Pattern-Based Logical Isolation for Safety-Critical Multicore Systems

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

von der KIT-Fakultät für
Elektrotechnik und Informationstechnik
des Karlsruher Instituts für Technologie (KIT)
angenommene

Dissertation

von

Tobias Konstantin Dörr, M.Sc.

Tag der mündlichen Prüfung: 9. September 2024

Hauptreferent: Prof. Dr.-Ing. Dr. h. c. Jürgen Becker
Korreferent: Prof. Dr.-Ing. habil. Mario Trapp

1st edition, October 2024
Karlsruhe, Germany

https://doi.org/10.5445/IR/1000175388

Pattern-Based Logical Isolation for Safety-Critical Multicore Systems

Tobias Dörr

Zusammenfassung

Computerplattformen für eingebettete Systeme zeichnen sich durch stetig wachsende Performanz und zunehmende Heterogenität aus. Unter Verwendung solcher Plattformen ist es möglich, bislang getrennt realisierte Anwendungen auf ein und demselben Chip zu integrieren. Eine solche Integration bietet zahlreiche Vorteile, stellt aus Sicht der funktionalen Sicherheit allerdings auch eine massive Herausforderung dar. Im sicherheitskritischen Umfeld kann das Fehlverhalten bestimmter Anwendungen katastrophale Folgen haben. Zur Umsetzung der On-Chip-Integration ist es daher häufig unumgänglich, kritische von weniger kritischen Anwendungen zu isolieren.

Die Literatur kennt eine Vielzahl von Ansätzen zur Realisierung räumlicher Isolation in Multicore-Systemen. So sind Plattformen, die ein Multiprocessor System-on-Chip (MPSoC) bilden, immer häufiger durch Zugriffsschutzeinheiten im geteilten Interconnect ausgestattet. Diese sogenannten Access Protection Units (APUs) sind konfigurierbar und setzen eine Partitionierung relevanter On-Chip-Ressourcen um. Anders als der Hypervisor oder das Betriebssystem, welche zur Trennung von Anwendungen auf einem bestimmten Prozessor zum Einsatz kommen, sind APUs für die Isolation zwischen *verschiedenen* Prozessoren bestimmt. In der Praxis ist das Aufstellen korrekter APU-Konfigurationen jedoch oft problematisch. Insbesondere wenn mehrere MPSoCs miteinander verbunden werden, erfordert der Konfigurationsprozess detailliertes Wissen über alle beteiligten Plattformen und die auf den Plattformen ausgeführte Software. Derzeit verfügbare Entwurfsmethoden berücksichtigen solche Aspekte nicht oder nur eingeschränkt.

In dieser Arbeit werden APUs als Mechanismen zur Erfüllung funktionaler Sicherheitsanforderungen betrachtet. Basierend auf dem Konzept des kaskadierenden Fehlers (CF) gemäß ISO 26262 wird dafür zunächst die Abstraktion der *logischen Isolation* eingeführt; sie beschreibt, wie APU-Konfigurationen und ergänzende Techniken zur Einschränkung von CFs beitragen. Unter Ver-

wendung dieser Abstraktion wird eine neue Methodik für die strukturierte Konfiguration von APUs auf einem oder mehreren MPSoCs vorgestellt.

Die vorgestellte Methodik setzt eine modellbasierte Formalisierung der angedachten Hardware-, Laufzeit- und Softwarearchitektur voraus. Sie generiert APU-Konfigurationscode und leitet das CF-Potential ab, das trotz Anwendung dieses Konfigurationscodes auf den verschiedenen Systemschichten verbleibt. CF-Potential, das von fehleranfälligen Systemelementen zur Anwendungslogik führt (und so physischen Schaden hervorrufen kann), wird automatisch bestimmt und unter Berücksichtigung relevanter Sicherheitsanforderungen auf inakzeptables Risiko geprüft. In Form eines flexiblen „Patterns" sind alle diese Schritte automatisiert und auf verschiedene MPSoCs aus einer erweiterbaren Plattformbibliothek anwendbar.

Um die praktische Realisierbarkeit des Patterns zu demonstrieren, stellt diese Arbeit eine vollständige Implementierung vor. Die Plattformbibliothek wird dabei mit Modellen zweier kommerzieller MPSoCs befüllt: des i.MX 8M von NXP sowie des Zynq UltraScale+ MPSoC von AMD/Xilinx. In Form einer Fallstudie kommt das Pattern zum Einsatz, um APU-Konfigurationscode für die Verwendung des Ethernet-Controllers auf dem Zynq UltraScale+ MPSoC zu generieren und Aussagen über ausgewählte Sicherheitseigenschaften zu treffen. Eine zweite Fallstudie untersucht, inwiefern das Patterns auf eine bekannte sowie eine eigens entwickelte Architektur für Fail-Operational-Systeme anwendbar ist. Sie zeigt, dass das Pattern auch dann zur Argumentation über inakzeptable Risiken einsetzbar ist, wenn kein APU-Konfigurationscode erzeugt wird und APUs daher ungenutzt bleiben.

Die Pattern-Implementierung wird durch verschiedene Entwurfs- und Analysewerkzeuge ergänzt. Dazu gehört ein Optimierungsansatz auf Basis des Integer Linear Programming (ILP), der zur Minimierung oder Maximierung bestimmter Gütekriterien dient. Dieser kann beispielsweise eingesetzt werden, um die Anzahl der APUs zu reduzieren, die zur Realisierung eines sicheren Mixed-Criticality-Systems konfiguriert werden müssen.

Das Pattern als flexible, eigenständige Komponente wird abschließend in die akademische XANDAR-Toolchain zur Synthese eingebetteter Softwaresysteme integriert. Diese prototypische Integration schließt die Arbeit ab und dient als Validierung des zugrundeliegenden Systemmodells.

Abstract

Embedded computing platforms are characterized by steadily growing performance and increasing heterogeneity. Using such platforms, it is possible to perform an on-chip integration of applications that were previously implemented on separate hardware. This integration offers numerous benefits, but it also poses a major functional safety challenge. In safety-critical environments, failures of critical applications can be catastrophic, and isolating such applications from less critical ones is often essential.

Literature describes various approaches for spatial isolation in multicore systems. In the context of Multiprocessor System-on-Chip (MPSoC) devices, one such approach is the use of hardware units that operate at the on-chip interconnect level and can be configured to partition relevant platform resources. Today, such Access Protection Units (APUs) are available in most off-the-shelf MPSoCs. Unlike hypervisors and operating systems, which are used to separate applications on a specific processor, APUs are meant for the isolation between *different* processors. In practice, however, finding correct APU configurations is often problematic. Especially when multiple MPSoCs are interconnected, detailed knowledge about all involved platforms and the software executed on them is required. Currently available design methods do not fully account for these aspects.

In this work, APUs are treated as mechanisms to meet functional safety requirements. Based on the Cascading Failure (CF) concept from ISO 26262, the *logical isolation* abstraction is introduced to describe how APU configurations and complementary mechanisms constrain CF potential. Building upon this abstraction, a novel design methodology for the structured configuration of APUs in a single or a network of MPSoCs is presented.

The presented design methodology relies on a model-based formalization of the envisaged hardware, runtime, and software architecture. It generates APU configuration code and derives remaining CF potential across all system layers.

CF potential that leads from error-prone system elements to the application logic (and may therefore cause physical harm) is automatically determined and, considering user-provided safety requirements, checked for unreasonable risk. In the form of a flexible *pattern*, these steps are fully automated and applicable to various MPSoCs from an extensible platform library.

By presenting a full reference implementation of the pattern, this thesis demonstrates the feasibility of the approach. As a real-world example, models of two off-the-shelf MPSoCs are added to the platform library: the i.MX 8M by NXP and the Zynq UltraScale+ MPSoC by AMD/Xilinx. In the form of a case study, the pattern is applied to generate APU configuration code for Ethernet controller usage on the Zynq UltraScale+ MPSoC and to reason about selected safety properties of the created design. A second case study considers the pattern's applicability to one state-of-the-art and one custom architecture for fail-operational systems. It shows that the methodology can also be used to argue about unreasonable risk if no APU configurations are generated and APUs therefore remain unused.

The pattern implementation is supplemented by various design and analysis tools. This includes an optimization approach based on Integer Linear Programming (ILP), which minimizes or maximizes certain figures of merit. It can be applied, for example, to minimize the number of APUs that need to be configured to construct a safe mixed-criticality system.

The pattern, as a flexible standalone component, is finally integrated into the academic XANDAR toolchain for embedded software system synthesis. This prototypical integration concludes the work and serves as a validation of the underlying system model.

Preface

This work originates from my time as a researcher at the Karlsruhe Institute of Technology (KIT). During this time, I had the opportunity to contribute to various projects on the functional safety of embedded software systems.

In this field, the state of the art offers many standard solutions for common design problems. Fault tree construction and software partitioning through hypervisors are two of the various techniques used to build safe systems.

With respect to the safe on-chip integration of different applications, however, I perceived the set of available standard solutions as too limited. Especially in the context of MPSoCs, I noticed that the manual execution of numerous design steps was necessary to achieve a sufficient on-chip isolation.

The desire to structure and automate this process is what motivated me to write this thesis. First and foremost, I perceive the resulting work as a successful feasibility study. It demonstrates that in the design of safety-critical multicore systems, it is possible to achieve a much higher degree of automation than what is available in current state-of-the-art solutions. On a more abstract level, I also like to think of the work as a novel and more structured way of reasoning about failure propagation in embedded software systems.

None of these results would have been possible without the continuous support of my advisor, Prof. Jürgen Becker from KIT. By integrating me into an inspiring project landscape, placing a lot of trust in me early on, and providing me with valuable feedback at any time, he played a key role in the success of this work. For this, I owe him a great debt of gratitude.

I would also like to thank Prof. Mario Trapp from the Technical University of Munich (TUM) for overseeing my thesis as the second examiner. His contributions to the area of safety were a great help during the finalization of this work, and I am grateful for the excellent cooperation over the past months.

All colleagues, project partners, and co-authors with whom I have worked over the years, I would like to thank for the pleasant collaboration. I was

always able to rely on their help and appreciate the fruitful discussions we had. I would also like to thank the students who I was able to supervise as part of their bachelor's and master's theses. For me, each supervision was a very valuable opportunity to broaden my horizons.

While completing this thesis and preparing for the exam, I was very fortunate to receive help from many people—whether by proofreading individual passages, providing feedback on the content, or simply giving their opinion on which variant of a figure they preferred. This help was extremely valuable to me, and I am very grateful to have received it.

Last but not least, special thanks go to my family and my friends. Without their support, this work would not exist.

Tobias Dörr
October 2024

Table of contents

List of figures

List of tables

List of abbreviations

Chapter 1

Introduction

A safety-critical system is one whose failure can lead to physical harm, for example in the form of serious injury or significant property damage [15].

In various domains, the design of safety-critical systems is therefore subject to legal or normative requirements. ISO 26262 [16], for example, is a safety standard concerned with the Electrical/Electronic (E/E) architecture of road vehicles. It provides guidelines on how to design, implement, verify, and validate measures to ensure a safe system operation.

1.1 Context and motivation

In modern embedded systems, following such safety guidelines is complicated by a steadily growing number of functional requirements.

Self-driving cars, for instance, depend on perception, localization, motion planning, and trajectory control tasks [17]. The execution of such tasks requires considerable computing power, especially due to the increasing reliance on Artificial Intelligence (AI) algorithms. It is estimated that full self-driving behavior requires a performance of 1000 Tera Operations per Second (TOPS) and beyond [18]. This figure exceeds the capabilities of traditional Central Processing Units (CPUs) by several orders of magnitude.

Difficulties to maintain the Dennard scaling process [19], which had driven improvements in computational performance for over 30 years, turned the exploitation of large-scale parallelism, heterogeneous cores, and hardware accelerators into a key strategy for further performance growth [20].

Commercially available platforms for embedded computing reflect this paradigm shift: they are increasingly equipped with different CPU core clusters, Graphics Processing Units (GPUs), and more specialized hardware accelerators. To exploit these capabilities, the on-chip integration of previously separated applications has become an appealing goal. In the automotive domain, for example, it is possible to observe an ongoing trend toward vehicle-centralized architectures [21]. Beyond improving performance, this trend provides other benefits, such as reduced wiring harness weight [22].

The on-chip integration of different applications, however, means that resources like memory or the on-chip interconnect will have to be shared between them. Especially in mixed-criticality systems, where applications of different criticality levels coexist, this sharing increases the likelihood of interferences and a violation of safety requirements.

ISO 26262 considers this issue from a failure propagation perspective. Therefore, it defines *freedom from interference* as follows [16]:

> "absence of cascading failures between two or more elements that could lead to the violation of a safety requirement"

An informative annex of ISO 26262-6 [23] lists three categories of anomalies that can cause interferences between software elements:

1) timing and execution,
2) memory, and
3) exchange of information.

An example from the second category is an application writing to memory that is actually allocated to a different application.

In modern multicore processors, the potential for such anomalies is particularly pronounced. To avoid them, controlling the access that applications have to shared resources is of major importance. This control is generally achieved through temporal and spatial isolation.

Temporal isolation for off-the-shelf multicore systems is a major research topic. Approaches that contribute to this goal include statically partitioning hypervisors, cache partitioning, and memory bandwidth management (cf. Section 2.3.2). For the purposes of spatial isolation, most commercially available multicore platforms are equipped with hardware-enforced access protection features. As it will be shown in Chapter 2, these features are either built into a CPU, e.g., as Memory Management Unit (MMU), or they operate at the level of the shared on-chip interconnect.

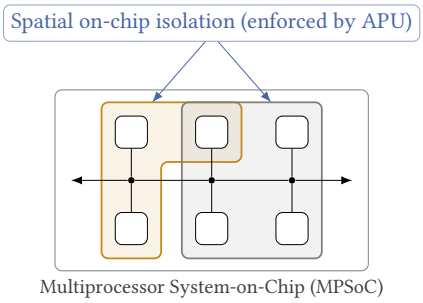


Figure 1.1: Informal illustration of how an APU can be used to 'partition' components of an MPSoC. Each square[1] visualizes an on-chip component, such as a CPU, a memory module, or a peripheral device. Shaded polygons represent partitions of components that are permitted to interact via the shared on-chip interconnect.

From a research point of view, access protection mechanisms that operate at the level of the shared interconnect are interesting components. In this thesis, they are referred to as Access Protection Units (APUs).

On modern Multiprocessor System-on-Chip (MPSoC) devices, APUs are crucial components for the spatial isolation between different CPUs. While a detailed introduction of their functionality is postponed to Chapter 2, a first illustration of how an APU can be used to partition on-chip resources is shown in Figure 1.1. By applying a particular APU configuration, it is possible to specify which on-chip components shall be able to issue transactions directed at a particular destination. During runtime, this specification is enforced by the APU, which constitutes a physical part of the on-chip interconnect.

In practice, APU configurations are typically generated by semiconductor vendor toolchains. The official toolchain for the Zynq UltraScale+ MPSoC by AMD/Xilinx,[2] for example, is described in [24]. Such toolchains deal with low-level aspects of the APU configuration process, but knowledge about which components shall be able to interact must still be specified by the developer. With the growing trend to integrate different applications on one MPSoC, specifying this knowledge and *proving* that a given specification leads to the required spatial isolation becomes a difficult task.

[1] Over the course of this thesis, illustrated shapes are often drawn with rounded corners. When such shapes are referenced, the aspect of rounded corners is not explicitly mentioned. In this case, this means that the term 'square' refers to a square with rounded corners.

[2] This platform will be introduced in more detail in Section 2.1.

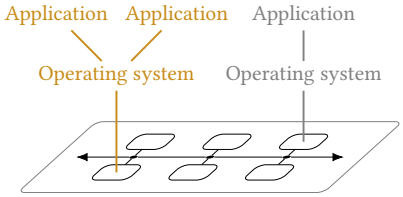


Figure 1.2: Vertical integration of an MPSoC with two operating systems and their applications. The configuration of its APU has an impact on both of these layers.

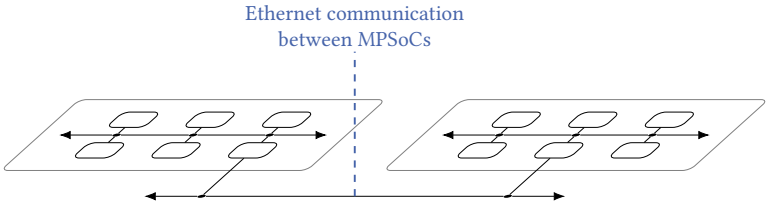


Figure 1.3: Horizontal integration of two MPSoCs via Ethernet. The configuration of their APUs has an impact on feasible off-chip transactions.

First, this is due to the impact that an applied APU configuration has on *vertically* integrated software, such as operating systems and their applications (cf. Figure 1.2). By controlling transactions between *hardware* resources, APUs have a significant impact on the interference potential between the *software* that is deployed to an MPSoC. This hardware-software mapping is not captured by commercially available toolchains, but it is a key factor that needs to be considered during the APU configuration process. To date, keeping track of this mapping is therefore the developer's responsibility.

Second, the *horizontal* integration aspect of MPSoCs needs to be considered. If multiple MPSoCs are connected using off-chip interconnects (cf. Figure 1.3), their combined APU configurations decide over physically feasible interactions. Commercially available vendor toolchains are not designed to capture this global perspective. Especially if MPSoCs from different vendors are interconnected, maintaining this perspective can be difficult.

Finally, it is necessary to ensure that the combined impact of all APU configurations is in line with relevant safety requirements. This assessment is a cross-layer process that needs to consider logical application knowledge, but it is again not covered by state-of-the-art toolchains.

In summary, it is possible to say that APUs are essential components in the ongoing integration trend, but a cross-layer methodology for their structured application in safety-critical environments is currently missing.

1.2 Problem formulation

The goal of this work is to create a *safety-centric* methodology that closes the identified gap. Because of the vertical integration aspect, this methodology must consider not only hardware components of MPSoCs, but also the runtime and application software executed on them. These system elements (cf. Definition 3.2 on page 42) are assumed to fulfill the following premise:

► **Premise 1.1:** Safety of the intended functionality

The intended functionality of system elements is safe.

Safety of the intended functional is an active field of research and the topic of ISO 21448 [25], for instance. This work assumes that sufficient measures to achieve this property have been applied. With this premise in place, the only possibility for physical harm is in response to a deviation from the intended functionality. Based on [16], this condition will be referred to as a *failure*:

► **Definition 1.1:** Failure

The *failure* of a system element is the termination of its intended behavior in response to an abnormal condition.

For the purposes of this work, the *propagation* of such failures is of major importance. Based on [16], this propagation is formalized as follows:

► **Definition 1.2:** Cascading failure

A *cascading failure* is the failure of a system element that then causes one or more other system elements to fail.

In the remainder of this work, Cascading Failures (CFs) are primarily referred to using their acronym, especially in compound terms.

Property	Vendor toolchain [24]	Objective of this work
User interface	Platform-specific	Platform-independent
APU configuration	Generated (single MPSoC)	Generated (multiple MPSoCs)
Impact analysis	Manually/externally	Automatically
Safety assessment	Manually/externally	Automatically

Table 1.1: Comparison between a representative vendor toolchain for APU configuration and the automated cross-layer methodology pursued by this thesis.

Spatial isolation enforced by an APU is able to eliminate the potential for certain CFs. However, it is not the only measure to restrict how failures can propagate. An operating system might configure the MMU of its processor to isolate its applications from each other, for example. Such a protection complements the underlying APU configuration and can therefore be relevant to decide if the overall system fulfills its safety requirements. Inspired by the terminology in [26], the following abstraction is introduced to refer to all such measures—often, but not always, based on spatial isolation:

► **Definition 1.3:** Logical isolation

Logical isolation restricts the possible interaction between system elements to prevent CFs between them.

Based on these definitions, the primary objective of this work is to define and evaluate a design methodology that combines the following features:

1) a platform-independent interface to describe APU configurations,
2) automatic generators for APU configuration code of different MPSoCs,
3) an automatic procedure that analyzes the impact that APU configurations and other logical isolation measures have on potential CFs, and
4) an automatic procedure that decides if the resulting CF potential is sufficiently limited to meet relevant safety requirements.

Table 1.1 shows a qualitative comparison between a state-of-the-art toolchain and this objective. The seamless integration of impact analysis and safety assessment steps has the potential to decrease the manual design effort, but it needs to be evaluated if such an approach is flexible enough to create and assess practical system implementations. To conduct this evaluation using representative case studies is a secondary objective of this work.

1.3 Contributions and outline

The main contribution of this thesis is a tool-supported design methodology built from the following key features:

1) an extensible library that stores knowledge about MPSoC architectures and platform-specific APU configuration algorithms,
2) a formal model to describe the hardware, the runtime, and the software architecture of an envisaged software system,
3) a platform-agnostic algorithm to auto-generate APU configuration code for selected MPSoCs from a system model,
4) an automatic procedure that generates a directed graph to capture the potential for CFs in an embedded software system, and
5) two alternative approaches to assess whether a particular CF potential violates relevant safety requirements—one based on an explicit whitelist and another one based on integrity levels.

Collectively, these features fulfill the objective outlined in Section 1.2. In this thesis, the features are defined in the form of a universal *pattern*, fully implemented, and evaluated using two practical case studies: (1) safe Ethernet usage on the Zynq UltraScale+ MPSoC and (2) a fail-operational architecture.

