worlds/. Accessed 14th December 2021.

Ashmore, R., & Lennon, E. (2017). Progress Towards the Assurance of Non-traditional Software. In Parsons, M., & Kelly, T. (Eds.) *Developments in System Safety Engineering, Proceedings of the 25th Safety-critical Systems Symposium, February 2017*. Independently Published.

Ashmore, R., & Sharp, J. (2020). Assurance Argument Elements for Off-the-shelf, Complex Computational Hardware. In Casimiro, A., Ortmeier, F., Bitsch, F., & Ferreira, P. (Eds.). *Computer Safety, Reliability, and Security: 39th International Conference, SAFECOMP 2020, Lisbon, Portugal, September 16–18, 2020, Proceedings* (pp. 260-269). Springer.

Ashmore, R., Calinescu, R., & Paterson, C. (2021). Assuring the Machine Learning lifecycle: Desiderata, methods, and challenges. *ACM Computing Surveys (CSUR), 54(5),* 1-39. https://doi.org/10.1145/3453444

Badelt, S., Shin, S. W., Johnson, R. F., Dong, Q., Thachuk, C., & Winfree, E. (2017). A General-Purpose CRN-to-DSD Compiler with Formal Verification, Optimization, and Simulation Capabilities. In Brijder, R., & Qian, L. (Eds). *DNA Computing and Molecular Programming: 23rd International Conference, DNA 23, Austin, TX, USA, September 24–28, 2017* (pp. 232-248). Springer, Cham, Switzerland. https://doi.org/10.1007/978-3-319-66799-7_15

Balasubramanian, A., Baranowski, M.S., Burtsev, A., Panda, A., Rakamarić, Z., & Ryzhyk, L. (2017). System Programming in Rust: Beyond Safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems, May 2017* (pp. 156-161). https://doi.org/10.1145/3102980.3103006

Bearden, S.R., Pei, Y.R., & Di Ventra, M. (2020) Efficient solution of Boolean satisfiability problems with digital memcomputing. *Scientific Reports, 10(1),* 1-8. Retrieved from: https://www.nature.com/articles/s41598-020-76666-2 Accessed 20th January 2022.

CAST, the Certification Authorities Software Team (2016). *Multi-core Processors.* Position Paper CAST-32A, November 2016

Childs, A.M., Cleve, R., Deotto, E., Farhi, E., Gutmann, S., & Spielman, D. A. (2003). Exponential algorithmic speedup by a quantum walk. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing, June 2003* (pp. 59-68) https://doi.org/10.1145/780542.780552

Crawley, F., & Tyler, B. (2015). *HAZOP: Guide to best practice*. Elsevier, 3rd edition (21 April 2015)

Davies, M., Srinivasa, N., Lin, T. H., Chinya, G., Cao, Y., Choday, S.H., Dimou, G., Joshi, P., Imam, N., Jain, S., & Liao, Y. (2018). Loihi: A Neuromorphic Manycore Processor with On-Chip Learning. *IEEE Micro, 38(1), January 2018,* 82-99 https://doi.org/10.1109/MM.2018.112130359

Dixit, H.D., Pendharkar, S., Beadon, M., Mason, C., Chakravarthy, T., Muthiah, B., & Sankar, S. (2021) *Silent Data Corruptions at Scale.* arXiv. Retrieved from: https://arxiv.org/pdf/2102.11245v1.pdf Accessed 20th January 2022.

Domas, C. (2017*). Breaking the x86 ISA.* Black Hat, USA. Retrieved from: https://www.blackhat.com/docs/us-17/thursday/us-17-Domas-Breaking-The-x86-Instruction-Set-wp.pdf Accessed 20th January 2022.