To complement the pattern, this work presents an Integer Linear Programming (ILP) approach to decide if it is necessary to apply certain isolation measures to obtain a safe system, and it describes how the pattern was integrated into the XANDAR toolchain for software system synthesis [1].

Further contributions of this thesis are (1) a cost-efficient approach to achieve fail-operational behavior, which was relevant in the context of the second case study, and (2) a timing-aware software development methodology, which facilitated the toolchain integration work.

Timing interferences due to sharing of on-chip resources are mentioned at selected locations, but their exhaustive treatment is beyond the scope of this work. A second topic explicitly beyond the scope of this work is general computer security. While safety-related effects of potential attacks can be addressed using the proposed design methodology, other security concerns (such as confidentiality or privacy) cannot.

The remainder of this thesis is structured as follows: Chapter 2 summarizes relevant background knowledge and reviews related work. Chapter 3 gives a high-level overview of the pattern and presents its system model, before the next two chapters cover individual pattern steps in more detail: Chapter 4 presents the logical isolation aspect and the determination

of CF potential; Chapter 5 describes the safety assessment procedure and the safety-driven ILP approach. Following this, Chapter 6 presents a full reference implementation of the pattern and evaluates the methodology in the context of two case studies. Chapter 7 uses the XANDAR toolchain to demonstrate a possible integration of the pattern, before Chapter 8 closes this thesis with a summary of application potential and directions for future work.

1.4 Previous publications

This thesis is based on several first-author publications:

1) Initial concepts for auto-generating APU configurations and using a graph to reason about their effects were presented in [2] and [3].
2) The level-based safety assessment procedure goes back to [4].
3) Integrating the APU configuration concept into the XANDAR toolchain was sketched in [5] and first performed in [6].
4) The timing-aware software development methodology, which was contributed to the XANDAR toolchain, is covered in [7].
5) Different fail-operational architectures, which are the topic of the second case study, were presented in [8–10].

In the broader context of this thesis, the author further contributed to [11–14]. These contributions are not directly related to the presented pattern and will therefore not be referenced further.

Chapter 2

Background and related work

This chapter covers relevant background knowledge in two steps. It first reviews fundamental aspects about multicore systems, which are necessary to understand the technical component of this work. Then, it summarizes mathematical concepts applied in the remainder of this thesis.

Based on a review of related work, this chapter closes with a comparison between the proposed pattern and similar approaches.

2.1 Multicore systems

For the purposes of this work, a *multicore* processor is defined as an Integrated Circuit (IC) built from multiple CPU cores. Each core implements arithmetic, logic, branching, and data transfer functions [27, p. 15]. The multicore processor is a kind of *multiprocessor*, i.e., a computer built from tightly coupled CPU cores. In a multicore processor, this tight coupling is achieved via monolithic integration. Traditional multiprocessors are characterized by two properties: the cores access a shared address space and are typically managed by a single operating system [27, p. 345]. The physical memory organization of multiprocessors can be either centralized or distributed.

▹ Example 2.1: *The architecture of a multicore processor with centralized shared memory is shown in Figure 2.1. Inputs and outputs are memory-mapped and therefore shown as logically belonging to the main memory.*

A deviation from these traditional properties is becoming increasingly common. In modern architectures, the address space does not necessarily have to

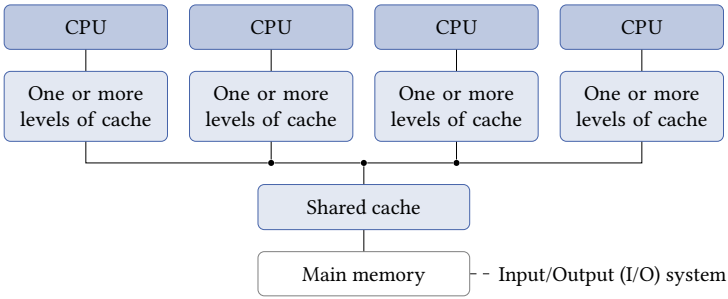


Figure 2.1: Example of a multicore chip with four CPU cores and a memory-mapped input/output system. Architecture replicated from [27].

be shared among all cores. Furthermore, it is possible to run each core independently of others, either with or without an operating system. This method of operation is commonly referred to as Asymmetric Multiprocessing (AMP). In contrast, the previously discussed case of a single operating system managing all cores is referred to as Symmetric Multiprocessing (SMP). Such deviations are particularly common in manycore architectures and heterogeneous computing systems, both of which are discussed below.

2.1.1 Manycore architectures

The *manycore* processor is a multicore processor with a large number of cores. Therefore, it is particularly suited for workloads with a significant degree of inherent parallelism. Manycore chips are often organized in *clusters*, where each cluster has its own private address space and is connected to the other clusters via a Network-on-Chip (NoC) infrastructure.

▹ Example 2.2: *Commercial manycore processors are the Kalray MPPA-256 with a total of 256 user cores [28] and the second-generation Intel Xeon Phi with up to 72 cores [29]. In the Kalray MPPA-256, two parallel NoCs connect 16 compute clusters and four input/output subsystems with each other; every cluster and subsystem has a private address space [28].*

Such a memory organization is also a common choice for current manycore implementations from academia; recent examples of such architectures can be found in [30] and [31], respectively.

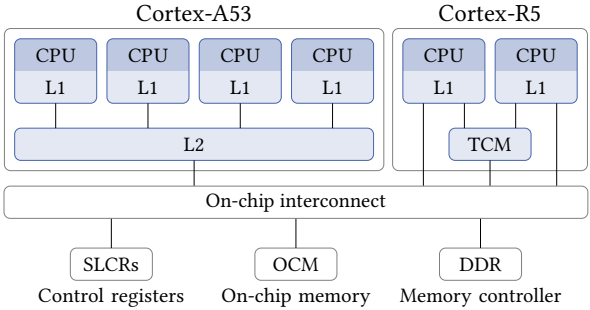


Figure 2.2: CPUs of the Xilinx Zynq UltraScale+ MPSoC (EG device) according to the block diagram in [33]. L1 and L2 refer to level-1 and level-2 cache, respectively. TCM is low-latency memory attached to the pair of Cortex-R5 cores.

2.1.2 Heterogeneous computing systems

Computing systems can be classified as *homogeneous* or *heterogeneous*. Processing units of a homogeneous system are identical, while heterogeneous systems comprise different types of processing units.

In a homogeneous system, processing units are often CPU cores of the same Instruction Set Architecture (ISA). In heterogeneous systems, CPU cores of different ISAs may be combined. The simultaneous integration of CPU cores and specialized coprocessors, such as a Graphics Processing Unit (GPU), is another example of a heterogeneous system [32].

In the context of this thesis, one type of heterogeneous computing systems is particularly relevant: multicore processors that integrate different clusters of CPU cores. Within each cluster, the CPU cores are homogeneous and often operated in SMP mode. Beyond cluster boundaries, however, the platform itself is of heterogeneous nature. In literature, each such cluster is sometimes referred to as an independent CPU of the platform.

▹ Example 2.3: *The aforementioned Zynq UltraScale+ MPSoC from Xilinx is a common example of a heterogeneous computing system.*

Figure 2.2 shows that the platform comprises two independent clusters of CPU cores: a Cortex-A53 and a Cortex-R5 processor, both Intellectual Property (IP) cores by Arm. The two core clusters implement different ISAs, which makes the platform heterogeneous. The cores within each cluster, however, are homogeneous. It is therefore possible to say that the platform comprises 6 CPU cores organized into two independent CPUs.

2.1.3 Multiprocessor System-on-Chip (MPSoC) devices

A single IC implementing numerous functions of an electronic system is referred to as a System-on-Chip (SoC) device. In practice, a SoC device often combines a microprocessor with peripherals such as Input/Output (I/O) controllers or a Field-Programmable Gate Array (FPGA). SoCs are comparable to microcontrollers but typically regarded as more sophisticated.

Multiprocessor System-on-Chip (MPSoC) devices are SoCs based on multiple processing cores. They can again be categorized into homogeneous and heterogeneous systems [34], where a heterogeneous MPSoC is built from processing cores of different kinds. In emerging mixed-criticality domains, such as autonomous driving, heterogeneous MPSoCs provide benefits in terms of cost, area, power, and performance [35].

▹ Example 2.4: *In addition to the components that are depicted in Figure 2.2, the Zynq UltraScale+ MPSoC implements a large variety of I/O controllers and an FPGA [33]; it is therefore an MPSoC in the sense of the definition above.*

Comparable products are i.MX 8 devices by NXP. The i.MX 8M, for example, integrates four Cortex-A53 cores with a Cortex-M4 core on a single chip.

2.1.3.1 Bus protocols for on-chip interconnects

MPSoC resources are interconnected using dedicated on-chip protocols such as those from the Advanced Microcontroller Bus Architecture (AMBA) standard by Arm. In individual specification documents, this standard defines, for example, the Advanced Extensible Interface (AXI) for high-bandwidth applications and the Advanced Peripheral Bus (APB) for isolated low-bandwidth purposes. AXI is a memory-mapped protocol that provides burst support and backward compatibility with APB [36]. In the AXI protocol, data is transferred between a *master* and a *slave* component using the five transaction channels shown in Table 2.1. Each channel transfers information into one direction, but a two-way handshake mechanism allows the receiving node to indicate whether it is ready to receive this information at a given time.

Every transaction is initiated by a master and targeted at a slave.[1] Read transactions make use of the former two channels (AR and R), while write transactions use the latter three channels (AW, W, and B).

[1] In the 2021 version of the AXI specification, this terminology has been updated. At the time of writing, however, the master-slave terminology is still predominant in consulted MPSoC manuals and will therefore be used in the following.

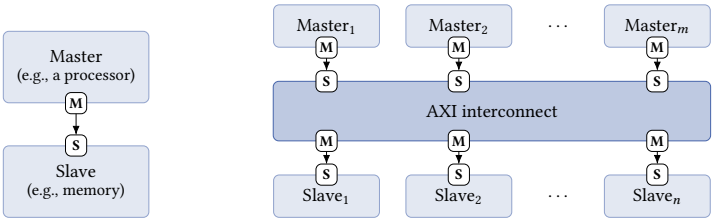
Name	Prefix	Direction		
Read address	AR	Master	→	Slave
Read data	R	Slave	→	Master
Write address	AW	Master	→	Slave
Write data	W	Master	→	Slave
Write response	B	Slave	→	Master

Table 2.1: AXI transaction channels and their signal prefixes according to [36]. The direction column reports the direction of payload signals in each channel, i.e., it excludes signals for the two-way handshake.

The data transfer direction depends on the type of the executed transaction: a write transaction transfers data from the master to the slave, while a read transaction transfers data into the opposite direction.

Terminology The scope of the AXI specification [36] is limited to transaction channels and their interfaces. In practice, however, entire components implementing these interfaces are often referred to as 'master' or 'slave' as well. Therefore, a component implementing both a master and a slave interface can be seen as both a master and a slave. This does not change the fact that such a component comprises two fully independent AXI ports, each implementing the respective interface. In the following, the simplified terminology is used when applicable, while the terms *master port* and *slave port* are used when it is necessary to emphasize that a statement refers to the AXI portion of an on-chip component in a strict sense.

Point-to-point connection Figure 2.3a shows a point-to-point connection from an AXI master to an AXI slave. A white square in the figure represents the master and the slave port, respectively. These ports are the actual endpoints of all transaction channels. However, as described above, the associated on-chip components are labeled as master or slave as well. In a point-to-point topology, the master is able to send transactions to only the connected slave. Using the address channels (AR and AW), read and write transactions can be targeted at specific locations of the address space realized by the slave. If the slave is a region of on-chip Random Access Memory (RAM), for example, the address transmitted via the AR or the AW channel will typically correspond to a particular memory location. In general, however, the interpretation of such an address is entirely controlled by the slave.



(a) Point-to-point (b) AXI interconnect

Figure 2.3: Comparison between an AXI-based point-to-point connection and a typical AXI system using an m-to-n interconnect. White squares represent AXI master (M) and slave (S) interfaces, respectively.

AXI interconnect Due to the significant number of on-chip components in an MPSoC, implementing point-to-point connections for all master-slave pairs is impractical. As shown in Figure 2.3b, a typical AXI system will therefore connect masters to slaves via a shared interconnect. Such interconnects are specifically considered by the AXI specification [36] and described as conventional devices that implement symmetrical master and slave ports to forward requests. Neither the system topology nor the arbitration scheme is prescribed by the protocol. One topology mentioned by the specification is to use shared address channels and multiple data channels. This topology, which is also referred to as a Shared-Address, Multiple-Data (SAMD) architecture [37], supports multiple simultaneous data transfers. A common arbitration scheme for practical implementations is round-robin scheduling [38]. The routing strategy that determines which slave to forward a particular transaction to is controlled by the interconnect itself. In typical implementations, this decision is made based on the address transmitted via the AR or the AW channel. In this case, every outgoing master port corresponds to one or more memory regions; a port will then be selected whenever an address from one of its associated memory regions is targeted.

2.1.3.2 On-chip interconnect of the Zynq UltraScale+ MPSoC

Using the AXI protocol as an example, the previous section gave a broad overview of bus protocols for the interconnection of on-chip resources. This section covers the interconnect of the Zynq UltraScale+ MPSoC and highlights an implementation detail that will become relevant when on-chip isolation mechanisms of this platform are discussed in Section 2.1.4.

Slave	Address space	
	Base	Size
Double Data Rate (DDR) memory		
→ Lower memory region	0x00000000	2 GiB
→ Control registers	0xFD070000	64 KiB
On-Chip Memory (OCM) module		
→ Memory region	0xFFFC0000	256 KiB
→ Control registers	0xFF960000	64 KiB
System-Level Control Registers (SLCRs)		
→ LPD registers (non-secure)	0xFF410000	64 KiB
→ FPD registers (non-secure)	0xFD610000	64 KiB
Controller Area Network (CAN) controller[1]	0xFF060000	64 KiB
Gigabit Ethernet (GEM) controller[1]	0xFF0B0000	64 KiB
Direct Memory Access (DMA) channels[1]	0xFFA80000	64 KiB
Tightly Coupled Memory (TCM) of the Cortex-R5[1,2]	0xFFE00000	128 KiB

[1] The reported address space refers to one of multiple module instances.
[2] This entry assumes that the Cortex-R5 cores are operated in lockstep mode.

Table 2.2: Selected slave components attached to the hard-wired on-chip interconnect of the Zynq UltraScale+ MPSoC along with their address space position [33, 39].

The hard-wired logic of the Zynq UltraScale+ MPSoC employs a NIC-400 by Arm [33] to implement the memory-mapped on-chip interconnect in both the Low-Power Domain (LPD) and the Full-Power Domain (FPD) of the device.

The interconnect is primarily based on the AXI protocol, but certain peripherals are integrated via APB. It supports addresses up to 40 bits wide and, therefore, implements a physical address space of up to 1 TiB. Sample masters with access to this address space are:

1) The two CPUs of the platform (Arm Cortex-A53 and Arm Cortex-A5)
2) The integrated GPU (of type Arm Mali-400)
3) Direct Memory Access (DMA) functions of components such as:
   (a) General-purpose devices for DMA (in both the LPD and the FPD)
   (b) Each of the four Ethernet (GEM) controllers (in the LPD)
   (c) The memory controller for NAND flash access (in the LPD)

Address space The address space of the Zynq UltraScale+ MPSoC is documented as a global one shared by all bus masters. Table 2.2 gives an overview

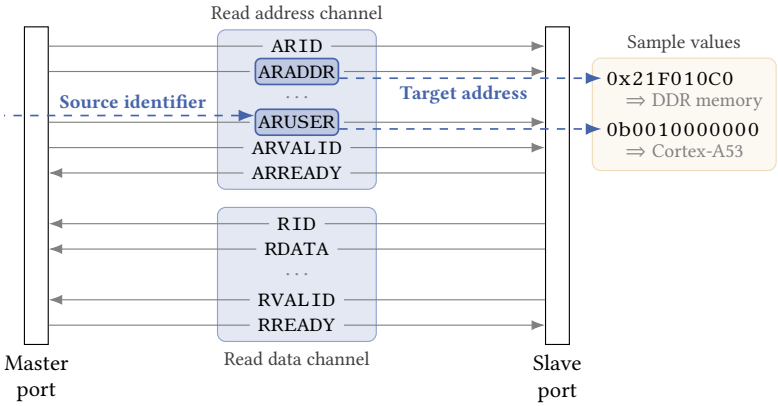


Figure 2.4: Signals involved in an AXI4 read transaction, extended with the usage of ARADDR and ARUSER by the Zynq UltraScale+ MPSoC (dashed arrows).

of selected slaves mapped to this address space. Not every master has access to the full address space, however. From the perspective of a particular master, parts of it can be masked or inaccessible due to an insufficient width of the outgoing address signals. From the perspective of the Cortex-R5, for example, the Tightly Coupled Memory (TCM) from Table 2.2 is also mapped to the base address `0x0` and, therefore, masks parts of the DDR memory.

Master identifiers With respect to spatial isolation, a particular aspect about the implementation of the interconnect is important: AxUSER signals from the AXI4 specification are used to encode and forward the source of every transaction traversing the interconnect. AWUSER and ARUSER are signals for user-defined extensions to write and read transactions, respectively. Every master is associated with a fixed 10-bit identifier, which is then transmitted using this extension mechanism of AXI4. Combined with the address, which encodes the targeted slave component, this identifier allows the interconnect to keep track of both the source and the destination of any transaction. Figure 2.4 visualizes this feature for a read transaction initiated by a Cortex-A53 core and targeted at the DDR memory. It must be noted, however, that the mapping from masters to master identifiers is not necessarily unique. In particular, certain cache coherency transactions will carry the same master identifier as one of the Cortex-R5 cores [24].

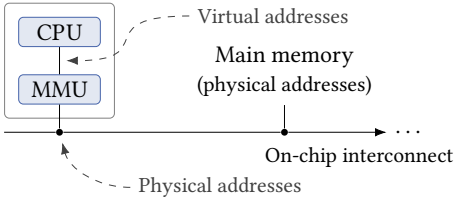


Figure 2.5: Operation principle of a Memory Management Unit (MMU). Visualization adapted from [40, p. 195], where peripherals are shown as off-chip components.

2.1.4 On-chip isolation mechanisms

Especially in heterogeneous MPSoCs, restricting bus masters in their access to the global address space is an important aspect of on-chip isolation. Therefore, such platforms are generally equipped with a combination of hardware mechanisms to achieve such restrictions. The applicability of the mechanisms presented in the following is not limited to MPSoCs, however.

2.1.4.1 Memory Management Units (MMUs)

The primary purpose of a Memory Management Unit (MMU) is to provide processes executed on a CPU with virtual address spaces. As shown in Figure 2.5, the MMU translates virtual addresses issued by a CPU core to physical addresses implemented by the attached interconnect. Operating systems executed on CPUs with an MMU often use this capability to provide every process with a dedicated region of physical memory. An example of such a CPU is the Cortex-A53 processor by Arm.

MMUs often implement *memory protection* by ensuring that a process is only able to access those parts of the physical memory space that are explicitly assigned to it. On platforms with a memory-mapped integration of on-chip resources (such as the Zynq UltraScale+ MPSoC discussed in Section 2.1.3.2), this also gives operating systems the ability to control the access that processes have to such resources.

2.1.4.2 Memory Protection Units (MPUs)

Processors without the need for a virtualization of the physical address space can still benefit from hardware support for memory protection. Such hardware support is provided by a Memory Protection Unit (MPU). It lacks the

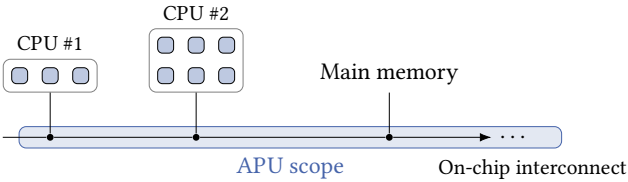


Figure 2.6: Logical protection scope of a sample APU. Other on-chip peripherals (such as I/O controllers) are not explicitly shown, but they are covered by the APU.

virtualization capability of MMUs, but operating systems can use it to ensure that processes access only portions of the physical memory space explicitly assigned to them. Examples of processors with MPUs are the Cortex-R5 or the Cortex-M4 by Arm.

2.1.4.3 Trusted Execution Environment (TEE)

The concept of a Trusted Execution Environment (TEE) describes the provisioning of protected processing environments with memory and storage capabilities [41]. It originates from the field of trusted computing and protects these environments against general software attacks generated in rich operating systems [42]. The implementation of such a TEE is typically based on hardware isolation mechanisms such as the Arm TrustZone technology.

Software executed on a processor with TrustZone support runs in either the *secure* or the *non-secure* state; these worlds are then isolated using strong hardware-enforced separation [43]. An important property of this protection is that its scope extends beyond the processor itself. In the sense of system-wide security, these isolation measures are integrated into the interconnect and on-chip peripherals as well [44]. In this regard, the TrustZone technology complements local protection delivered by MMUs and MPUs.

2.1.4.4 Access Protection Units (APUs)

Access Protection Units (APUs) are integrated into a shared on-chip interconnect of an MPSoC and enforce spatial isolation between attached resources. To do so, they examine all the transactions that traverse the interconnect (cf. Figure 2.6) and ensure that only permitted ones reach their respective destination. Today, they are common entities of commercially available MPSoCs.

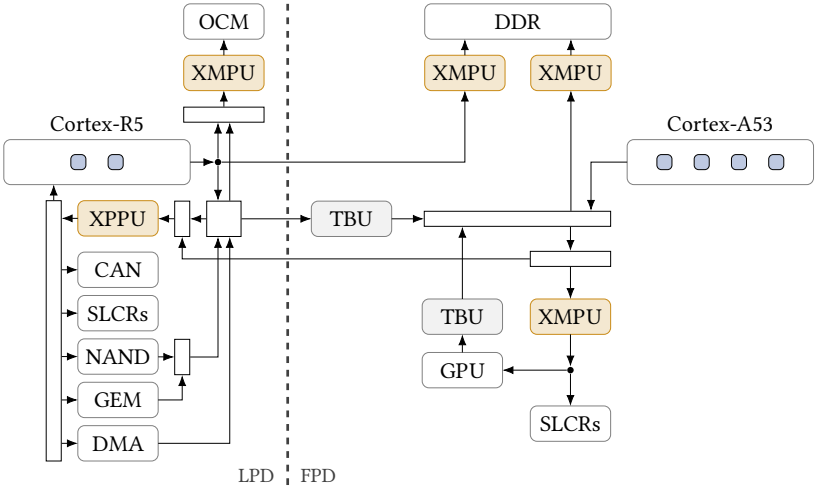


Figure 2.7: Selected components of the Zynq UltraScale+ MPSoC, including the XPPU and four of eight XMPUs. Visualization based on the full block diagram from [33].

APUs are typically runtime-configurable and use inherent properties about a transaction (source, target, …) to decide whether it is permitted. Compared to MMUs and MPUs, their protection scope extends to the entire on-chip interconnect; it is not limited to a particular CPU.

▹ Example 2.5: *The on-chip interconnect of the Zynq UltraScale+ MPSoC comprises multiple APUs: a Xilinx Peripheral Protection Unit (XPPU) and eight instances of the Xilinx Memory Protection Unit (XMPU). Figure 2.7 shows how this XPPU and a subset of all XMPUs are integrated into the AXI4 network. Both types of APUs make use of master identifiers transmitted via* AxUSER *signals to determine the source of a transaction. Combined with three other properties (addressed slave component, transaction type, and TrustZone state), the isolation units consult their internal configurations and 'poison' requests that are not permitted. Poisoned requests are either rejected by the designated receiver or, alternatively, routed to a dummy receiver from the very start [24].*

▹ Example 2.6: *In i.MX 8M devices, the Resource Domain Controller (RDC) is used to assign a bus master to one of four domains and to specify the access permissions of each domain [45]. The decision process is based on the source, the destination, and the transaction type.*

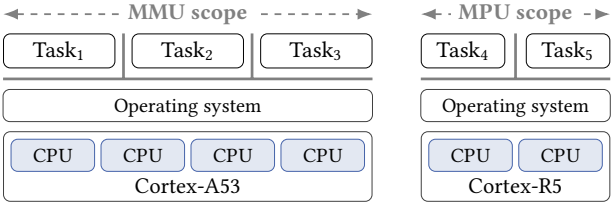


Figure 2.8: Usage of local isolation units by two sample operating systems running on the Zynq UltraScale+ MPSoC. Thick lines indicate spatial memory isolation.

2.1.4.5 Input/Output Memory Management Units (IOMMUs)

Like the APU, an Input/Output Memory Management Unit (IOMMU) operates at the level of the on-chip interconnect. Instead of partitioning on-chip resources, however, its primary purpose is to implement address translation for DMA-based transactions. It is typically managed by a particular CPU and participates in the memory management of this CPU. IOMMUs can therefore be seen as an extension of a particular MMU. From a memory protection perspective, their functionality is orthogonal to that provided by APUs.

▹ Example 2.7: *The Zynq UltraScale+ MPSoC features an IOMMU, which is for instance usable from the Cortex-A53. In the block diagram (Figure 2.7), part of its functionality is shown in the form of Translation Buffer Units (TBUs).*

2.1.5 Operating systems and hypervisors

A bottom-up view of an Operating System (OS) is that it allocates processors, memories, and other on-chip peripherals among tasks. This multiplexing of resources can be performed in time or in space [40]. When only one CPU core is available, processor sharing must be performed via time multiplexing. In a multicore scenario, operating systems managing multiple cores in SMP mode gain the ability to distribute tasks among these cores. Memory and other on-chip peripherals are typically space-multiplexed to avoid time-consuming swapping procedures and interferences, respectively.

For space-multiplexed memory, local protection units such as an MMU can be used to enforce a strong memory isolation at the OS level. Figure 2.8 shows an example of such a scenario for the Zynq UltraScale+ MPSoC. However, it is important to emphasize that (1) not every operating system makes use of these units and (2) the OS itself, which might not be fully trusted, will retain full

access to the underlying on-chip interconnect. In such scenarios, APUs can be employed as a global protection layer that replaces or complements the local protection that is managed by the OS.

On platforms considered by this work, shared peripheral devices are mapped to the global address space. The above statement that APUs can serve as a valuable protection layer is therefore transferrable to them.

The statement can further be transferred to bare-metal hypervisors, such as XtratuM [46]. Like an OS, bare-metal hypervisors allocate processors, memories, and other peripheral devices among their guest partitions. Depending on the specific environment, however, the local protection they implement might not be sufficient and can benefit from an APU configuration introducing a global protection layer beneath.

2.2 Mathematical foundation

The purpose of this section is to introduce *notations* and *terms* that are used for mathematical descriptions in this thesis. For a more extensive treatment of addressed topics, the reader is referred to the references cited below.

2.2.1 Fundamentals

▹ Remark 2.1: *The following description is inspired by [47].*

In this thesis, *sets* are denoted using capital letters, and their *elements* are wrapped in curly braces: $A = \{a_1, a_2\}$ is a set with elements a_1 and a_2, for example. The *cardinality* of a set is denoted using vertical bars, e.g., $|A| = 2$. *Set membership* is denoted using '$\in$', *set inclusion* is denoted using '$\subseteq$', and *proper set inclusion* using '$\subset$'. The *union* of sets A and B is denoted $A \cup B$, and their *intersection* is denoted $A \cap B$. Two sets are *disjoint* if they share no elements, i.e., if their intersection is equal to the empty set $\varnothing$.

$\mathbb{R}$ is the set of *real* numbers, $\mathbb{Z}$ contains all *integers*, $\mathbb{N}_0$ contains the naturals *including* zero, and $\mathbb{N}^+$ contains the naturals *excluding* zero.

A *tuple* is expressed by wrapping its elements in angular brackets: $\langle x, y \rangle$ is the tuple built from x and y. A tuple built from n elements is an *n-tuple*. In the context of linear algebra, an n-tuple is occasionally interpreted as a column vector with n rows. Vectors are denoted using a bold lowercase letter, such as $\boldsymbol{x}$ or $\boldsymbol{\sigma}$. Matrices (as a generalization of vectors) are expressed using ordinary capital letters. The difference between a set and a matrix is evident from the context in which the respective symbol appears.

Given sets X and Y, the Cartesian product $X \times Y$ is the set of all tuples $\langle x, y \rangle$ with $x \in X$ and $y \in Y$. The n-ary Cartesian product $X_1 \times \cdots \times X_n$ is the set of all $\langle x_1, \ldots, x_n \rangle$ with $x_k \in X_k$ for all $k = 1, \ldots, n$.

The Cartesian square X^2 is equal to the Cartesian product $X \times X$, and the n-ary Cartesian power X^n is its generalization to $X \times \cdots \times X$.

Set-builder notation can be used to specify elements of a set using predicates, for example as follows: $X \times Y = \{\langle x, y \rangle : x \in X \text{ and } y \in Y\}$. The power set $\mathscr{P}(S)$ contains all subsets of S, i.e., $\mathscr{P}(S) = \{S : S \subseteq X\}$.

A *(binary) relation* over sets X and Y is a fixed subset of $X \times Y$. In this context, X is the *domain* and Y is the *codomain*. Binary relations are denoted using capital letters (such as R) or designated symbols (such as $\leq$). A binary relation *on* set X is a subset of X^2. If a binary relation R holds for $\langle x, y \rangle$, we write xRy; if its symbol allows for it, we cross out the symbol to express that a relation does *not* hold for a pair of elements: given relation $\leq$ on $\mathbb{Z}$, e.g., $5 \leq 10$ means that $\langle 5, 10 \rangle \in (\leq)$, and $4 \not\leq 2$ means that $\langle 4, 2 \rangle \notin (\leq)$.

A binary relation R on X is *reflexive* if xRx holds for all $x \in X$. The relation is *antisymmetric* if for all $a, b \in X$, aRb and bRa implies $a = b$. Furthermore, it is *transitive* if for all $a, b, c \in X$, aRb and bRc implies aRc. Finally, the relation is *strongly connected* if for all $a, b \in X$, aRb or bRa.

A *partial order* on a set X is a binary relation on X that is (1) reflexive, (2) antisymmetric, and (3) transitive. A *total order* on X is a partial order on X that is also strongly connected.

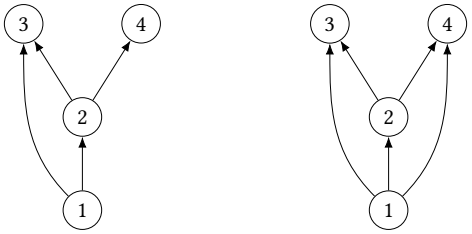
Given a binary relation R on X, the *transitive closure* of R is the smallest relation on X that is transitive and contains R. Its *reflexive transitive closure* is the smallest relation on X that is reflexive, transitive, and contains R.

A *function* from a set X to a set Y maps each element of X to exactly one element of Y. Since a function is a particular kind of binary relation, X is called the *domain* and Y the *codomain* of this function. We write $f : X \to Y$ to associate a function f with its domain and its codomain. Like common mathematical functions (log, sin, …), most functions defined in this work carry a multi-letter name instead of a single symbol.

2.2.2 Graph theory

▹ Remark 2.2: *The following definitions are adapted from [48].*

A *directed graph* or *digraph* is a tuple $G = \langle V, E \rangle$ in which V is a set of *vertices*, and the binary relation $E \subseteq V \times V$ is a set of *(directed) edges*. For a directed edge $\langle u, v \rangle \in E$, the vertex u is its *tail*, the vertex v is its *head*, the



(a) Binary relation R (b) Transitive closure of R

Figure 2.9: Graph representations of two binary relations on the set $X = \{1, 2, 3, 4\}$. First for $R = \{\langle 1, 2\rangle, \langle 2, 3\rangle, \langle 2, 4\rangle, \langle 1, 3\rangle\}$, then for the transitive closure of R.

vertex u is a *predecessor* of v, and the vertex v is a *successor* of u. A *loop* is an edge whose tail is equal to its head. Note that this definition does not allow for multiple parallel edges (i.e., with the same tail and the same head).

The graphical representation of a digraph is a diagram in which vertices are visualized as nodes (in the form of circles, rectangles, ...), and each edge is shown as an arrow from the tail to the head of that edge.

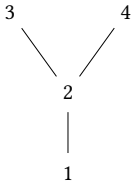
A binary relation R on X can be represented as a digraph by transforming each $x \in X$ into a vertex and each $r \in R$ into an edge. Such graph representations for two sample relations are visualized in Figure 2.9.

A *directed walk* is a sequence of $k \geq 1$ edges $W = \langle e_1, e_2, \ldots, e_k\rangle$, where for each $i = 1, \ldots, k-1$, the head of e_i is the tail of e_{i+1}.

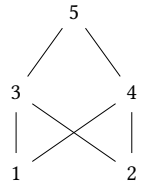
A *directed path* is a directed walk in which all involved vertices are distinct. We say that vertex v is *reachable* from vertex u if there is a directed path from u to v. In this terminology, a vertex is not reachable from itself; there may still exist a directed walk starting and ending at the same vertex.

Algorithmically, vertices reachable from a certain source vertex can be determined using depth-first search. As described in [49, p. 567], depth-first search runs with a worst-case time complexity $O(|V| + |E|)$. The problem of determining the reachability from *all* vertices can be reduced to finding the transitive closure of a digraph; [49, p. 648] describes how the Floyd-Warshall algorithm does so in a worst-case time of $O(|V|^3)$.

For a given digraph $G = \langle V, E\rangle$, a digraph with vertex set $V' \subseteq V$ and edge set $E' \subseteq E$ is a *subgraph* of G. Given a digraph $G = \langle V, E\rangle$ and a vertex subset $V' \subseteq V$, the subgraph *induced* by V' is denoted $G[V']$, has vertex set V', and its edge set contains all $e \in E$ whose head and tail are in V'.



(a) $X = \{1, 2, 3, 4\}$, partially ordered by the reflexive transitive closure of $1 \leq 2$, $2 \leq 3$, and $2 \leq 4$.



(b) $Y = \{1, 2, 3, 4, 5\}$, partially ordered by the reflexive transitive closure of $1 \leq 3$, $1 \leq 4$, $2 \leq 3$, $2 \leq 4$, $3 \leq 5$, and $4 \leq 5$.

Figure 2.10: Hasse diagrams for two partially ordered sets.

2.2.3 Ordered sets and Hasse diagrams

A *partially ordered set* or *poset* combines a set and a partial order on this set. Analogously, a *totally ordered set* is the combination of a set and a total order on this set. In both cases, we write $\langle X, \leq \rangle$ to express that '$\leq$' is a partial or total order on X, respectively. A simple example of a totally ordered set is $\mathbb{R}$ along with its natural less-than-or-equal relation, i.e., $\langle \mathbb{R}, \leq \rangle$.

For a partially ordered set $\langle X, \leq \rangle$, we occasionally use a textual description to express that its binary relation does or does not hold for a pair $a, b \in X$. The statement that 'a is greater than or equal to b' means $b \leq a$, for example.

A finite poset $\langle X, \leq \rangle$ can be visualized through the use of a *Hasse diagram*. This diagram is a drawing in which every $x \in X$ is represented as a vertex, and an *upward* line leads from $x \in X$ to $y \in X$ if y covers x, i.e., if $x \neq y$, $x \leq y$, and there is no third element $z \in X$ with $x \leq z \leq y$ [50, p. 1].

▹ Example 2.8: *The binary relation R introduced in Figure 2.9 is antisymmetric. Therefore, its reflexive transitive closure is a partial order. Referring to this partial order as '$\leq$', we obtain the finite poset $\langle X, \leq \rangle$. The corresponding Hasse diagram is visualized in Figure 2.10a. A second Hasse diagram is shown in Figure 2.10b; it describes a partial order with $|\leq| = 13$ elements, including $3 \leq 3$ and $1 \leq 5$.*

2.2.4 Lattice theory

▹ Remark 2.3: *The following definitions are borrowed from [51].*

A *semilattice* is a poset $\langle X, \leq \rangle$ subject to additional constraints: if the least upper bound $\sup\{a, b\}$ exists for all $a, b \in X$, it is a *join-semilattice*; if the greatest lower bound $\inf\{a, b\}$ exists for all $a, b \in X$, it is a *meet-semilattice*.

Using the *join* operation ($\vee$), $\sup\{a, b\}$ is also denoted $a \vee b$. Analogously, using the *meet* operation, $\inf\{a, b\}$ is equivalent to $a \wedge b$.

A poset $\langle X, \leq \rangle$ is a *lattice* if both $\sup\{a, b\}$ and $\inf\{a, b\}$ exist for all $a, b \in X$. This means that both semilattice types are a special case of a lattice.

It can be shown that join ($\vee$) and meet ($\wedge$) are idempotent, commutative, and associative [51, p. 10]. The following rules connect these operations to the underlying '$\leq$' relation: $a \leq b \Leftrightarrow a \vee b = b$ and $a \leq b \Leftrightarrow a \wedge b = a$ [51, p. 11].

▹ Example 2.9: *The poset from Figure 2.10a is a meet-semilattice. For this poset, selected meet results are* $3 \wedge 4 = 2$ *and* $2 \wedge 3 = 2$*, for example. Since the least upper bound* $3 \vee 4$ *does not exist, however, it is not a join-semilattice.*

Figure 2.10b shows neither a join- nor a meet-semilattice. It is not a join-semilattice, for example, because $1 \vee 2$ *does not exist: both* 3 *and* 4 *are upper bounds of* $\{1, 2\}$*, but neither of them is the least upper bound.*

2.2.5 Linear programs

▹ Remark 2.4: *The following definitions are based on [52].*

A *linear program* is the problem of finding a vector $\boldsymbol{x} \in \mathbb{R}^n$ that maximizes (or minimizes) the value of a linear *objective* function, given the constraint that $\boldsymbol{x}$ must satisfy a given system of linear inequalities (or equations). Without loss of generality, the following definitions cover the *maximization* such that a system of *inequalities* is fulfilled.

Given a vector $\boldsymbol{c} \in \mathbb{R}^n$, the objective function can be formulated as the matrix product $\boldsymbol{c}^T\boldsymbol{x} = c_1x_1 + \cdots + c_nx_n$. Given a matrix $A \in \mathbb{R}^{m \times n}$ and a vector $\boldsymbol{b} \in \mathbb{R}^m$, the expression $A\boldsymbol{x} \leq \boldsymbol{b}$ describes the m inequalities to satisfy. Linear programs can be solved in polynomial time [52, p. 105]. For the purposes of this book, we refer to this process as Linear Programming (LP). A widespread approach to solve a linear program is the *simplex* algorithm. It is often (but not always) able to find a solution or determine that no solution exists in polynomial time. The simplex algorithm expects a linear program to be provided in a *standard form* such as the following:

$$\begin{aligned} \underset{\boldsymbol{x} \in \mathbb{R}^n}{\text{maximize}} \quad & \boldsymbol{c}^T\boldsymbol{x} \\ \text{subject to} \quad & A\boldsymbol{x} \leq \boldsymbol{b}, \\ & \boldsymbol{x} \geq \boldsymbol{0}. \end{aligned}$$

lp_solve 5.5 [53] is free (LGPL-licensed) software able to process this standard form and apply a variant of the simplex algorithm to solve given LP formulations; it will be employed in Section 5.4 of this thesis.

An *integer program* constrains entries of $\boldsymbol{x}$ to $\mathbb{Z}$. We refer to the process of solving this adapted formulation as Integer Linear Programming (ILP). This thesis uses the following standard form for ILP problems:

$$\begin{aligned} \underset{\boldsymbol{x}\in\mathbb{Z}^n}{\text{maximize}} \quad & \boldsymbol{c}^T\boldsymbol{x} \\ \text{subject to} \quad & A\boldsymbol{x} \leq \boldsymbol{b}, \\ & \boldsymbol{x} \geq \boldsymbol{0}. \end{aligned}$$

ILP problems are NP-hard [52, p. 30], i.e., they are computationally more difficult to solve than LP problems. To date, no polynomial-time algorithm for the solution of arbitrary ILP problems exists. The *branch and bound* technique can often be applied to solve ILP problems in an acceptable time. In certain cases, it is possible to solve an ILP problem by solving its *LP relaxation*, i.e., by replacing $\boldsymbol{x} \in \mathbb{Z}^n$ with $\boldsymbol{x} \in \mathbb{R}^n$. A sufficient condition for this is that A is a *totally unimodular* matrix and $\boldsymbol{b}$ is an *integer* vector [52, p. 145].

lp_solve is able to solve ILP problems using branch and bound. Its input format allows users to constrain certain $\boldsymbol{x}$ entries to $\{0, 1\}$ instead of $\mathbb{Z}$. Section 5.4 uses an adapted standard form to capture such formulations.

2.3 Related work

This section begins with a summary of related work. It then compares the proposed methodology with selected approaches from this summary.

2.3.1 Spatial isolation in multicore systems

Various techniques for *spatial* isolation in multicore systems have been proposed. They are based on specialized architectures, specialized isolation components, or a combination of these solutions [54].

2.3.1.1 Architecture-based solutions

An example of a specialized architecture is one that associates cores with local resources (memories, I/O controllers, …) that these cores have exclusive access to; the Kalray MPPA-256 [28] features such an architecture. Similarly, Kliem et al. [55] proposed an FPGA-based architecture that is built from local, physically segregated subsystems. Gracioli et al. [56] described a similar approach applicable to MPSoCs that combine hard-wired logic with an FPGA: based

on the Zynq UltraScale+ MPSoC, they have shown how the Cortex-A53 cores can be used to implement spatially isolated partitions, each associated with scratchpad memory in the FPGA of the device.

2.3.1.2 Component-based solutions

Hardware virtualization through type-1 hypervisors is a common approach to partition platform resources among applications—not only in a multicore context. To control access to a shared address space, they typically leverage MMU, MPU, and IOMMU features that are readily available on state-of-the-art platforms. To partition shared (often last-level) caches, techniques such as coloring have been proposed; Modica et al. [57], for instance, have shown how Arm virtualization extensions can be used to implement cache coloring for a custom hypervisor. Examples of hypervisors with scheduling functionality are KVM [58] and XtratuM [46]. In the multicore context, statically partitioning hypervisors such as Jailhouse [59] have become a popular solution for embedded systems with mixed-criticality requirements. Jailhouse is spawned as a Linux kernel module, which transforms the running SMP system into a hypervisor operating in AMP mode; De Bonfils Lavernelle et al. [60] analyzed the spatial isolation enforced by this transformation and, as a secondary contribution, defined a generic methodology to reason about the spatial isolation properties of hypervisors. A recently introduced solution similar to Jailhouse but independent of Linux is the Bao hypervisor [61]; it was originally created for the Armv8 architecture. As follow-up work, Sá et al. [62] developed and released a RISC-V port of Bao.