Dyakonov, M. (2019). When will useful quantum computers be constructed? Not in the foreseeable future, this physicist argues. Here's why: The case against: Quantum computing. *IEEE Spectrum, 56(3), March 2019,* (pp 24–29) https://doi.org/10.1109/MSPEC.2019.8651931

Ellis, S.J., Klinge, T.H., Lathrop, J.I., Lutz, J.H., Lutz, R.R., Miner, A.S., & Potter H.D. (2019). Runtime Fault Detection in Programmed Molecular Systems. *ACM Transactions on Software Engineering and Methodology (TOSEM), 28(2) April 2019, Article No.: 6,* (pp 1–20) https://doi.org/10.1145/3295740

Fu, X., Riesebos, L., Lao, L., Almudever, C.G., Sebastiano, F., Versluis, R., Charbon, E., Bertels, K. (2016). A heterogeneous quantum computer architecture. In *Proceedings of the ACM International Conference on Computing Frontiers, May 2016,* (pp 323–330) https://doi.org/10.1145/2903150.2906827

Gheorghiu, A., Kapourniotis, T., Kashefi, E. (2019). Verification of quantum computation: An overview of existing approaches. *Theory of Computing Systems, 63(4), May 2019* (pp.715-808) https://doi.org/10.1007/s00224-018-9872-3

Goodin, D. (2017). Apple scrambles after 40 malicious "XcodeGhost" apps haunt App Store. *Ars Technica.* Retrieved from: https://arstechnica.com/information-technology/2015/09/apple-scrambles-after-40-malicious-xcodeghost-apps-haunt-app-store/ Accessed 20th January2022.

Hawkins, R., Habli, I., & Kelly, T. (2013). The principles of software safety assurance. In *31st International System Safety Conference, 2013*. Retrieved from: https://www-users.cs.york.ac.uk/rhawkins/papers/HawkinsISSC13.pdf Accessed 20th January 2022.

Holloway, C. M. (2019). *Understanding the overarching properties.* NASA Technical Memorandum NASA/TM-2019-220292. Retrieved from: https://ntrs.nasa.gov/api/citations/20190029284/downloads/NASA-TM-2019-220292Replacement.pdf Accessed 20th January 2022.

IBM. (2021) *IBM Quantum Experience.* Retrieved from: http://research.ibm.com/quantum/, Accessed 14th December 2021.

IBM. (n.d.) *Qiskit: Open-Source Quantum Development*. Retrieved from: https://qiskit.org/. Accessed 14th December 2021.

Johnson, C. (2016). Role of Regulators in Safeguarding the Interface between Autonomous Systems and the General Public. In Hewett, J. (Ed.). *Proceedings of the 34th International System Safety Conference, Orlando, USA 8-12 August 2016* Retrieved from: http://www.dcs.gla.ac.uk/~johnson/papers/ISSC16/regulator.pdf Accessed 20th January 2022.

Johnson, R., Dong, Q., & Winfree, E. (2019). Verifying chemical reaction network implementations: a bisimulation approach. *Theoretical Computer Science, 765,* (pp.3-46). Retrieved from: https://authors.library.caltech.edu/96464/1/1-s2.0-S0304397518300136-main.pdf Accessed 20th January 2022.

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., & Sewell, T. (2009). seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, October 2009,* (pp. 207-220) https://doi.org/10.1145/1629575.1629596

Kwiatkowska, M., Norman, G., & Parker, D. (2011). PRISM 4.0: Verification of probabilistic real-time systems. In Gopalakrishnan, G., & Qadeer, S. *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. (pp. 585-591). Springer. Berlin, Heidelberg, Germany.

Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., & Ferdinand, C. (2016). CompCert-a formally verified optimizing compiler. In *Proceeding of the 8th European Congress on Embedded Real Time Software and Systems, ERTS 2016, Toulouse, France* Retrieved from: https://www.researchgate.net/publication/293814383_CompCert_-_A_Formally_Verified_Optimizing_Compiler Accessed 20th January 2022.

Lutz, R. R., Lutz, J. H., Lathrop, J. I., Klinge, T. H., Mathur, D., Stull, D. M., Bergquist, T. G., & Henderson, E. R. (2012). Requirements analysis for a product family of DNA nanodevices. In *2012 20th IEEE International Requirements Engineering Conference (RE)* (pp. 211-220). https://doi.ieeecomputersociety.org/10.1109/RE.2012.6345806

Murdock, K., Oswald, D., Garcia, F.D., Van Bulck, J., Gruss, D., & Piessens, F. (2020). Plundervolt: Software-based fault injection attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy (SP)* (pp. 1466-1482). https://doi.ieeecomputersociety.org/10.1109/SP40000.2020.00057

Mutlu, O., & Kim, J. S. (2019). RowHammer: A retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 39(8), August 2020.* 1555-1571 https://doi.org/10.1109/TCAD.2019.2915318

NVIDIA Corporation. (2021). *Jetson AGX Xavier Developer Kit.* Retrieved from: https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit. Accessed 14th December 2021.