To a certain degree, spatial isolation can also be achieved between applications of an OS. In the context of the Automotive Open System Architecture (AUTOSAR) standard, for example, an OS needs to provide certain memory protection functions to be compliant with the Classic Platform [63]. As part of the SIL2LinuxMP project, Allende et al. [26] investigated the applicability of Linux in mixed-critical multicore systems. Building upon IEC 61508 [64], they complemented the memory virtualization of Linux with separate kernel namespaces, system-call filtering, and other protection means. To reason about the safety of this approach, they introduced the *logical isolation* concept, which also served as the inspiration for Definition 1.3 of this thesis. According to [26], this concept describes the requirement that relevant behavior of elements cannot be impaired by the behavior of other elements. The authors make use of the fact that this property can be achieved without *full* spatial (and temporal) isolation—it is sufficient if spatial (and temporal) interferences are reduced to

a tolerable level. Compared with the bottom-up approach traditionally used to comply with IEC 61508, this strategy incorporates application-specific isolation requirements from the beginning. The authors finally evaluated their approach using a multicore partitioning for a safety-critical wind turbine use case, which was proposed by Perez et al. [65].

In the approaches covered so far, partitioning components that operate at the level of the on-chip interconnect (i.e., APUs) played no or only a marginal role. Such components have been proposed and applied in other contexts, however. Nojiri et al. [66] introduced a *domain partitioning* concept, which is based on *physical partitioning controllers* at different locations of the on-chip interconnect. These controllers are an early example of APUs available on modern MPSoCs (cf. Section 2.1.4.4). To validate their approach, the authors fabricated two dual-core processors and used the partitioning controllers to achieve spatial isolation between real-time and 'IT' functions in an automotive system. In a related work, Hattendorf et al. [67] evaluated the role of *shared* memory access control units and concluded that they are a lightweight solution to achieve spatial isolation between cores.

Targeting the *secure* co-hosting of applications on SoCs, Porquet et al. [68] introduced a multi-compartment model that is similar to TrustZone and uses hardware firewalls to filter transactions traversing the on-chip interconnect. In follow-up work [69], they employed these firewalls (as NoC-MPUs) to partition the shared memory of a NoC-based multicore platform. In this context, the firewalls contribute to the achievement of desired confidentiality and integrity properties. Other solutions to achieve these properties in a NoC context include packet tags [70] and static scheduling [71]. A security-centric solution similar to the NoC-MPU, but not defined in the context of NoCs, was proposed by Tan et al. [72, 73]. In their work, the authors introduced a *distributed* approach to manage and enforce access permissions in a heterogeneous MPSoC; the approach is driven by *isolation units* that are able to modify the access permissions of local and remote tasks during runtime.

2.3.2 Temporal isolation in multicore systems

As described above, the treatment of timing interferences goes beyond the scope of this work. If the pattern is applied in a real-time context, it expects its *user* to ensure that sufficient temporal isolation is achieved (cf. Premise 4.6 on page 113). Methods to do so have been studied extensively and are often related to the Worst-Case Execution Time (WCET) concept.

A large variety of specialized hardware that facilitates the determination of (tight) WCET bounds has been proposed, e.g., in the form of time-predictable processors [74–76] and mixed-criticality memory controllers [77, 78]. Such hardware is often complemented with tailored programming models [79, 80] and integrated into time-predictable multicore architectures [81–83].

Off-the-shelf platforms considered by this work are not necessarily based on such approaches, however. Their shared-resource architectures are primarily designed for flexibility and average performance, which aggravates the WCET determination process. On such platforms, timing interferences can be reduced through statically partitioning hypervisors, such as Jailhouse [59], or the use of fixed schedules, as they are implemented by XtratuM [46]. Other approaches to tackle contention issues include cache partitioning [57, 61, 84], memory bandwidth management [57, 85–87], and interrupt coloring [88]. In the context of SIL2LinuxMP [26], spatial isolation measures were complemented with CPU shielding to reduce timing interferences.

Further temporal isolation measures for multicore systems are documented in a 2021 survey by Cerrolaza et al. [54].

2.3.3 Decoding nets and the de-facto OS

In this thesis, the on-chip topology of MPSoCs is abstracted as one global address space.[2] This view is in line with the *documentation* on many commercially available platforms. In comparison to the actual complexity of modern on-chip topologies, however, it is (heavily) simplified.

Therefore, Achermann et al. [89] introduced the *decoding net* to reason about memory accesses and interrupts in modern computer systems. This directed graph of hardware components (CPUs, caches, memories, …) is a formal model of how the system handles addresses (including interrupt requests) emitted by each of these components. Therefore, it is first specified how each component handles the reception of a particular address: it can either *accept* it (like a memory), it can *translate* it (like an MMU), or both (like a cache). This assignment is *static* in the sense that it does not capture the dynamic behavior of any component in the system. The combined description of all static component behavior then constitutes the formal model.

Decoding nets are comparable to the Devicetree concept [90], which formalizes the description of system hardware and is especially prevalent in the Linux ecosystem. In comparison, however, the Devicetree is more focused on the

[2] Definition 3.8 introduces the *address space origin* concept to facilitate this abstraction.

initialization phase of an OS and less suited to reason about the correctness of memory accesses and interrupts during runtime.

In follow-up work, Achermann et al. have shown how an automated application of the concept can be used to prove the correctness of a virtual memory abstraction [91] and demonstrated that it is possible to generate page tables and memory maps from a decoding net [92].

Considering the growing complexity of computer systems, Fiedler et al. [93] defined the *de-facto OS* of System-on-Chip (SoC) platforms as the combination of a classical OS (such as Linux), relevant firmware, and 'hidden' cores of the system. Based on decoding nets [89], they created a formal model of modern SoC hardware to derive guarantees that a particular de-facto OS provides. Such guarantees consider the recursive nature of MMU and IOMMU configurations, which means that these configurations are perceived as entities whose unintended modification can lead to further bugs and vulnerabilities. Using given trust assumptions, the approach is able to reason about both integrity and confidentiality (excluding potential side-channel attacks).

Based on this, Fiedler et al. [94] derived the de-facto OS of the i.MX 8X platform by NXP. As part of this model, the RDC of the device is reflected by one decoding net per partition. In their paper, the authors report multiple observations that are also applicable to this thesis. Adapted to the terminology of this work, they can be summarized as follows:

1) Resource relations in modern SoCs are complex.
2) OS-enforced isolation does not lead to SoC-level guarantees.
3) APUs can tackle this issue but are not used by traditional OS kernels.

2.3.4 Information Flow Tracking (IFT) approaches

Information Flow Tracking (IFT) is a computer security technique that captures how information moves through a system [95, 96]. One of its early occurrences dates back to Denning [97], who created a formal model to specify information flow requirements in computer systems. Compared with access control mechanisms, which capture only how information is released from a source, information flow also considers the *dissemination* of released information.

The information flow model by Denning [97] is a 5-tuple

$$\langle N, P, SC, \oplus, \rightarrow \rangle,$$

where N is a set of *objects*, P a set of *processes*, SC a set of *security classes*, $\oplus$ is a *class-combining operator*, and $\rightarrow$ a *flow relation*. Processes are responsible for

information flow between objects, and examples of objects are files or program variables. Security classes describe classes of information, for example in terms of their respective security clearance (unclassified, confidential, …). Objects (and optionally processes) are associated with a security class. An operation involving security classes x and y leads to security class $x \oplus y$. $x \rightarrow y$ means that information from security class x it permitted to flow to security class y. Denning showed that if $\langle SC, \rightarrow \rangle$ forms a universally bounded lattice and $\oplus$ is the join operation, this model is particularly applicable to reasoning about information flow. For this case, she proposed (1) run-time enforcement and (2) compile-time certification mechanisms that ensure the fulfillment of a specified information flow policy between objects. While the approach was demonstrated in the context of confidentiality, it can also be applied to enforce integrity: in earlier work, Biba [98] showed that the problem of enforcing a (strict) integrity policy is the dual of enforcing confidentiality.

From a safety point of view, a CF that propagates from system element x to system element y can be regarded as a special information flow that is triggered by x and compromises the integrity of y. This way, an adaption of the lattice-based IFT formalism by Denning becomes applicable to the topic of this thesis. In fact, the second safety assessment procedure in Section 5.3 is derived from this formalism (but requires only a meet-semilattice).

Depending on the utilized verification technique, IFT approaches are often classified into *static* and *dynamic* approaches. Static approaches derive information flows from static system knowledge, while dynamic approaches monitor runtime behavior. In addition, IFT approaches can be categorized according to the abstraction layer they consider. For the purposes of this work, we distinguish *hardware-agnostic* from *hardware-aware* techniques.

2.3.4.1 Hardware-agnostic techniques

This section covers selected IFT approaches that target application software, operating systems, and other hardware-agnostic layers. They do not necessarily make use of a lattice-based information flow model—it is sufficient that they track the propagation of information through a computing system.

The application of static IFT to programming languages has been researched intensively. Leveraging the lattice model from [97], Denning et al. [99] proposed a compile-time certification mechanism that enforces confidentiality in software programs. The mechanism determines specified information flows between storage objects (such as constants, variables, or files) and compares these flows to statically specified security requirements (in the

form of a flow relation $\rightarrow$). Both explicit and implicit information flow in programs is covered, but covert channels are beyond the scope of the work. Volpano et al. [100] built upon this approach, formulated it as a *type system*, and proved the soundness of this type system. The subsequent evolution of language-based information-flow security is summarized by Sabelfeld et al. [101]. More recently, these language-based concepts have been combined with the Correctness-by-Construction (CbC) paradigm for program development: building upon the type system from [101], Schaefer et al. [102] described a *confidentiality-by-construction* approach that enforces a two-level security policy during the program construction phase.[3] Based on this, Runge et al. [103] extended the approach to arbitrary security lattices, which can also be used to express integrity requirements. The application to an object-oriented language was later discussed by Runge et al. [104].

The static IFT concept has further been applied at the operating system level. Murray et al. [105], for instance, employed it to prove that given certain restrictions (no DMA transfers, no covert channels, ...), the seL4 separation kernel enforces both confidentiality and integrity.

Complementing these techniques, a wide variety of dynamic approaches for software IFT have been proposed. They are generally based on the introduction of dedicated IFT instructions, either through source code instrumentation or binary rewriting [106]. Representative examples are TaintTrace [107], Dytan [108], DTA++ [109], and TaintDroid [110]. A more extensive overview of these and related techniques is given by Brant et al. [106]. Finally, it should be noted that there are dynamic techniques that enforce information flow as an inherent design property: Zeldovich et al. [111], for instance, applied such an approach to the HiStar operating system.

2.3.4.2 Hardware-aware techniques

IFT has been used at various levels of hardware abstraction, ranging from the analog circuit to the system level. These techniques may (but do not necessarily need to) make use of a lattice-based security model similar to that by Denning [97]. The following paragraphs highlight selected examples from an extensive survey on hardware IFT [96].

At the circuit level, Guo et al. [112] presented a static IFT technique to identify capacitor-based vulnerabilities they refer to as *charge-domain Trojans*. They demonstrated the applicability of their approach using a microcontroller design that was injected with such vulnerabilities.

[3] The by-construction perspective on non-functional properties is revisited in Section 7.1.1.

Gate-level IFT [113] complements a given network of Boolean logic with *tracking* or *shadow* logic that monitors how digital signals propagate through the network. The introduced logic can be used to enforce the desired information flow policy at the level of individual bits—either during runtime or as part of the design phase. Oberg et al. [95] applied a design-time variant of the approach to analyze information flows in two bus protocols: I^2C and USB. Using a process of refinements and IFT-based tests, they were able to identify and eliminate undesired information flow between bus nodes. In a comparable work by Oberg et al. [114], design-time IFT was used to secure a Wishbone crossbar against timing information flow. Hu et al. [115, 116] extended the gate-level IFT approach in such a way that arbitrary security lattices can be used to describe an information flow policy.

A notable IFT technique from the register-transfer level is to extend hardware description languages in such a way that they facilitate an automatic information flow analysis. An example of such a technique is SecVerilog, which was presented by Zhang et al. [117]. Such static techniques are analogous to the language-based solutions covered in Section 2.3.4.1.

The dynamic IFT approach by Suh et al. [118] is one of the first solutions that operate at the ISA level. To eliminate attacks based on stack smashing, string formatting, and similar techniques, Suh et al. associated each data block with a one-bit tag to indicate its authenticity. Management of these security tags was performed using custom hardware extensions and, optionally, instructions introduced via binary annotation. Other solutions at the architecture level delegate IFT metadata processing to dedicated cores or coprocessors. Early examples of this delegation strategy were proposed by Kannan et al. [119] or, in a multicore environment, by Nagarajan et al. [120].

In the context of high-level synthesis, methods that automatically apply static IFT or extend the synthesis result with dynamic IFT logic have been proposed. An example of the former is ASSURE [121], and the generation of dynamic IFT logic is for instance covered by TaintHLS [122].

Finally, IFT has also been applied to system-level scenarios. At this level, it tracks information flows that are due to explicit transactions, for example via a shared on-chip interconnect of a heterogeneous SoC. One example is the WHISK architecture proposed by Porquet et al. [123]; it implements dynamic IFT for heterogeneous SoCs that contain one or more loosely coupled accelerators. Also implementing dynamic IFT for loosely coupled accelerators, but operating at a coarser granularity, is the PAGURUS methodology by Piccolboni et al. [124]. This technique was evaluated using a custom SoC designed with the embedded scalable platforms methodology [125, 126]. It does not

require the accelerator architecture to be modified and is therefore applicable to third-party components. Other system-level approaches are based on the virtual prototyping concept [127, 128]: using transaction-level modeling in SystemC, they emulate the application of dynamic IFT under consideration of SoC-specific properties, such as the behavior of on-chip peripherals.

2.3.5 Model-based safety analysis

Shifting the focus from security back to safety, we now discuss model-based safety analysis as the final area of related work.

With fault trees, Markov chains, and similar techniques, approaches to conduct safety analyses have been available for decades. The structural differences between these approaches and underlying system models are often significant, however [129]. This can turn the information exchange between system and safety engineers into a time-consuming task or, in the worst case, lead to hazardous consistency issues. Model-based safety analysis is a discipline that keeps safety models aligned with those describing functionality, architecture, and other system properties.

Model-based safety analysis has been the topic of extensive research. AltaRica [129, 130], for example, is a high-level language that uses the notion of constraint automata to describe the behavior of systems; models can then be compiled to low-level representations such as fault trees. Rauzy [131] showed that a specifically limited subset of AltaRica can be used to describe both functional and anomalous aspects of a system in one formal specification. Similarly, the SAML framework proposed by Güdemann et al. [132] allows users to capture software, hardware, environment, and failure aspects as a single (textual) model. This model can then be used to conduct quantitative and qualitative safety analyses, such as the derivation of possible failure modes. Other examples of model-based safety analyses are the simultaneous creation of software architectures and fault trees [133], the derivation of safety artifacts from SysML models [134], or an application of techniques from the COMPASS toolset [135, 136].

A system description approach particularly related to this thesis is the Architecture Analysis and Design Language (AADL). This modeling language was developed for the specification and (repeated) analysis of real-time embedded system [137]. It is standardized by SAE International [138] and offers both a textual and a graphical notation. Examples of representable system properties are processors, memories, threads, and logical data flow. To extend the core of AADL with additional modeling features, various annexes

Approach	Year	APU-related activities[a]		
		Design	Configuration	Analysis
Nojiri et al. [66]	2009	●	○	○
Porquet et al. [69]	2011	●	○	○
AMD/Xilinx [24]	2021[b]	●	●	○
Achermann et al. [89]	2017	○	○	◐
Fiedler et al. [94]	2023	○	○	●
This work	2024	○	●	●

[a] Legend: ● explicitly covered; ◐ possible, but not explicitly covered; ○ unaddressed.
[b] The underlying MPSoC was announced in 2015.

Table 2.3: Coverage of the APU-related activities by related work, a representative off-the-self MPSoC [24], and this thesis.

have been developed. One such extension is the error modeling annex EMV2, which is intended for fault modeling and automated safety analysis. In this context, Delange et al. [139] presented tools that translate EMV2 annotations into certification documents such as a fault tree analysis, and Brunel et al. [140] studied the translation from EMV2 to AltaRica. Such analysis features coexist with a rich collection of other approaches that build upon AADL. Using the Ocarina [141] or RAMSES [142] toolchain, for example, it is possible to transform an AADL model into executable code for embedded target platforms.

At the time of writing, discussions about the definition of limited subsets of the AADL standard have started [143].

2.3.6 Comparison with the proposed methodology

Three contributions of this thesis are particularly related to the work outlined above: the consideration of APUs, the automatic analysis of hardware interactions, and the safety assessment procedure. They are now individually positioned in the context of state-of-the-art solutions from above.

2.3.6.1 Consideration of APUs

With respect to the APU concept, the proposed methodology is concerned with the automation of two main activities: their configuration and the analysis of the configuration impact. Table 2.3 shows a qualitative comparison between this and the APU consideration scope of most closely related work.

Approach	Properties of the analysis technique		
	Subject (*what?*)	Scope (*where?*)	Means (*how?*)
Achermann et al. [89]	Memory/interrupts	Computer system	Formal analysis
Fiedler et al. [94]	Memory/interrupts	SoC (incl. devices)	Formal analysis
Oberg et al. [95]	Information flow	Bus realization	Formal analysis
Pieper et al. [128]	Information flow	SoC (incl. devices)	Emulation
This work	Cascading failures	MPSoC network	Formal analysis

Table 2.4: Qualitative comparison between the analysis technique of this and closest related work. All listed approaches perform a design-time analysis.

The proposed methodology complements previous work outlined in Section 2.3.1: while Nojiri et al. [66] and Porquet et al. [69] presented specific APU designs, their work did not cover automated configuration or analysis tasks. More recently, toolchains provided by MPSoC vendors have been increasingly equipped with functions to configure APUs (cf. Section 1.1). However, the purpose of these functions is only to translate platform-specific knowledge about accepted or prohibited on-chip transactions into corresponding configuration code; they are not concerned with an analysis of their impact.

Decoding nets (cf. Section 2.3.3) can be used to analyze the impact of APU configurations in a fine-grained manner. The approach by Achermann et al. [89] is applicable to do so, but this is not explicitly claimed or demonstrated by the authors. The work by Fiedler et al. [94] reflects APU-enforced partitions using dedicated decoding nets. Like this thesis, it therefore makes APU configurations accessible to a formal treatment. Unlike this thesis, it is not concerned with automating the configuration process.

▹ Remark 2.5: *Other spatial isolation measures from Section 2.3.1, such as hypervisors and operating systems, are also related to the proposed methodology: they are one of the logical isolation measures that can be considered by the pattern. More specifically, they are possible means to achieve process isolation, which is introduced in Chapter 3 and formalized in Chapter 4.*

2.3.6.2 Automatic analysis of hardware interactions

From a hardware point of view, the analysis technique of this work is further characterized and compared in Table 2.4. The goal of this technique is to formally reason about CF potential in a network of MPSoCs.

Decoding nets by Achermann et al. [89] are also a static analysis technique, but they are concerned with memory accesses and interrupts in general computer systems. As such, they address a different (more generic) problem than this work's analysis technique. Most importantly, they were not tailored to the needs of safety-critical systems and, for example, do not capture the impact of indirect dependencies between on-chip components of MPSoCs.

Furthermore, the de-facto OS proposed by Fiedler et al. [94] and this thesis share several properties. Both are based on a static formal model of SoC characteristics. Furthermore, they are both based on the concept of placing a certain level of 'trust' in components and running an automated procedure to determine potentially problematic interaction potential. At the same time, the approaches differ in their respective focus: while the de-facto OS is concerned with a fine-grained analysis of generic isolation properties in a single SoC (and uses decoding nets to reason about them), this work emphasizes CF potential in an entire network of execution platforms.

Methodologically, this work's analysis technique is further comparable to the IFT procedure performed during design-time. Table 2.4 lists two approaches that are particularly related: the gate-level IFT approach that Oberg et al. [95] apply to control information flows in off-chip buses and the emulation of dynamic IFT by Pieper et al. [128]. Both differ from this work in the fact that they are concerned with general information flow (rather than CFs). Like this thesis, however, Oberg et al. [95] synthesize a static representation of relevant hardware interactions and use it to formally reason about the propagation of undesired events (in their case: taints). The similarity with Pieper et al. [128] is due to the fact that interactions with attached on-chip peripherals are explicitly covered. Rather than statically reasoning about CFs, however, their work leverages an executable SystemC model.

2.3.6.3 Safety assessment procedure

The safety assessment procedure introduced by this thesis is seamlessly integrated into the overall methodology. User intervention is needed only to capture relevant safety requirements. Apart from that, the procedure obtains all relevant knowledge from the system model and inferred CF potential. As such, it is an example of model-based safety analysis in the sense of Section 2.3.5. Instead of on fault trees or Markov chains, however, the procedure operates on a directed graph that is inspired by the field of IFT.

Finally, there is one major similarity in the way Allende et al. [26] and this thesis approach the safety assessment. Both works recognize that (full) spatial

and temporal isolation are not a necessary requirement to achieve safety. Based on this, they both introduce the concept of *logical isolation* to reason about interactions that have the potential to jeopardize the integrity of critical system elements (cf. Section 1.2 and Section 2.3.1). This integrated top-down approach constitutes a novel technique to analyze the potential of problematic interferences in multicore systems: it takes relevant safety requirements into account and, based on them, ensures that the system is free of *dangerous* failure propagation as the primary objective.

Chapter 3

Concept and system model

This work is concerned with functional safety properties at three abstraction levels of an embedded software system: its hardware, its runtime, and its software layer. Execution platforms, such as MPSoCs, are the foundation of the hardware layer. The runtime layer is concerned with runtime environments, while the software layer considers application software.

MPSoCs are execution platforms of particular importance. On these devices, CPUs and on-chip resources are often tightly integrated with each other, which increases the risk of undesired and potentially hazardous on-chip interferences. At the same time, APUs found in commercially available MPSoCs can be used to achieve a logical on-chip isolation and, therefore, to avoid or constrain CFs at the hardware layer.

The concept proposed in this chapter generates APU configuration code, evaluates its effect in the context of other specified isolation measures, determines remaining CF potential in the system, and ensures that the identified CF potential does not violate a safety requirement.

These activities are guided by a system model that captures relevant design knowledge at all three abstraction levels. Interactions at the software layer, for instance, are considered and automatically assessed from a cross-layer perspective. In systems without APUs, it is possible to apply the static analysis procedure in isolation; in this case, the methodology does not actively eliminate CFs, but it still derives guarantees that can be used to reason about the functional safety of a system.

To emphasize the automated nature of the proposed concept, it is referred to as a *pattern* or, more specifically, a *safety pattern* for logical isolation.

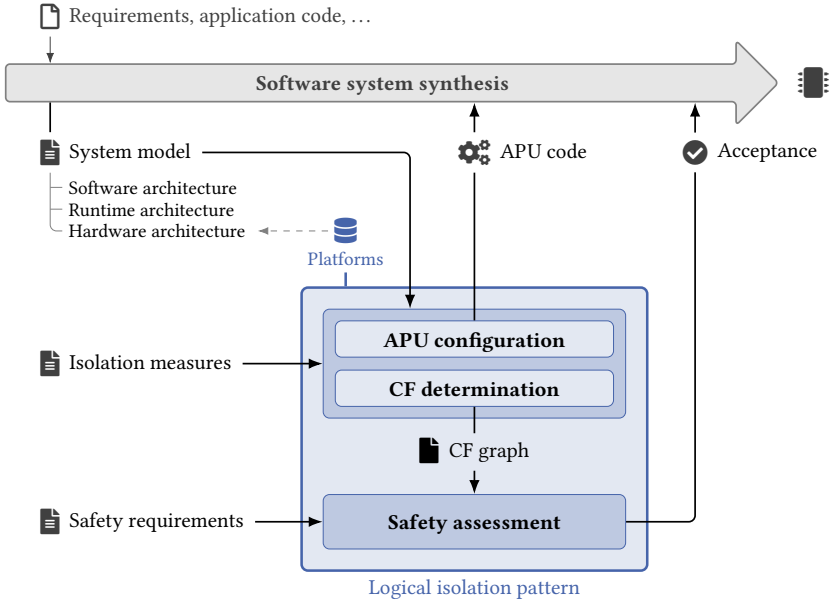


Figure 3.1: Concept and possible toolchain integration of the logical isolation pattern. The arrow at the top is an example of a superordinate synthesis process.

3.1 Overview of the logical isolation pattern

This section gives an informal explanation of the logical isolation pattern, its inputs, and its outputs. A high-level overview of these aspects is depicted in Figure 3.1, which is a generalized and refined version of results presented by the author in [6]. The pattern itself consists of two sequentially executed steps: (1) a combined *APU configuration* and *CF determination* process followed by (2) a *safety assessment* procedure. Furthermore, the pattern defines and exposes an *execution platform library*; this library contains all execution platform types known to and supported by the pattern. An example of such a type is the Zynq UltraScale+ MPSoC introduced in Example 2.3.

Without loss of generality, the pattern was designed as a reusable module to be invoked during the partly or fully automated synthesis of software systems. Industrial examples of such synthesis solutions are commercially available AUTOSAR toolchains. Academic examples are RAMSES [142], Lingua

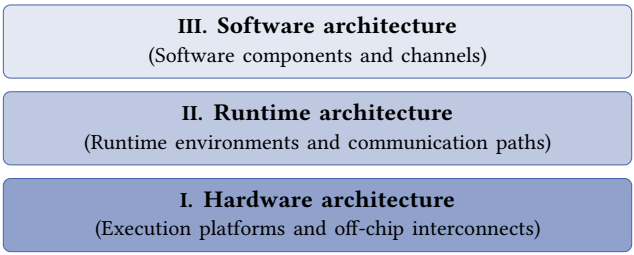


Figure 3.2: Layer structure of system model instances, with the primary model elements of each layer in parentheses. From the bottom up, the layers build on each other.

Franca [144], and the XANDAR toolchain [1]. A possible integration of such a synthesis process is visualized as the large arrow in Figure 3.1. It is important to emphasize, however, that both less and more integrated applications of the pattern are generally feasible.

3.1.1 APU configuration and CF determination

In this step, (1) APU configuration and (2) CF determination are performed in an integrated procedure. While its exact functionality will be the topic of Chapter 4, the following explanations convey a first complete overview of the step. It operates on two key inputs:

System model: The *system model* describes knowledge at three layers of the embedded system stack. These layers are the *hardware architecture*, which describes physical system properties; the *runtime architecture* deployed to a hardware architecture; and the *software architecture*, which is in turn executed by a runtime architecture. A hardware architecture can instantiate and optionally interconnect platforms from the execution platform library. This layer structure is visualized in Figure 3.2 and will be thoroughly covered in Section 3.2.

Isolation measures: A possibly empty set of *isolation measures* describes the applied strategy for logical isolation. As shown in Figure 3.3, an isolation measure is either a *generation request* or a *barrier declaration*. Generation requests instruct the pattern to generate configuration code for specific APUs of the system. Barrier declarations provide the pattern with knowledge about externally applied measures, such as an MMU-based protection or relevant application behavior.

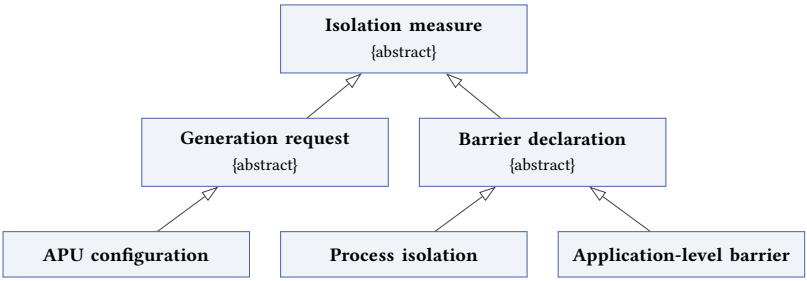


Figure 3.3: Types of isolation measures represented in the form of a class diagram. This work considers one generation request and two barrier declarations.

Based on its inputs, the procedure handles APU configuration requests, processes barrier declarations, and determines the remaining potential for CFs in the system. We introduce a dedicated term for the latter concept:

► Definition 3.1: CF potential

CF potential from system element v to system element v' is the possibility that a CF originates from v and leads to a failure of v'.

▹ Remark 3.1: *CF potential is transitive. If it exists from v_1 to v_2 and simultaneously from v_2 to v_3, then it also exists from v_1 to v_3.*

Definition 3.1 makes use of the *system element* notion, which has not been formalized yet. Using the three model layers, it is possible to do so:

► Definition 3.2: System element

A *system element* is a hardware, runtime, or software element that operates in the embedded system under consideration.

In other words, system elements are physical or logical entities that exist in a deployed system; they are not only a design-time concept.

▹ Example 3.1: *A specific MPSoC, its CPUs, and its on-chip memory are examples of system elements at the hardware layer. An example of a system element at the runtime layer is a Linux distribution executed on a CPU. At the software layer, logic executed by a Linux process can be represented as a system element.*

▹ Remark 3.2: *The above definition is based on ISO 26262 [16], which describes an element as "system, components (hardware or software), hardware parts, or software units." This notion includes single hardware parts, such as the CPU of a microcontroller, which are not covered by the "component" definition.*

System elements are strongly related to system model entities, which serve as an input to the pattern. However, there is not necessarily a one-to-one correspondence between these entities and system elements according to Definition 3.2. As it will be shown in Section 3.3, selected system model entities are translated into zero or more than one system element.

3.1.1.1 APU configuration

Generation requests are handled by the *APU configuration* process:

► Definition 3.3: APU configuration

APU configuration is the process that generates APU configuration code for each MPSoC for which this is requested by an isolation measure.

APU configuration code generated by this process is returned to the user invoking the logical isolation pattern. Each such configuration is a platform-specific artifact ready to be applied to its respective MPSoC.

▹ Example 3.2: *APU configuration code for the Zynq UltraScale+ MPSoC may be a C program that writes to XPPU and XMPU registers of the platform.*

3.1.1.2 CF determination

The task of CF determination can be formalized as follows:

► Definition 3.4: CF determination

CF determination is a structured process that determines all CF potential originating from each system element.

At this point, it makes sense to revisit Definition 3.1 and highlight the fact that CF potential describes the *possibility* of a CF between system elements. Therefore, a trivially correct result of CF determination is the statement that CF potential exists between all system element pairs. However, this

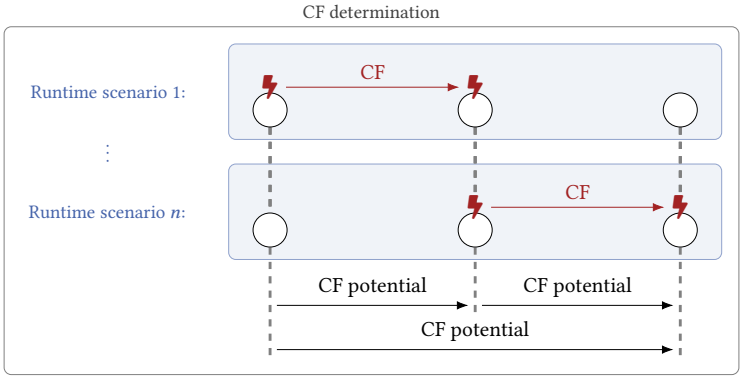


Figure 3.4: CF determination as a design-time concept. Circles represent system elements, while lightning bolts signify their failure. Red arrows represent a CF that occurs in the respective runtime scenario.

pessimistic result does not carry any significance. It gains significance if infeasible CF potential is actively removed from the result. To achieve this, the process combines fundamental knowledge about interactions in embedded software systems with the specified isolation measures.

As illustrated in Figure 3.4, CF determination is a design-time process that deals with all possible runtime scenarios. For each such scenario, its goal is to find an accurate description of CFs that may actually occur during system operation. The union of all these CFs is the CF potential. Knowledge determined by the procedure is captured in the form of a graph:

▶ Definition 3.5: CF graph

CF graphs are directed graphs $G_\gamma = \langle V, E \rangle$, where $v \in V$ represents a system element, and E contains a directed path from $v \in V$ to $v' \in V$ if and only if CF potential leads from v to v'.

Since the edge set of G_γ is defined in terms of its directed paths, the transitive closure of G_γ is semantically equivalent to G_γ itself. Section 4.3 exploits this property to reduce the number of edges that are to be added.

▹ Example 3.3: *The graph in Figure 3.5 is a CF graph for a sample system with system elements* $V = \{v_1, v_2, \ldots, v_{14}\}$. *According to this graph, a failure of*

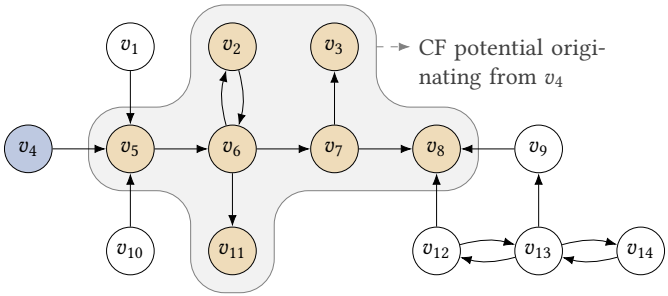


Figure 3.5: CF graph $G_\gamma = \langle V, E \rangle$ with $|V| = 14$ system elements and $|E| = 17$ directed edges. Shading shows the CF potential originating from v_4.

system element $v_4 \in V$, for example, has the potential to cause a failure of v_2, v_3, v_5, v_6, v_7, v_8, and v_{11}. These seven relationships might be conservative in the sense that, in practice, they do not actually give rise to a CF. If there is a potential for the CF, however, then there must be a directed path.

Unlike generated APU configurations, CF graphs are primarily an internal result. As shown in Figure 3.1, they are forwarded to the safety assessment step, which is informally described in the next section. However, a CF graph gives users invoking the pattern detailed knowledge about CF potential in a system. Therefore, it can be useful to return it as an additional output.

3.1.2 Safety assessment

CF potential is concerned with the propagation of all failures, not only with the ones that can lead to the violation of a safety requirement. However, the subset of potentially hazardous failures is the one relevant for the purposes of this work. The safety assessment operates on a CF graph to decide whether captured CF potential is acceptable from a functional safety point of view.

For given CF potential to actually cause harm, and therefore jeopardize safety, three events have to occur:

1) **Fault manifestation:** The system element from which the CF potential originates is affected by a fault that manifests as a failure.
2) **Failure propagation:** This failure propagates through the system to cause a failure of another system element.
3) **Physical harm:** By interacting with the physical environment in an unintended manner, the affected system element causes harm.

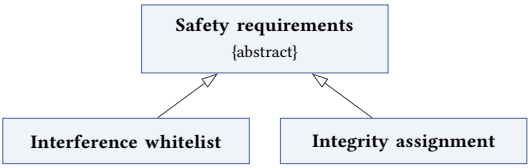


Figure 3.6: Safety requirements input depicted in the form of a class diagram. In this work, two alternative specification approaches are considered.

CF graphs are only concerned with failure propagation (item 2 of the list). They apply the conservative worst-case assumption that every directed path can actually give rise to a CF. Whether this possibility violates a safety requirement must be assessed based on the probability of a fault manifestation (item 1 of the list) and the severity of potential physical harm (item 3 of the list). To decide this question for particular CF potential, these two aspects will usually have to be considered in conjunction and relative to each other.

▹ Example 3.4: *Let v_1, v_2, and v_3 be system elements. v_1 is a CPU with a high susceptibility to faults. v_2 and v_3 represent software executed by this CPU. Therefore, there is CF potential from v_1 to both v_2 and v_3.*

Under the assumption that v_2 does not affect the physical environment at all, CF potential from v_1 to v_2 cannot be a safety issue. If, however, v_3 controls a safety-critical actuator, the CF potential from v_1 to v_3 might be a safety hazard and therefore unacceptable.

This knowledge needs to be provided to the logical isolation pattern in the form of a third external input: the *safety requirements*. They specify whether a CF between two given system elements is permitted. Based on this, safety assessment is defined as follows:

► **Definition 3.6:** Safety assessment

The *safety assessment* procedure ensures that CF potential originating from each system element is in line with the specified safety requirements; it *accepts* a design if and only if this is the case.

Figure 3.6 shows two alternative approaches to specify safety requirements. An *interference whitelist* is an exhaustive list of all accepted CF potential in the system. The lattice-based *integrity assignment* concept allows users of

the pattern to annotate selected system elements with provided and required integrity levels, respectively; from these assignments, the safety assessment determines the acceptance of all relevant CF potential.

▹ Remark 3.3: *CF potential is relevant if, according to the fault model described in Section 3.3, the system element it originates from can be expected to fail.*

The result returned by the safety assessment process is a binary acceptance value. A superordinate synthesis process, for instance, can use this result to abort the synthesis if safety cannot be guaranteed.

When a system is rejected, it is also practical to return reasons for this rejection as a secondary result. If interference whitelists are used, for example, an explicit list of CF potential violating the whitelist can support users of the safety pattern in further development steps.

3.2 Formal system model

The three-layered system model is a key input of the logical isolation pattern. Apart from the fact that it references the execution platform library, it is a self-contained artifact. Its entities are translated into zero or more system elements (according to Definition 3.2), which makes the system model an important prerequisite before the fault model can be described.

The system metamodel is inspired by state-of-the-art approaches for hardware/software system modeling, such as AADL and AUTOSAR. Since the proposed generation and analysis procedure is a specialized use case, however, suitable concepts from these approaches are incorporated into a tailored metamodel. This metamodel is described in the remainder of this section.

▹ Remark 3.4: *The capabilities of the metamodel are deliberately limited to aspects required by the logical isolation pattern. State-of-the-art modeling solutions, such as AADL, are generally more flexible. This means that they can often be used to express the knowledge captured by a system model as introduced below. Due to their complexity, however, they are less suited to communicate the bare minimum of knowledge that the pattern needs to be able to operate.*

System model entities Table 3.1 lists the eight types of system model entities that are explicitly described by the creator of a model instance. One of them is the execution platform, which is a special type of entity taken from the execution platform library of the pattern. Each such entity can enrich

Acronym	Model entity	Description
–	Execution platform[1]	Hardware platform instantiated by the system, e.g., a Zynq UltraScale+ MPSoC.
–	Off-chip interconnect	Off-chip bus attached to one or more peripheral devices[2], e.g., a CAN bus.
RTE	Runtime environment	Infrastructure software executed directly on a processing unit[2], e.g., a Linux distribution with certain management features.
LP	Local path	Dedicated region of a memory module[2] used for the communication between a pair of local RTEs, e.g., a portion of DDR.
GP	Global path	Virtual channel in an off-chip interconnect used for the communication between two remote RTEs, e.g., a set of CAN messages.
SWC	Software component	Application executed by an RTE, e.g., a binary spawned as a user-space Linux process.
–	Port (= SWC port)	Asynchronous input or output interface exposed by a particular SWC.
–	Channel	Unidirectional information flow path from one SWC port to another.

[1] Must be instantiated from the execution platform library of the safety pattern.
[2] Automatically inferred from knowledge about instantiated execution platforms (cf. Table 3.2), i.e., not explicitly described by the creator of a model instance.

Table 3.1: Entities explicitly instantiated by the creator of a model instance. From top to bottom, the hardware, runtime, and software architecture layer is covered.

Acronym	Model entity	Description
PROC	Processing unit	CPU or a comparable component of an execution platform, e.g., a Cortex-A53 processor.
MEM	Memory module	Memory owned by an execution platform, e.g., external DDR memory or the OCM module.
DEV	Peripheral device	On-chip peripheral of an execution platform, e.g., a CAN controller or the SLCRs.

Table 3.2: Inferred system model entities at the hardware architecture layer along with illustrative examples taken from the Zynq UltraScale+ MPSoC.

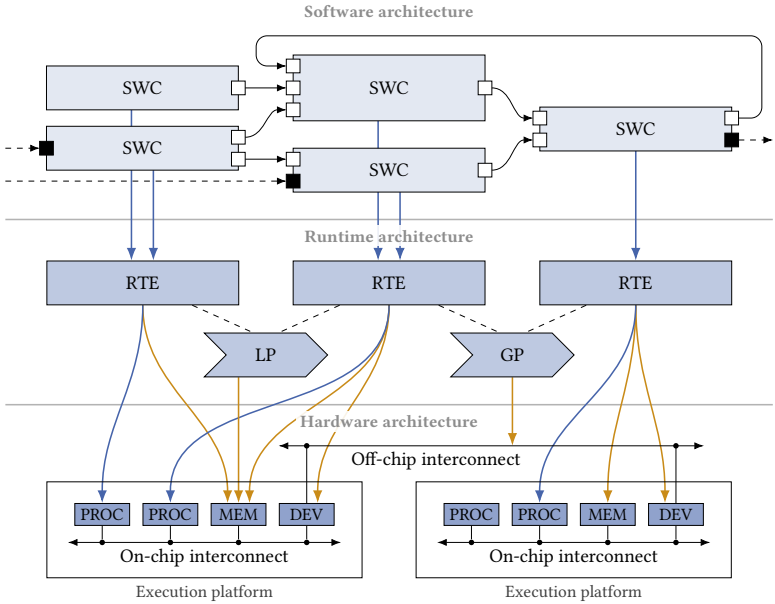


Figure 3.7: System model instance composed of 5 SWCs, 15 SWC ports, 3 RTEs, two communication paths, two execution platforms, and one off-chip interconnect.

the system model with instances of three additional model entities described in Table 3.2: processing units, memory modules, and peripheral devices. These instances become an implicit part of the overall system model and can be referenced by explicitly described entities.