Padmavathi, V., Vardhan, B. V., & Krishna, A. V. N. (2016). Quantum cryptography and quantum key distribution protocols: a survey. In *2016 IEEE 6th International Conference on Advanced Computing (IACC)* (pp. 556-562). https://doi.org/10.1109/IACC.2016.109

Paterson, C., & Calinescu, R. (2019). *Detection and mitigation of rare subclasses in neural network classifiers.* arXiv preprint arXiv:1911.12780, 2019. Retrieved from: https://www.researchgate.net/profile/Colin-Paterson-4/publication/337671290_Detection_and_Mitigation_of_Rare_Subclasses_in_Neural_Network_Classifiers/links/5df770984585159aa4809742/Detection-and-Mitigation-of-Rare-Subclasses-in-Neural-Network-Classifiers.pdf Accessed 20th January 2022.

Peisert, S., Schneier, B., Okhravi, H., Massacci, F., Benzel, T., Landwehr, C., Mannan, M., Mirkovic, J., Prakash, A., & Michael, J. B. (2021). Perspectives on the SolarWinds Incident. *IEEE Security & Privacy, 19(02),* (pp. 7-13). Retrieved from: https://www.computer.org/csdl/magazine/sp/2021/02/09382367/1saZVPHhZew Accessed 20th January 2022.

RTCA. (2000). *Design Assurance Guidance for Electronic Hardware*. RTCA/DO-254, RTCA, Inc. Also available as EUROCAE Document ED-80.

RTCA. (2011a). *Software Considerations in Airborne Systems and Equipment Certification*. RTCA/DO-178C, RTCA, Inc. Also available as EUROCAE Document ED-12C.

RTCA. (2011b). *Model-Based Development Supplement*. RTCA/DO-331, RTCA, Inc. Also available as EUROCAE Document ED-218.

RTCA. (2011c). *Formal Methods Supplement to DO-178C and DO-278A*. RTCA/DO-333, RTCA, Inc. Also available as EUROCAE Document ED-216.

RTCA. (2015). *Standards for Processing Aeronautical Data*. RTCA/DO-200, RTCA, Inc. Also available as EUROCAE Document ED-76.

Salay, R., & Czarnecki, K. (2018). *Using machine learning safely in automotive software: An assessment and adaption of software process requirements in ISO 26262*. arXiv preprint arXiv:1808.01614, 2018 Retrieved from: https://arxiv.org/ftp/arxiv/papers/1808/1808.01614.pdf Accessed 20th January 2022.

SCSC, Safety Critical Systems Club C.I.C. (2020). *Safety assurance objectives for autonomous systems*, SCSC-153A. Retrieved from: https://scsc.uk/r153A:1 Accessed 20th January 2022.

SCSC, Safety Critical Systems Club C.I.C. (2021). *Data Safety Guidance*. SCSC-127F. Retrieved from: https://scsc.uk/r127F:1 Accessed 20th January 2022.

Steiger, D. S., Häner, T., & Troyer, M. (2018). ProjectQ: an open source software framework for quantum computing. *Quantum*, *2*, p.49 https://doi.org/10.22331/q-2018-01-31-49

Taber, A., & Normand, E. (1993). Single event upset in avionics. *IEEE Transactions on Nuclear Science, vol. 40, no. 2, pp. 120-126, April 1993,* https://doi.org/10.1109/23.212327

Thompson, K. (2007). Reflections on trusting trust. In *ACM Turing Award Lectures* (p. 1983), January 2007. Association for Computing Machinery. New York, USA https://doi.org/10.1145/1283920.1283940

Watson, R. N. M., Moore, S. W., Sewell, P., & Neumann, P. G. (2019). *An introduction to CHERI*. University of Cambridge, Computer Laboratory, Technical Report Number 941, UCAM-CL-TR-941, ISSN 1476-2986

Winfree, E. (2019). Chemical reaction networks and stochastic local search. In *International Conference on DNA Computing and Molecular Programming* (pp. 1-20). Springer. Cham Switzerland.