▹ Remark 3.5: *In the remainder of this work, two of the acronyms from Table 3.1 will also be used from continuous text: Software Component (SWC) and Runtime Environment (RTE). Usage of other acronyms from both model entity tables is limited to diagrams, such as the one in Figure 3.7.*

Graphical visualization of a system model For illustration purposes, a sample instance of a system model is visualized in Figure 3.7. The depicted hardware architecture consists of two execution platforms, each instantiating two processing units, one memory module, and one peripheral device. At runtime level, three RTEs are each mapped to a dedicated processing unit. A

local path connects two RTEs mapped to the same execution platform, while a global path is used to connect two remote RTEs. At the software layer, five SWCs with a total of 15 ports are instantiated. Twelve of these ports are internal; they are drawn using an empty square and used to establish inter-SWC communication via a total of six channels. In addition, three environment ports, each depicted as a filled square, connect application software to the physical environment of the system.

Address space of execution platforms In the system model, each execution platform is associated with its own private address space. Memory modules and peripheral devices implement this address space, i.e., they serve as slave components. Processing units serve as master components and are generally capable of writing to or reading from any address. Selected peripheral devices, such as a DMA or an Ethernet controller, may also be equipped with a master port and therefore able to access the address space. For every execution platform, exactly one processing unit or peripheral device is defined to be the *address space origin*; this is the 'viewport' from which all specified addresses need to be interpreted. The physical entity that connects masters to slaves is a shared on-chip interconnect as shown in Figure 3.7. This shared on-chip interconnect may or may not be protected by an APU. Regardless of the APU protection status, selected master components might be able to circumvent this shared on-chip interconnect to access certain memory modules or peripheral devices. For example, a processing unit might have unrestricted access to a local memory module. If this is the case, then accesses by this processing unit to the local memory module might not be protected by a possible APU configuration.

3.2.1 Execution platform library

Before it is possible to formalize the system model, a specification of the execution platform library is necessary. This library is concerned with types of execution platforms rather than specific instances. It captures properties that all execution platform instances of a particular type have in common.

Since the execution platform library is provided by the safety pattern itself, the model entities covered in this section are—strictly speaking—not part of the system model. However, they are *referenced* from the system model and therefore an *implicit* part of it.

One property common to all execution platforms is that each memory module is mapped to a particular *memory region* from the *address space* of the

platform. To be able to reason about address spaces and memory regions, we formalize them using mathematical set theory:

► **Definition 3.7:** Address space and memory regions

The *address space* of execution platforms, $\mathbb{A}$, is the set of all physically possible memory addresses. A *memory region* is a sequential subset of $\mathbb{A}$:

$$[x, y]_{\mathbb{A}} = \{a \in \mathbb{A} : x \leq a \leq y\}, \quad x, y \in \mathbb{A}.$$

The set of all possible memory regions, $\mathbb{M}$, is given as follows:

$$\mathbb{M} = \{[x, y]_{\mathbb{A}} : x, y \in \mathbb{A}\}.$$

Since address spaces and memory regions are mathematical sets, conventional operations from set theory can be applied to them.

With this, the *execution platform type* is defined as follows:

► **Definition 3.8:** Execution platform type

An *execution platform type* is a tuple

$$\langle U,\ M,\ Q,\ Z,\ \Gamma_1,\ \Gamma_2,\ \Gamma_3,\ \hat{a},\ \mathrm{mmap}\rangle$$

with the following components:

1) The finite sets U, M, and Q represent processing units, memory modules, and peripheral devices, respectively.
2) $Z \subseteq Q$ represents peripheral devices equipped with a master port.
3) $\Gamma_1 \subseteq U \times M$ describes the memory modules that each processing unit can access without traversing the shared on-chip interconnect.
4) $\Gamma_2 \subseteq Q \times (U \cup M \cup Q)$ describes the platform components to which the failure of a peripheral device can directly propagate to.
5) $\Gamma_3 \subseteq Q$ represents peripheral devices whose failure can directly propagate to the entire underlying execution platform.
6) $\hat{a} \in U \cup Z$ is the address space origin, i.e., a reference master from which all specified addresses need to be interpreted.
7) $\mathrm{mmap}\colon M \to \mathbb{M}$ maps each $m \in M$ to its memory region. Memory regions do not overlap, i.e., the following condition holds for all $m_1, m_2 \in M$: $m_1 \neq m_2 \Rightarrow \mathrm{mmap}(m_1) \cap \mathrm{mmap}(m_2) = \varnothing$.

Collectively, elements from U, M, and Q are the *platform components* of an execution platform type. They are also referred to as $C = U \cup M \cup Q$.

▹ Remark 3.6: *In the following, references to any element from $\mathbb{A}$ or $\mathbb{M}$ are always relative to â of the underlying execution platform. For the sake of brevity, this implicit dependency is not explicitly stated when such references are made.*

Execution platform types are combined into the library and can optionally be complemented with APU configuration generators:

► **Definition 3.9:** Execution platform library

The *execution platform library* is a tuple

$$\Omega = \langle T, \text{gen} \rangle$$

with the following components:

1) T is a set of execution platform types according to Definition 3.8.
2) gen is a collection of procedures to generate APU configuration code for selected execution platform types from T.

In the remainder of this thesis, it will occasionally be necessary to refer to a particular tuple element of an execution platform type $t \in T$. To achieve this, the relevant tuple element (such as U or the 'mmap' function) is spelled out and suffixed with the underlying t in parentheses. $U(t)$, for example, refers to the processing units that are part of execution platform type $t \in T$. Since this notation is identical to that of a function call, two consecutive pairs of parentheses can appear when the 'mmap' function is first referenced and then called. $\text{mmap}(t)(m)$, for example, returns the memory region that the memory module $m \in M(t)$ is mapped to in $t \in T$.

▹ Example 3.5: *This framework is now used to model a certain portion of the Zynq UltraScale+ MPSoC as execution platform type. Figure 3.8 shows nine selected system entities to be represented: the two CPUs that were introduced in Example 2.3 and seven slave components from Table 2.2. To model the two CPUs as processing units, the following definition is made:*

$$U = \{u_{A53}, u_{R5}\}.$$

$u_{A53} \in U$, for instance, describes that in every execution platform instance of this type, a Cortex-A53 processor has access to the shared address space.

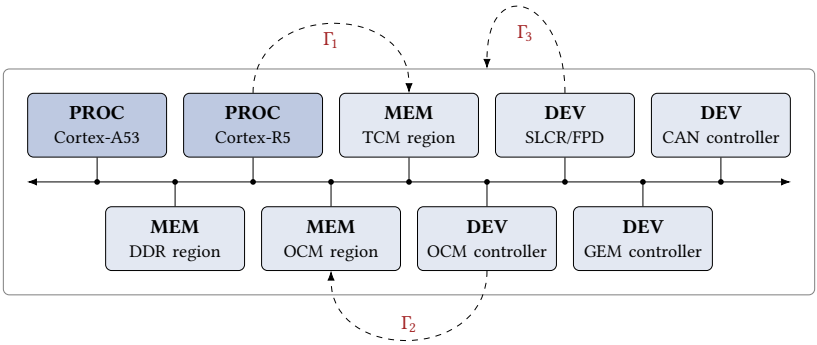


Figure 3.8: Execution platform type from selected Zynq UltraScale+ MPSoC components. Dashed arrows represent elements of Γ_1, Γ_2, and Γ_3, respectively.

$m \in M$ Memory module	$\text{mmap}(m)$ Memory region mapping
m_{DDR}	$[0\text{x}0, 0\text{x}7\text{FFFFFFF}]_{\mathbb{A}}$
m_{TCM}	$[0\text{xFFE00000}, 0\text{xFFE1FFFF}]_{\mathbb{A}}$
m_{OCM}	$[0\text{xFFFC0000}, 0\text{xFFFFFFFF}]_{\mathbb{A}}$

Table 3.3: 'mmap' definition for three memory modules of the Zynq UltraScale+ MPSoC. This assignment assumes that the Cortex-R5 is operated in lockstep mode and, therefore, that the modeled TCM is one unified memory of 128 KiB.

Afterward, the memory modules from Figure 3.8 are modeled:

$$M = \{m_{DDR}, m_{OCM}, m_{TCM}\}.$$

They represent read-write memory provided by external DDR, the OCM, and the TCM of the Cortex-R5, respectively. m_{TCM} *is simply represented as another module attached to the on-chip interconnect. This reflects the property that it is accessible from every master component in the system. The fact that it is actually part of the Cortex-R5, which can bypass the on-chip interconnect to access the TCM region, is modeled by choosing* $\Gamma_1 = \{\langle u_{R5}, m_{TCM}\rangle\}$. *To complete the memory region model, the address space origin is set to* $\hat{a} = u_{A53}$, *and* 'mmap' *values are derived from knowledge given in Table 2.2. As seen from the Cortex-A53, for*

example, the lower DDR region starts at address 0x0 and has a size of 2 GiB. The full derived table of 'mmap' *values is given as Table 3.3.*

Finally, to reflect the four represented types of peripheral devices, the peripheral device descriptor Q is chosen as follows:

$$Q = \{q_{CAN}, q_{GEM}, q_{OCM\text{-}CTRL}, q_{SLCR\text{-}FPD}\}.$$

As described on page 15, the Ethernet (GEM) controller integrates DMA functionality and is therefore equipped with a master port that grants it access to the address space of the device. We choose $Z = \{q_{GEM}\}$ to reflect this capability.

To be able to define Γ_2 and Γ_3, one must consider the indirect effects that write transactions to peripherals devices can have. By writing to the OCM control register $q_{OCM\text{-}CTRL}$, relevant characteristics of the OCM module may be affected. By writing to the SLCRs of the FPD, which are modeled as $q_{SLCR\text{-}FPD}$, it is possible to interfere with platform-level configurations. $\Gamma_2 = \{\langle q_{OCM\text{-}CTRL}, m_{OCM}\rangle\}$ and $\Gamma_3 = \{q_{SLCR\text{-}FPD}\}$ capture these relationships.

3.2.2 Hardware architecture (layer I)

An accurate model of relevant hardware properties is the first input required by the logical isolation pattern. This section describes a hardware architecture metamodel that was specifically developed to facilitate this goal.

The metamodel is inspired by the hardware abstraction of AADL [138] and a refined version of work the author previously published in [3]. Compared with AADL and the previous work, a key characteristic of the following metamodel is its automatic instantiation of execution platforms. For a given system, these instances are represented as follows:

► **Definition 3.10:** Platform specification

The *platform specification* is a tuple

$$\phi_X = \langle X, \text{type}\rangle$$

with the following components:

1) X is a finite set of execution platform instances.
2) $\text{type}: X \to T$ maps each execution platform instance to its type.

Each $x \in X$ is comparable to an *execution platform system* from AADL. According to the AADL standard [138], hardware can be clustered into "execution platform systems, i.e., into systems of execution platform components to model complex physical computing hardware [...]."

As it was informally described in Table 3.2, the instantiated execution platform components are processing units, memory modules, and peripheral devices. In the following, three functions will be used to refer to them:

► **Definition 3.11:** Instantiated platform components

Given a platform specification $\phi_X = \langle X, \text{type} \rangle$, the functions

$$\begin{aligned} \text{procs}(x) &= \{u^x : u \in U(\text{type}(x))\}, \ x \in X \\ \text{mems}(x) &= \{m^x : m \in M(\text{type}(x))\}, \ x \in X \\ \text{devs}(x) &= \{q^x : q \in Q(\text{type}(x))\}, \ x \in X \end{aligned}$$

map an execution platform to its processing units, memory modules, and peripheral devices, respectively. For each $x \in X$, the returned values are collectively referred to as the *instantiated platform components* of x.

Mathematically, all instantiated platform components are distinct objects, even if they originate from the same $u \in U$, $m \in M$, or $q \in Q$.

For the sake of brevity, the complete set of (instantiated) processing units, memory modules, and peripheral devices in a system will be abbreviated as

$$\hat{U} = \bigcup_{x \in X} \text{procs}(x), \quad \hat{M} = \bigcup_{x \in X} \text{mems}(x), \quad \text{and} \quad \hat{Q} = \bigcup_{x \in X} \text{devs}(x),$$

respectively. The union of these three sets,

$$\hat{C} = \hat{U} \cup \hat{M} \cup \hat{Q},$$

refers to all instantiated platform components in the system. In addition,

$$\hat{Z} = \{z^x : z \in Z(\text{type}(x))\} \subseteq \hat{Q}$$

is used to refer to those instantiated peripheral devices that are (also) a master of their respective on-chip interconnect.

Off-chip interconnects, such as a CAN bus, can be used to connect peripheral devices of several execution platforms. From a physical point of

view, such a connection is only feasible if used peripheral devices are I/O controllers of the correct type and if their external receive/transmit signals are actually accessible from outside. For example, a CAN bus [145] needs to interface CAN controllers whose receive and transmit signals are accessible via a transceiver. The following model feature formalizes off-chip interconnects:

► Definition 3.12: Bus specification

The *bus specification* is a tuple

$$\phi_B = \langle B, \text{controllers} \rangle$$

with the following components:

1) B is a finite set of off-chip interconnects.
2) $\text{controllers}\colon B \to \mathscr{P}(\hat{Q})$ maps each off-chip interconnect to the peripheral devices that are attached to it. Every peripheral device is associated with up to one off-chip interconnect.

▹ Remark 3.7: *The constraint that peripheral devices are associated with no more than one off-chip interconnect can also be expressed as follows:*

$$\forall \hat{q} \in \hat{Q}\colon |\{b \in B : \hat{q} \in \text{controllers}(b)\}| \leq 1.$$

For the sake of readability, however, Definition 3.12 describes this requirement textually. All following definitions in this chapter follow the same pattern, i.e., they formulate constraints textually rather than mathematically.

Instantiated peripheral devices with a master port (e.g., a DMA controller) can allocate memory regions. This is formalized as follows:

► Definition 3.13: Device configuration

The *device configuration* is a function

$$\phi_D = \text{malloc}\colon \hat{Z} \to \mathscr{P}(\mathbb{M}),$$

which maps each $\hat{z} \in \hat{Z}$ to an arbitrary number of exclusively owned memory regions, each taken from exactly one memory module of the execution platform that $\hat{z}$ is instantiated by.

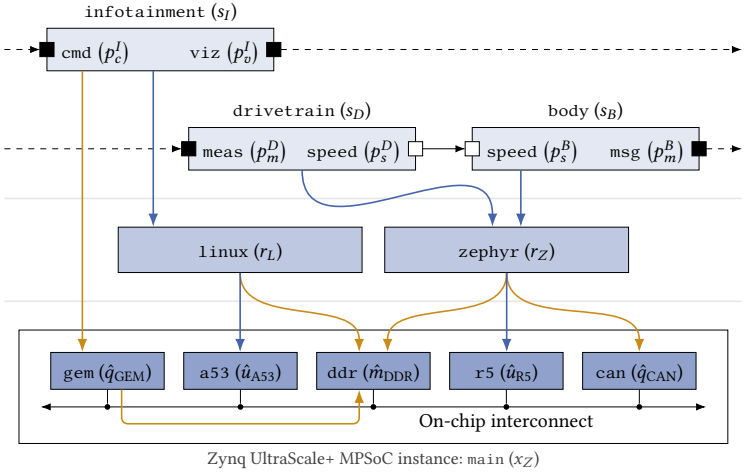


Figure 3.9: System model for a central car server that integrates infotainment, drivetrain, and body functions on two processors of the Zynq UltraScale+ MPSoC.

A memory region owned by a peripheral device must not be owned by any other system entity. Such memory regions are said to be *allocated* by their respective owner, and they become relevant when the peripheral devices are (by themselves) allocated to RTEs or SWCs: each RTE and each SWC will automatically obtain read/write access to the memory regions owned by their peripheral devices. For example, a SWC allocating a particular $\hat{z} \in \hat{Z}$ can use the memory regions from malloc($\hat{z}$) to exchange data with $\hat{z}$.

The combination of the three partial hardware models leads to the hardware architecture, i.e., to layer I of the full system model:

► **Definition 3.14:** Hardware architecture

A *hardware architecture* is a tuple $\langle \phi_X, \phi_B, \phi_D \rangle$, where

1) ϕ_X is a platform specification according to Definition 3.10,
2) ϕ_B is a bus specification according to Definition 3.12, and
3) ϕ_D is a device configuration according to Definition 3.13.

It implicitly contains the sets $\hat{U}$, $\hat{M}$, $\hat{Q}$, and $\hat{Z}$.

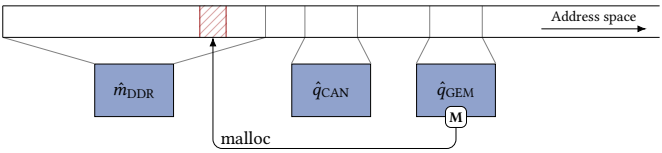


Figure 3.10: Schematic visualization of the global address space that x_Z implements in the car server example (not to scale). Using the 'malloc' function, a DDR memory region is allocated to $\hat{q}_{\text{GEM}}$.

▹ Example 3.6: *This is the first in a sequence of examples that collectively model the central car server use case shown in Figure 3.9.*

At the hardware layer, the system employs a Zynq UltraScale+ MPSoC, i.e., an instance of the execution platform type that was modeled in Example 3.5. Referring to this type as $t_Z \in T$, we model this by choosing

$$X = \{x_Z\} \quad \textit{and} \quad \text{type} = \{\langle x_Z,\, t_Z \rangle\}.$$

This enriches the model with nine instantiated platform components (five of which are shown in Figure 3.9). Since this example scenario does not comprise any off-chip communication, the bus specification is defined by choosing

$$B = \varnothing \quad \textit{and} \quad \text{controllers} = \varnothing.$$

As illustrated in Figure 3.10, a portion of DDR memory shall be allocated to the Ethernet controller ($\hat{q}_{GEM}$). Without loss of generality, we choose

$$\text{malloc} = \{\langle \hat{q}_{GEM},\, \{[\texttt{0x3C00000},\, \texttt{0x43FFFFF}]_{\mathbb{A}}\}\rangle\}$$

to allocate 8 MiB of memory, which is later to be shared between $\hat{q}_{GEM}$ and all entities that allocate $\hat{q}_{GEM}$. This concludes the hardware architecture model.

When this example is continued, the five relevant platform components will be referred to as indicated in Figure 3.9, e.g., by using $\hat{u}_{A53}$ to refer to the Cortex-A53 of the instantiated execution platform.

3.2.3 Runtime architecture (layer II)

The purpose of the runtime architecture is to execute all SWCs of a system. This includes the task of providing every SWC with required hardware resources, such as CPU time or memory. Furthermore, it is responsible for the implementation of inter-SWC communication; in this sense, it is comparable to

the Virtual Function Bus (VFB) concept from AUTOSAR [146] or the federated execution of a Lingua Franca (LF) program [147].

The runtime architecture model does not dictate how exactly these tasks are implemented in a given system. However, it assumes certain invariants that an implementation must fulfill. These invariants are built around the notion of the RTE, which is infrastructure software typically based on OS (such as Linux, Zephyr, …) or a hypervisor (such as XtratuM, Jailhouse, …).

► **Definition 3.15:** RTE specification

Based on a hardware architecture, the *RTE specification* is a tuple

$$\phi_R = \langle R, \text{proc}, \text{malloc}', \text{qalloc}' \rangle$$

with the following components:

1) R is a finite set of Runtime Environments (RTEs).
2) $\text{proc}: R \to \hat{U}$ maps each RTE to its dedicated processing unit.
3) $\text{malloc}': R \to \mathscr{P}(\mathbb{M})$ maps each RTE to a set of exclusively owned memory regions, each taken from exactly one memory module of the underlying execution platform.
4) $\text{qalloc}': R \to \mathscr{P}(\hat{Q})$ maps each RTE to a set of allocated peripheral devices, each from the underlying execution platform.

As described in item 2, each RTE operates on a dedicated processing unit. In other words, processing units host no more than one RTE.

The allocation described in item 3 maps each RTE to an arbitrary number of memory regions of the underlying execution platform. These memory regions are where an RTE stores instructions and data, both for its own and the operation of SWCs that it is responsible for.

The function in item 4 allows users to allocate peripheral devices to RTEs. The same peripheral device may be allocated to more than one system entity. In the context of this function, there are two important rules to understand:

1) As it was already described in Section 3.2.2, allocating a peripheral device with a master port ($\hat{z} \in \hat{Z}$) to an RTE means that this RTE will have read/write access to the memory owned by the peripheral device.
2) In addition, there is an analogous rule into the opposite direction: allocating a peripheral device with a master port ($\hat{z} \in \hat{Z}$) to an RTE means

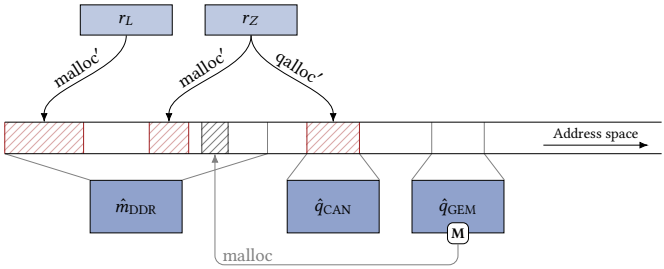


Figure 3.11: Allocation of two memory regions and one peripheral device to the RTEs of the car server example. The depicted address space is again not to scale.

that the peripheral device will have read/write access to the memory region owned by the RTE.

▹ Remark 3.8: *The number of apostrophes added to allocation functions (such as* malloc′ *and* qalloc′*) reflects the number of layers between the considered entity and the hardware architecture. This is why the memory allocation function in Definition 3.13 was suffixed with zero apostrophes, while the allocation functions in Definition 3.15 are suffixed with one apostrophe each.*

▹ Example 3.6 (continued): *The runtime architecture shown in Figure 3.9 is based on two operating systems. They are modeled as*

$$R = \{r_L, r_Z\},$$

where r_L represents a Linux distribution and r_Z a Zephyr instance. They are mapped to dedicated processing units by choosing

$$\text{proc} = \{\langle r_L, \hat{u}_{A53}\rangle, \langle r_Z, \hat{u}_{R5}\rangle\}.$$

As illustrated in Figure 3.11, each RTE allocates exactly one memory region from the DDR of the underlying execution platform. Without loss of generality, we select one 24 MiB and one 12 MiB wide region, which leads to

$$\begin{aligned}\text{malloc}' = \big\{&\langle r_L, \{[\texttt{0x0}, \texttt{17FFFFF}]_{\mathbb{A}}\}\rangle,\\ &\langle r_Z, \{[\texttt{0x2C00000}, \texttt{0x37FFFFF}]_{\mathbb{A}}\}\rangle\big\}.\end{aligned}$$

To grant r_Z access to the CAN controller of the platform, we further choose

$$\text{qalloc}' = \{\langle r_L, \varnothing\rangle, \langle r_Z, \{\hat{q}_{CAN}\}\rangle\},$$

which concludes this part of the example.

If two SWCs mapped to different RTEs need to interact, these RTEs need to collaborate to implement this communication. The resources they use to implement this communication are represented as either a local or a global path, both of which are covered by the following model artifact:

► **Definition 3.16:** Path specification

Based on a hardware architecture, the *path specification* is a tuple

$$\phi_Y = \langle L,\, G,\, \text{endpoints},\, \text{impl},\, \text{bus} \rangle$$

with the following components:

1) L is a finite set of local paths.
2) G is a finite set of global paths.
3) endpoints: $L \cup G \to \{K \subseteq R : |K| = 2\}$ maps each path to the unique unordered pair of RTEs connected by this path. A local path connects two RTEs from the same execution platform, while the RTEs of a global path are from different execution platforms.
4) impl: $L \to \mathscr{P}(\mathbb{M})$ maps each local path to a set of exclusive memory regions used to implement this path. Each region is part of a memory module of the underlying execution platform.
5) bus: $G \to B$ maps each global path to the off-chip interconnect used to implement it. Every endpoint of a path must allocate an I/O controller attached to the path's interconnect.

Every specified path—regardless of whether it is a local or a global one—is defined between an unordered pair of RTEs. For a given path, these RTEs are referred to as its endpoints. As described in item 3 above, endpoints must be unique, i.e., there may only be one path for the same pair of RTEs.

A local path describes memory regions that the two endpoints of this path (i.e., the two connected RTEs) utilize to exchange necessary messages. To do so, these RTEs have read/write access to all memory regions owned by this path. According to item 4 of the above definition, memory regions allocated to a local path are exclusively owned by it, i.e., they must not be allocated to another system entity (such as an RTE, another local path, …).

Global paths are implemented by exactly one off-chip interconnect (which, in turn, may be used to implement more than one global path). The communication between an RTE and this off-chip interconnect is achieved via I/O controllers, which must already be allocated to each endpoint via the qalloc′ func-

tion from Definition 3.15. The precise I/O controller that an RTE uses to communicate with a specific global path is not relevant for the purposes of this work and, therefore, not reflected in the model.

▹ Example 3.6 (continued): *SWCs in the car server example from Figure 3.9 do not communicate across RTE boundaries. Therefore, inter-SWC communication can always be handled without leveraging local or global paths. In the system model, we reflect this by choosing*

$$L = G = \varnothing,$$

which automatically implies

$$\text{endpoints} = \varnothing, \quad \text{impl} = \varnothing, \quad \textit{and} \quad \text{bus} = \varnothing.$$

This concludes the path specification.

Finally, the combination of specified RTEs, local paths, and global paths is combined to layer II of the full system model:

► **Definition 3.17:** Runtime architecture

The *runtime architecture* is a tuple $\langle \phi_R, \phi_Y \rangle$, where

1) ϕ_R is an RTE specification according to Definition 3.15 and
2) ϕ_Y is a path specification according to Definition 3.16.

It implicitly contains the underlying hardware architecture.

3.2.4 Software architecture (layer III)

At the highest level of abstraction, a software architecture describes relevant application properties in the form of SWCs and their interactions.

The model does not dictate how a software architecture is implemented, but it defines several invariants that an implementation must conform to. One such invariant is that SWCs are conceptually concurrent processes executed by exactly one RTE. They communicate via asynchronous input and output interfaces called ports. SWCs use ports to interact with each other or to access the physical environment. Emitting data via an output port does not affect the future behavior of its SWC; this implies that write access is non-blocking, which is in line with sender-receiver communication in AUTOSAR [146]. SWCs may

allocate peripheral devices, e.g., to write to the physical environment of the system. Formally, they are defined as follows:

► Definition 3.18: SWC specification

Based on a runtime architecture, the *SWC specification* is a tuple

$$\phi_S = \langle S,\ \mathrm{rte},\ \mathrm{qalloc}'' \rangle$$

with the following components:

1) S is a finite set of Software Components (SWCs).
2) $\mathrm{rte} : S \to R$ maps each SWC to the RTE it is executed on.
3) $\mathrm{qalloc}'' : S \to \mathscr{P}(\hat{Q})$ maps each SWC to a set of allocated peripheral devices, each from the underlying execution platform.

In practice, a SWC will often be realized as a process or task of the OS that is used to implement the underlying RTE.

The ability of SWCs (or processes, tasks, …) to allocate peripheral devices is captured in item 3 of the above definition; it is analogous to the identically named capability of RTEs (cf. Definition 3.15). As described in Section 3.2.2, allocating a peripheral device with a master port ($\hat{z} \in \hat{Z}$) to a SWC has a semantic implication: it means that this SWC will also receive read/write access to the memory regions that are owned by this peripheral device, i.e., owned by $\hat{z}$. Since a SWC may not allocate memory regions, there is no analogous rule into the opposite direction (as there was in the RTE case).

▹ Example 3.6 (continued): *As it was shown in Figure 3.9, the car server example defines three SWCs: one controlling an infotainment system (s_I), another one implementing drivetrain control functions (s_D), and a third one responsible for body control tasks (s_B). In the model, they are defined by choosing*

$$S = \{s_I,\ s_D,\ s_B\}.$$

The infotainment controller is executed by the Linux runtime (r_L), while the Zephyr system (r_Z) hosts the two other SWCs. This assignment is modeled by

$$\mathrm{rte} = \{\langle s_I,\ r_L\rangle,\ \langle s_D,\ r_Z\rangle,\ \langle s_B,\ r_Z\rangle\}.$$

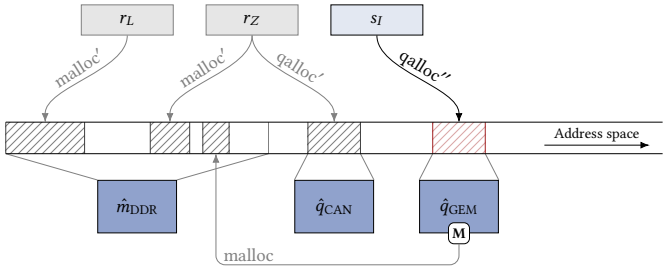


Figure 3.12: Extended resource mapping for the car server example. To complement RTE-based allocations, s_I has access to the Ethernet controller.

Following the structure from Figure 3.9, the infotainment controller requires access to the Ethernet controller ($\hat{q}_{GEM}$). The other SWCs do not need to allocate peripheral devices. In combination, this is reflected by choosing

$$\text{qalloc}'' = \{\langle s_I, \{\hat{q}_{GEM}\}\rangle, \langle s_D, \varnothing\rangle, \langle s_B, \varnothing\rangle\}.$$

This finally leads to the allocations shown in Figure 3.12. Note that the arrows in this visualization show only allocations, not accesses to the address space. The infotainment controller, s_I, will additionally have access to the memory region owned by $\hat{q}_{GEM}$, for example.

Ports are used to model all interactions of SWCs—both internal and external ones. In the software architecture model, they are formalized as follows:

► **Definition 3.19:** Port specification

The *port specification* is a tuple

$$\phi_P = \langle P, \text{swc}, \text{scope}, \text{dir}\rangle$$

with the following components:

1) P is a finite set of SWC ports.
2) $\text{swc}: P \to S$ maps each port to the SWC it is part of.
3) $\text{scope}: P \to \{\text{Int}, \text{Env}\}$ specifies the scope of each port.
4) $\text{dir}: P \to \{\text{In}, \text{Out}\}$ specifies the data flow direction of each port.

A port is part of exactly one SWC, which is covered by item 2 of the above definition. The scope, which is described by the function in item 3, captures

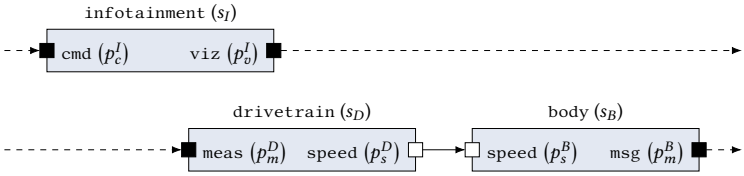


Figure 3.13: Software architecture of the car server example in Figure 3.9.

whether a port exchanges messages with other SWCs (scope(·) = Int) or interacts with the physical environment of the system (scope(·) = Env). The port direction, which is defined by item 4, captures whether the port is used to receive (dir(·) = In) or transmit (dir(·) = Out) data.

▹ Remark 3.9: *If it is necessary to emphasize the scope of a port, the entity will be loosely referred to as either an 'internal port' or 'environment port' in the following. Analogously, the terms 'input' and 'output' will be used to highlight the direction of a particular port. If applicable, this terminology is occasionally combined, for instance to refer to an 'environment output' of a SWC.*

Internal communication (between SWCs) is represented in the form of channels, whose model representation is the following relation:

► **Definition 3.20:** Channel specification

The *channel specification* is a relation

$$\phi_H \subseteq P \times P$$

of logical data paths, each referred to as a channel.

$\langle p, p' \rangle \in \phi_H$ implies that p and p' are SWC ports with internal scope, i.e., neither p nor p' is an environment port. It further means that p has output direction and p' has input direction. Output p is said to *write* to the channel, while input p' is *written* by the channel.

SWCs whose ports are connected via a channel must be mapped to the same RTE or, alternatively, there must be a local or global path that connects the RTEs executing the SWCs (cf. Definition 3.16).

	$p \in P$					
Specifier	p_c^I	p_v^I	p_m^D	p_s^D	p_s^B	p_m^B
swc(p)	s_I	s_I	s_D	s_D	s_B	s_B
scope(p)	Env	Env	Env	Int	Int	Env
dir(p)	In	Out	In	Out	In	Out

Table 3.4: Port specification for the software architecture in Figure 3.13.

▹ Example 3.6 (continued): *Using these definitions, it is now possible to complete the model of the car server from Figure 3.9. For convenience, the software architecture of this system is replicated in Figure 3.13. It comprises a total of six SWC ports, exactly two per SWC. For the purposes of this example, the ports will be referred to as p_x^y, where x describes the data exchanged via a port and $y \in \{I, D, B\}$ is used to associate it with the respective SWC that it is part of.*

The infotainment controller (s_I) is a standalone SWC. Its environment input p_c^I reads user commands from the physical environment, while the environment output p_v^I visualizes requested content on a display. The drivetrain controller (s_D) reads sensor measurements via p_m^D, processes them to derive the vehicle speed, and distributes this value via the internal output p_s^D. The operation of the body controller (s_B), in turn, is based on the vehicle speed that it expects to read via the internal input p_s^B. To activate the central locking system when a threshold speed is exceeded, for instance, it writes to the body network modeled as environment output p_m^B. This leads to $P = \{p_c^I, p_v^I, p_m^D, p_s^D, p_s^B, p_m^B\}$ and the function assignments given in Table 3.4. Finally, $\phi_H = \{\langle p_s^D, p_s^B \rangle\}$ reflects the channel that forwards calculated vehicle speeds to s_B.

The combination of all model entities described in this section is the software architecture, i.e., layer III of the full system model:

► **Definition 3.21:** Software architecture

The *software architecture* is a tuple $\langle \phi_S, \phi_P, \phi_H \rangle$, where

1) ϕ_S is a SWC specification according to Definition 3.18,
2) ϕ_P is a port specification according to Definition 3.19, and
3) ϕ_H is a channel specification as described in Definition 3.20.

It implicitly contains the underlying runtime architecture.

Since the software architecture is the highest abstraction layer covered by this work, Definition 3.21 concludes the formal system metamodel.

3.2.5 Auxiliary functions

Sets, functions, and relations directly embedded into the system model are populated by users of the safety pattern. To be able to reason about a system model, it is beneficial to complement them with auxiliary functions that query certain details. Eleven such functions are defined below.

3.2.5.1 Execution platform lookup

The function '$\mathrm{pf}\colon \hat{C} \to X$' maps instantiated platform components to their execution platform. It is defined as follows:

$$\mathrm{pf}(\hat{c}) = \begin{cases} x \in X : \hat{c} \in \mathrm{procs}(x), & \hat{c} \in \hat{U}, \\ x \in X : \hat{c} \in \mathrm{mems}(x), & \hat{c} \in \hat{M}, \\ x \in X : \hat{c} \in \mathrm{devs}(x), & \hat{c} \in \hat{Q}. \end{cases}$$

Based on this, the function '$\mathrm{pf}'\colon R \to X$' performs a two-step lookup to return the execution platform of a particular RTE:

$$\mathrm{pf}'(r) = \mathrm{pf}(\mathrm{proc}(r)).$$

Moving one step further away from the hardware layer, the auxiliary function '$\mathrm{pf}''\colon L \to X$' determines the platform of a given local path:

$$\mathrm{pf}''(\ell) = \mathrm{pf}'(r \in R : r \in \mathrm{endpoints}(\ell)).$$

3.2.5.2 SWC-related lookups

It can be useful to query all SWCs executed by a particular RTE. The auxiliary function '$\mathrm{swcs}\colon R \to \mathscr{P}(S)$' achieves this as follows:

$$\mathrm{swcs}(r) = \{s \in S : \mathrm{rte}(s) = r\}.$$

To query SWC input and SWC output ports of a particular SWC, the functions '$\mathrm{inputs}\colon S \to \mathscr{P}(P)$' and '$\mathrm{outputs}\colon S \to \mathscr{P}(P)$' are defined as follows:

$$\mathrm{inputs}(s) = \{p \in P : \mathrm{swc}(p) = s \wedge \mathrm{dir}(p) = \mathrm{In}\},$$
$$\mathrm{outputs}(s) = \{p \in P : \mathrm{swc}(p) = s \wedge \mathrm{dir}(p) = \mathrm{Out}\}.$$

3.2.5.3 Port-related lookups

It can be necessary to determine the RTE managing a particular SWC port. The function ‘$\mathrm{rte}' : P \to R$’ achieves this as follows:

$$\mathrm{rte}'(p) = \mathrm{rte}(\mathrm{swc}(p)).$$

Based on this, the function ‘$\mathrm{sinks} : (L \cup G) \to \mathscr{P}(P)$’ returns all SWC ports that may read values from a local or global path. It is defined as follows:

$$\mathrm{sinks}(y) = \{p' \in P : \langle p, p' \rangle \in \phi_H \wedge \mathrm{endpoints}(y) = \{\mathrm{rte}'(p), \mathrm{rte}'(p')\}\}.$$

3.2.5.4 Allocation-related lookups

Since the memory region implemented by an instantiated memory module is not a direct property of this instance, it takes several intermediate steps to obtain this knowledge from the underlying platform type. To simplify this process, we define the function ‘$\mathrm{region} : \hat{M} \to \mathbb{M}$’ as follows:

$$\mathrm{region}(\hat{m}) = \mathrm{mmap}(t)(m),$$

where $\hat{m} = m^x$ such that $x \in X$, $t = \mathrm{type}(x)$, and $m \in M(t)$.

In total, three entities from the system model are able to allocate memory regions: peripheral devices with a master port, RTEs, and local paths. For each such entity, it can be necessary to query the instantiated memory modules that implement these regions. The function ‘$\mathrm{mems}' : (\hat{Z} \cup R \cup L) \to \mathscr{P}(\hat{M})$’ achieves this as follows:

$$\mathrm{mems}'(v) = \begin{cases} \{\hat{m} \in \mathrm{mems}(\mathrm{pf}(v)) : \mathrm{contains}(\hat{m}, \mathrm{malloc}(v))\}, & v \in \hat{Z}, \\ \{\hat{m} \in \mathrm{mems}(\mathrm{pf}'(v)) : \mathrm{contains}(\hat{m}, \mathrm{malloc}'(v))\}, & v \in R, \\ \{\hat{m} \in \mathrm{mems}(\mathrm{pf}''(v)) : \mathrm{contains}(\hat{m}, \mathrm{impl}(v))\}, & v \in L, \end{cases}$$

where $\mathrm{contains}(\hat{m}, \alpha) = (\exists \mu \in \alpha : \mu \subseteq \mathrm{region}(\hat{m}))$, i.e., a predicate that evaluates to true if and only if the instantiated memory module $\hat{m}$ implements at least one of the memory regions contained in α.

For consistency, we also define ‘$\mathrm{devs}' : (R \cup S) \to \mathscr{P}(\hat{Q})$’ to return the peripheral devices that are allocated to an entity:

$$\mathrm{devs}'(v) = \begin{cases} \mathrm{qalloc}'(v), & v \in R, \\ \mathrm{qalloc}''(v), & v \in S. \end{cases}$$

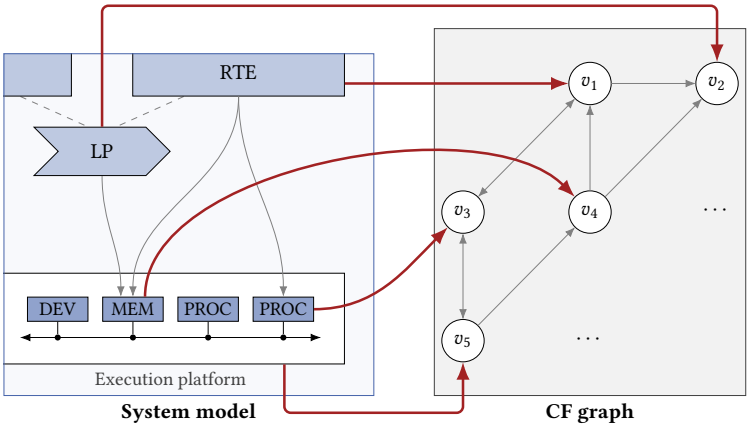


Figure 3.14: Mapping of selected entities from a generic system model to vertices of the CF graph, i.e., to system elements according to Definition 3.2.

3.3 Fault model for system elements

To close this chapter, the CF determination concept from Section 3.1 is connected to the formal system model from Section 3.2. The former is a procedure to reason about the potential for CFs, while the latter is a descriptive framework that allows users of the pattern to describe relevant system knowledge. Therefore, entities from the formal system model do not necessarily have the correct granularity to conduct the CF determination.

3.3.1 System element mapping

Mapping system model entities to system elements, which were introduced in Definition 3.2, is the proposed approach to solve this mismatch. System elements are the vertices of the CF graph. As shown in Figure 3.14, various system model entities are directly represented as exactly one such vertex. In fact, this applies to all entities except for the following:

SWCs ($s \in S$): To analyze how a SWC contributes to the CF potential, decoupling its implementation from its application-level behavior is beneficial. Consider, for instance, that an erroneous algorithm is correctly translated to C code, compiled, and executed. In this case, the SWC implementation is correct, but the SWC logic is faulty. In the CF graph, this will

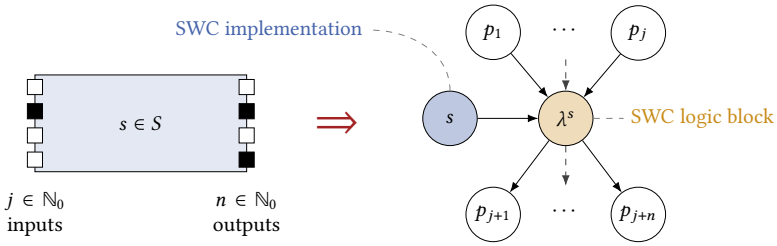


Figure 3.15: Decomposition of a SWC, $s \in S$, into a system element reflecting its implementation (s) and another one describing realized application logic (λ^s).

later be represented by decomposing this $s \in S$ into an implementation vertex (s) and a logic vertex (λ^s). Figure 3.15 visualizes this decomposition. As shown in Chapter 4, it can even be beneficial to decompose a SWC into $k \in \mathbb{N}^+$ logic vertices ($\lambda^s_1, \ldots, \lambda^s_k$). Together, these two kinds of vertices are the system elements derived from SWCs.

Channels ($\langle p, p' \rangle \in \phi_H$): Channels will later be represented as edges instead of CF graph vertices. Chapter 4 will show why this is a possible and straightforward solution. Despite their presence in the formal system model, they are therefore not system elements in the sense of Definition 3.2; this is consistent with the decision to represent channels as tuples of SWC ports instead of independent objects.