Appendix A. Outline Comparison Mappings

Table 6, below, provides an outline mapping between the objectives listed in CAST-32A and the generic topic areas identified in this paper. Table 7 provides a similar mapping between the objectives in the computation-level framework of SCSC-153A and the topic areas of this paper.

Table 6 ~ Outline Mapping from CAST-32A Objectives to Generic Topic Areas

CAST-32A Objective	**Program**			**Substrate**								**Count**
	Requirements are misunderstood	Some expected be-haviour is not present	Some unexpected behaviour is present	CBBI Initialisation (Static, Internal)	CBBI Interference (Dynamic, Internal)	CBBI Environment (Dynamic, External)	CBBI Integration (Static, External)	CBBI Malicious	CNB Expectation from Experience	CNB Expectation from Documentation	CNB Expectation from Development	
MCP_Planning_1				X	X		X			X	X	**5**
MCP_Resource_ Usage_1				X	X							**2**
MCP_Resource_ Usage_2					X	X		X				**3**
MCP_Planning_2				X	X		X			X		**4**
MCP_Resource_ Usage_3					X					X		**2**
MCP_Resource_ Usage_4					X		X					**2**
MCP_Software_1	X	X	X		X							**4**
MCP_Software_2	X	X	X		X				X	X		**6**
MCP_Error_ Handling_1							X	X				**2**
MCP_Accomplishment_ Summary_1												**0**
Count	**2**	**2**	**2**	**3**	**8**	**1**	**4**	**2**	**1**	**4**	**1**	**-**

Table 7 ~ Outline Mapping from SCSC-153A Computation-Level Objectives to Generic Topic Areas

SCSC-153A Computation-Level Objective	Program			Substrate								Count
	Requirements are misunderstood	Some expected behaviour is not present	Some unexpected behaviour is present	CBBI Initialisation (Static, Internal)	CBBI Interference (Dynamic, Internal)	CBBI Environment (Dynamic, External)	CBBI Integration (Static, External)	CBBI Malicious	CNB Expectation from Experience	CNB Expectation from Documentation	CNB Expectation from Development	
COM 1-1: Data is acquired and controlled appropriately.	X											**1**
COM1-2: Pre-processing methods do not introduce errors.		X	X									**2**
COM1-3: Data captures the required algorithm behaviour.		X										**1**
COM1-4: Adverse effects arising from distribution shift are protected against.			X									**1**
COM2-1: Functional requirements imposed on the algorithm are defined and satisfied.	X	X	X									**3**
COM2-2: Non-functional requirements imposed on the algorithm are defined and satisfied.	X	X	X									**3**
COM2-3: Algorithm performance is measured objectively.		X										**1**
COM2-4: Performance boundaries are established and complied with.		X										**1**
COM2-5: The algorithm is verified with an appropriate level of coverage.		X	X									**2**
COM2-6: The test environment is appropriate.		X										**1**
COM2-7: Each algorithm variant is tested appropriately.		X	X									**2**

SCSC-153A Computation-Level Objective	Program			Substrate								Count
	Requirements are misunderstood	Some expected behaviour is not present	Some unexpected behaviour is present	CBBI Initialisation (Static, Internal)	CBBI Interference (Dynamic, Internal)	CBBI Environment (Dynamic, External)	CBBI Integration (Static, External)	CBBI Malicious	CNB Expectation from Experience	CNB Expectation from Documentation	CNB Expectation from Development	
COM3-1: An appropriate algorithm type is used.		X	X									**2**
COM3-2: Typical errors are identified and protected against.			X									**1**
COM3-3: The algorithm's behaviour is explainable.	X	X	X									**3**
COM3-4: Post-incident analysis is supported.		X	X									**2**
COM4-1: The software is developed and maintained using appropriate standards.	X	X	X									**3**
COM4-2: Software misbehaviour does not result in incorrect outputs from the algorithm.			X									**1**
COM5-1: Appropriate computational hardware standards are employed.				X	X	X	X	X	X	X	X	**8**
COM5-2: Hardware misbehaviour does not result in incorrect outputs from the algorithm.					X	X		X				**3**
Count	**5**	**13**	**12**	**1**	**2**	**2**	**1**	**2**	**1**	**1**	**1**	**-**