Instantiated platform components ($\hat{c} \in \hat{C}$): Since the set of instantiated platform components is automatically derived from the execution platform library, typical systems will use only a small subset of them. An instantiated platform component is a system element if and only if it is in use. The rationale for this is given in Section 4.1.1.

There is another aspect of the system model whose mapping necessitates a more detailed explanation: memory regions allocated to an RTE, a peripheral device with master access, or a local path. In CF graphs, such memory regions are represented by their respective owner:

- All memory regions $\mu \in \text{malloc}(\hat{z})$, $\hat{z} \in \hat{Z}$, are represented by $\hat{z}$.
- All memory regions $\mu \in \text{malloc}'(r)$, $r \in R$, are represented by r.
- All memory regions $\mu \in \text{impl}(\ell)$, $\ell \in L$, are represented by ℓ.

While it would be possible to represent these memory regions using dedicated vertices, the memory region and its owner would have to be connected using

System entity	System element	Fault susceptibility[a]	
		Systematic	Random
$x \in X$	Execution platform x	●	●
$\hat{u} \in \hat{U}$	Processing unit $\hat{u}$	●	●
$\hat{m} \in \hat{M}$	Memory module $\hat{m}$	●	●
$\hat{q} \in \hat{Q}$	Peripheral device $\hat{q}$	●[b]	●[b]
$b \in B$	Off-chip interconnect b	●	●
$r \in R$	Runtime environment r	●[b]	○
$\ell \in L$	Local path ℓ	○	○
$g \in G$	Global path g	○	○
$s \in S$	SWC implementation s	●	○
	SWC logic block $\lambda_1^s, \ldots, \lambda_k^s$	●	○
$p \in P$	SWC port p	(○ / ●)[c]	○

[a] Legend: ● yes (⇒ subject to systematic/random faults); ○ no (⇒ free of faults).
[b] Memory regions represented by this system element cannot become faulty. Faults of their underlying memory module, $\hat{m} \in \hat{M}$, are captured by the corresponding system element.
[c] Only for environment inputs. A fault of a $p \in P$ (with $\text{scope}(p) = \text{Env}$ and $\text{dir}(p) = \text{In}$) represents the possibility that p reads erroneous data from the environment.

Table 3.5: Susceptibility of each system element type to systematic and random faults, respectively. System elements with at least one 'yes' entry can become faulty.

bidirectional edges. Merging them into one system element is a practical approach to reduce the complexity of CF graphs.

3.3.2 Fault susceptibility

As described above, it is assumed that system elements may fail during system runtime. By definition, such a failure is the result of a fault manifestation. With the 11 types of system elements defined, it is now possible to describe the fault types that are covered by the scope of this work.

▹ Remark 3.10: *Here, the term 'fault' refers to only the root cause of a (potentially cascading) failure. The fact that a failure can in turn present as a fault to dependent system parts is not the topic of this section.*

In general, the pattern is able to track failures that are caused by both systematic and random faults. At the hardware architecture level, a design error in the processing unit of an execution platform is an example of a systematic

fault covered by the fault model. The well-known floating division bug of early Pentium microprocessors [148] is a real-world example of such a case. A single-event transients that manifests as a flipped bit in the register of a processing unit is an example of a random fault that can cause the system element to fail. As shown in Table 3.5, system elements at the hardware layer are assumed to be susceptible to both systematic and random faults.

Higher abstraction levels are software-based and can, by definition, not suffer from random faults. At the runtime architecture level, every RTE is considered to be susceptible to systematic faults; it might fail to schedule processes as required, for example. Local and global paths are inherently fault-free, as again shown in Table 3.5. This is a consequence of the fact that they are only 'virtual' elements realized by RTEs, memory modules, peripheral devices, and off-chip interconnects. They can fail in response to a failure of those elements, but they will never introduce a fault by themselves.

At the software architecture layer, SWC implementations and SWC logic are both susceptible to systematic faults. A faulty binary implementing a SWC might contain instructions that access memory regions not allocated to the SWC, for instance. SWC ports are, in principle, virtual elements that do not cause faults by themselves. Environment inputs are an exception, however: if there is the possibility that they introduce erroneous inputs into the system, this is interpreted as a systematic fault of the respective input port.

▹ Remark 3.11: *While environment inputs are concerned with the correctness of the inputs they read from external sources, environment outputs play a pivotal role in the safety assessment procedure. Chapter 5 will show that the safety requirements of environment outputs are a key factor in determining whether a specific CF potential is acceptable.*

This concludes the fault model of the logical isolation pattern. In the next chapter, the impact of failures that result from covered faults is investigated and reflected in the CF graph creation algorithm.

Chapter 4

APU configuration and CF determination procedure

The procedure described in this chapter is a novel approach to restrict and analyze the potential for CFs in multicore-based software systems. It forms the basis of the safety assessment strategy described in Chapter 5, but an application to other use cases is feasible. To emphasize its transferable nature, safety considerations are mostly omitted from this chapter.

As described in Section 3.1.1, the integrated procedure operates on two key inputs: a system model instance and a specification of isolation measures. Conceptually, the isolation measure specification is optional. Without it, the derived CF graph will be as pessimistic as it can be for a given system model. Specified isolation measures reduce this pessimism, for example by running APU configuration procedures, and need to be reflected in generated CF graphs. This observation motivates the structure of this chapter:

- Section 4.1 introduces the reader to CF graphs.
- Afterward, the meaning and treatment of isolation measures is covered in Section 4.2. This includes a description of the procedure to auto-generate APU configuration code.
- Section 4.3 closes this chapter with a thorough description of the CF determination process.

In combination, the latter two sections describe a repeatable procedure for the prevention and the analysis of CFs.

4.1 Introduction to CF graphs

Before the repeatable procedure is described, this section attempts to give the reader an intuitive understanding of CF graphs and the way they are influenced by isolation measures. As part of this, a graphical notation used to visualize CF graphs is introduced.

4.1.1 Structure and visualization

According to Definition 3.5, a CF graph is a directed graph in which a vertex $v \in V$ represents a system element and the edge set E contains a directed path from $v \in V$ to $v' \in V$ if and only if CF potential leads from v to v'.

4.1.1.1 CF graph vertices

As described in Section 3.3.1, the pattern considers eleven types of system elements. They originate either directly from the system model or, in the case of SWC logic blocks, are derived from it. The set of SWC logic blocks will be formally introduced in Section 4.2. Until then, this set is simply referred to as Λ^S. Based on this notation, the vertex set V can be described as follows:

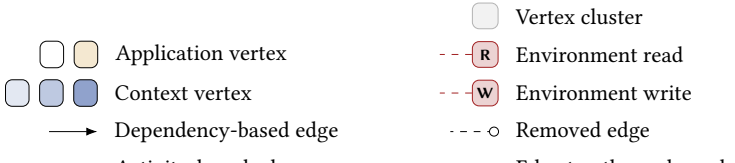
$$V \subseteq \underbrace{\Big(X \cup \overbrace{\big(\hat{U} \cup \hat{M} \cup \hat{Q}\big)}^{\text{Instantiated platform components } (\hat{C})} \cup\ B\Big)}_{\text{Hardware architecture}} \quad \cup \quad \underbrace{\big(R \cup L \cup G\big)}_{\text{Runtime architecture}} \quad \cup \quad \underbrace{\big(S \cup \Lambda^S \cup P\big)}_{\text{Software architecture}}.$$

This relation defines the set from which CF graph vertices are drawn. To prevent the CF graph from becoming unnecessarily complex, instantiated platform components that remain unused are deliberately excluded from V. The precise definition of this 'in use' property is given in Section 4.3. For now, the set of used components is simply referred to as

$$\hat{C}_+ = \{\hat{c} \in \hat{C} : \hat{c} \text{ is in use}\}$$

Based on this, it is possible to give an equation for the vertex set:

$$V = \underbrace{\big(X \cup \hat{C}_+ \cup B\big)}_{\text{Hardware architecture}} \quad \cup \quad \underbrace{\big(R \cup L \cup G\big)}_{\text{Runtime architecture}} \quad \cup \quad \underbrace{\big(S \cup \Lambda^S \cup P\big)}_{\text{Software architecture}}.$$



(a) Vertices and edges (b) Visualization helpers

Figure 4.1: Graphical notation used for the visualization of CF graphs.

To give CF graphs a clear and comprehensible structure, it is beneficial to partition V into two disjoint sets: the application vertices ($V_\alpha = \Lambda^S \cup P$) and the context vertices ($V_\beta = V \setminus V_\alpha$). Semantically, these two subsets of V differ in the system aspects they capture:

Application vertices ($v \in V_\alpha$): This vertex category represents either a port or a SWC logic block. Combined with channels, these are exactly the system elements that capture the nominal behavior of a system. This behavior is fully determined by *application* properties, which gives this class of CF graph vertices its name.

Context vertices ($v \in V_\beta$): The remaining nine types of system elements are unrelated to the nominal behavior of the considered system. As described in Section 3.2, for example, SWC implementation vertices ($s \in S$) are only concerned with implementation-related failures of a SWC. Like system elements from the runtime and the hardware architecture, these aspects are interpreted as the *context* of an application, which is again the aspect that gives this vertex class its name.

Figure 4.1a introduces the graphical notation used to visualize CF graphs. According to this legend, vertices are represented as rectangles with a black border. Different background colors indicate whether a vertex belongs to the application or the context portion of the graph.

To reduce the visual complexity of a CF graph visualization, its vertices are labeled using the textual naming scheme

$$\langle \text{elem} \rangle (\langle \text{identifier} \rangle),$$

where ⟨elem⟩ is a placeholder for the system element type and ⟨identifier⟩ is a descriptor to identify the specific system element of given type. The value

$\lambda^s \in \Lambda^S \Rightarrow$ logic	$p \in P \Rightarrow$ port

(a) Application vertices

$s \in S \Rightarrow$ swc	$r \in R \Rightarrow$ rte	$\hat{u} \in \hat{U} \Rightarrow$ proc
$\ell \in L \Rightarrow$ path	$x \in X \Rightarrow$ platform	$\hat{m} \in \hat{M} \Rightarrow$ mem
$g \in G \Rightarrow$ path	$b \in B \Rightarrow$ bus	$\hat{q} \in \hat{Q} \Rightarrow$ dev

(b) Context vertices

Figure 4.2: ⟨elem⟩ values used to label vertices of CF graphs.

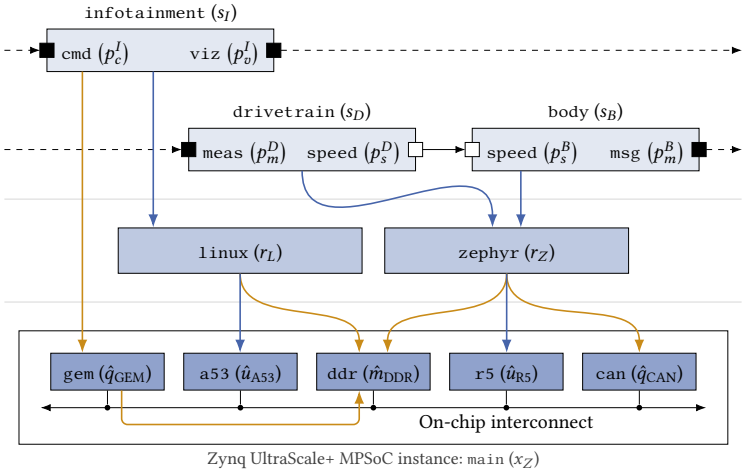


Figure 4.3: Central car server system from Example 3.6, here repeated for convenience.

of ⟨identifier⟩ is not directly captured in the formal system model from Section 3.2. The formal way of referring to a particular vertex is to use the respective symbol. In this thesis, however, entities of the system model are often associated with human-readable names, and these names are then used to construct the ⟨identifier⟩. The mapping from each system element to the associated ⟨elem⟩ expression is documented in Figure 4.2.

▹ Example 4.1: *The system in Figure 4.3, which is already known from Chapter 3, implements infotainment, drivetrain control, and body control features on a single execution platform (*`main`*). The functionality is partitioned in the sense that the SWC for infotainment tasks (*`infotainment`*) does not communicate*

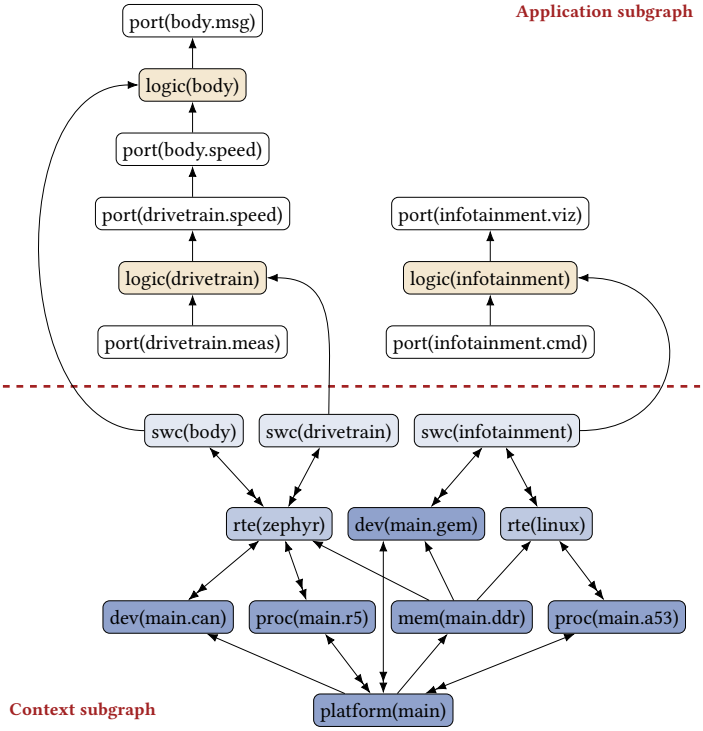


Figure 4.4: CF graph for the car server from Example 4.1. The dashed line separates application vertices (above) from context vertices (below).

to the other SWCs in the system (`drivetrain` *and* `body`*). The runtime-level implementation of this functionality is also partitioned, i.e., the two groups of SWCs are mapped to dedicated RTEs: a Linux distribution (*`linux`*) executed on the Cortex-A53 processor (*`main.a53`*) and a Zephyr instance (*`zephyr`*) mapped to the Cortex-R5 processor (*`main.r5`*). Each of the RTEs uses a DDR memory (*`main.ddr`*) region to store relevant code and data. To interact with the environment,* `infotainment` *allocates the Ethernet controller (*`main.gem`*). To communicate with the SWC, this controller in turn allocates a DDR memory region (see Figure 3.12). Environment interactions of the other SWCs are managed by the Zephyr instance, which allocates a CAN controller (*`main.can`*) to do so.*

Under the assumption that no isolation measures are applied, the CF graph for this system contains the 20 vertices shown in Figure 4.4.

▹ Remark 4.1: *Apart from manual fine-tuning of its layout, the CF graph in Figure 4.4 was automatically generated by the reference implementation of the pattern, which will later be the topic of Chapter 6. The same tool was used to auto-generate all following CF graph examples.*

4.1.1.2 CF graph edges

Edges of the CF graph are added between the following vertex pairs:

$$E \subseteq \big(V_\alpha \times V_\alpha\big) \cup \big(V_\beta \times V_\beta\big) \cup \big(V_\beta \times V_\alpha\big).$$

Edges from an application vertex ($v_\alpha \in V_\alpha$) to a context vertex ($v_\beta \in V_\beta$) do not occur, since failures cannot propagate into this direction. The following general statement about CF graph edges is true:

$$\langle v_1, v_2 \rangle \in E \Rightarrow \text{CF potential leads from } v_1 \text{ to } v_2.$$

The converse of this statement does not necessarily apply, i.e., not every CF potential is represented by a dedicated edge (cf. Section 4.3).

If there is an edge, however, it is assigned to one of two categories: dependency-based edges (E_δ) and activity-based edges (E_η). As in the vertex case, E is partitioned into these subsets, i.e., $E_\delta \cup E_\eta = E$ and $E_\delta \cap E_\eta = \varnothing$. The subsets differ in the semantics of contained edges as follows:

Dependency-based edges ($e \in E_\delta$): A directed edge $\langle v_1, v_2 \rangle \in E_\delta$ implies that a failure of v_1 can potentially trigger a CF of v_2, because v_2 is implemented by or implemented using v_1.

Activity-based edges ($e \in E_\eta$): A directed edge $\langle v_1, v_2 \rangle \in E_\eta$ implies that although $\langle v_1, v_2 \rangle \notin E_\delta$, a failure of v_1 can potentially trigger and perform an activity that causes v_2 to fail.

▹ Example 4.2: *If a given RTE uses a particular memory module to store its kernel data, a dependency-based edge from this module to the RTE is added.*

An activity-based edge occurs, for instance, if an RTE allocates a peripheral device. Since the RTE has full control over this peripheral device, it can potentially configure it in an erroneous manner.

In CF graphs, dependency-based edges are drawn with one arrow tip, while activity-based edges have two arrow tips (cf. Figure 4.1a).

▹ Example 4.1 (continued): *Assuming that no isolation measures are specified, the CF graph derived from Figure 4.3 contains $|E_\delta| = 25$ dependency-based and $|E_\eta| = 10$ activity-based edges, all of which are shown in Figure 4.4.*

▹ Remark 4.2: $\langle v_1, v_2 \rangle \in E$ *does not mean that* v_2 *is inherently faulty. It only means that in the considered context,* v_2 *may deviate from its intended functionality. Suppose that a CAN controller, which is modeled as a peripheral device, is free of systematic and random faults. If it is erroneously configured by the RTE, it can still fail in the context of the overall system.*

4.1.1.3 Visualization of CF graphs

CF graphs can be decomposed into two subgraphs: one induced by V_α and another induced by V_β. Following the notation from Section 2.2, these graphs will be referred to as $G_\gamma[V_\alpha]$ and $G_\gamma[V_\beta]$, respectively. Edges that lead from a context to an application vertex ($e \in V_\beta \times V_\alpha$) are dropped during this decomposition, i.e., they will not be included in any of the two subgraphs.

Induced subgraphs can be used to reason about CF potential in particular portions of a system. $G_\gamma[V_\alpha]$ captures the nominal behavior at the software level, while $G_\gamma[V_\beta]$ is concerned with the context that applications run in.

This decomposition is particularly useful for visualization purposes. It can reduce visual complexity and facilitate the human observer's understanding of important relationships between system elements.

Comprehensibility of CF graphs can further be supported by incorporating additional knowledge into their visualization. The graphical notation used to do so is introduced by the legend in Figure 4.1b on page 75. In the application subgraph, for example, clustering logic and port vertices of the same SWC can be beneficial and makes it possible to drop the SWC name from each individual vertex. In the context subgraph, clustering vertices that originate from the same model layer can simplify the interpretation of important relationships. Vertices in these clusters will occasionally be referred to as the software context, runtime context, and hardware context, respectively. Furthermore, to increase comprehensibility, it can make sense to add labels to those vertices from which outgoing context-to-application edges have been removed; by listing all application vertices to which a context vertex is actually connected, dropped edges ($e \in V_\beta \times V_\alpha$) become visible again.

▹ Example 4.1 (continued): *The two subgraphs extracted from Figure 4.4 are shown in Figure 4.5 and Figure 4.6, respectively.*

The application subgraph in Figure 4.5, for instance, contains a directed path from `port(drivetrain.meas)` *to* `port(body.msg)`*. This means that data from the* `meas` *input of* `drivetrain` *is potentially able to interfere with the* `msg` *output of the* `body` *component. The context subgraph in Figure 4.6 contains an activity-based edge from* `proc(main.a53)` *to* `platform(main)`.

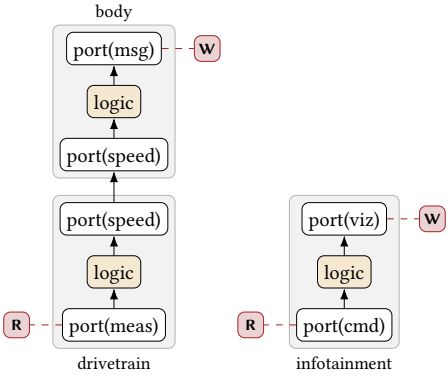


Figure 4.5: Visually enriched $G_\gamma[V_\alpha]$ for the car server example. Gray rectangles represent SWCs, while R/W pins highlight environment interactions.

This edge describes the fact that a failure of `proc(main.a53)` *will potentially lead to a failure of the entire underlying* `main` *platform.*

4.1.2 Illustrative examples of CF potential

In this section, selected portions of CF graphs will be discussed to convey a better intuition about the semantics of vertices and edges in G_γ.

4.1.2.1 Central car server with partitioned SWCs

We first discuss the partitioned version of the central car server without isolation measures, i.e., the scenario from Example 4.1. The corresponding context subgraph, $G_\gamma[V_\beta]$, is shown in Figure 4.6. Consider a failure of s_I, which is represented by `swc(infotainment)`. Since there is no logical isolation, the RTE hosting s_I remains unprotected from this failure. It must be assumed, for example, that SWC code accesses a memory region that the RTE has not allocated to the SWC. By doing so, the SWC can interfere with `rte(linux)` itself, which is captured by a directed edge from s_I to its RTE. CF potential of s_I does not end here, however. By interfering with the Linux system, s_I gains considerable control of the underlying CPU, which is visualized as `proc(main.a53)`. This on-chip component has full access over the entire address space of its MPSoC, i.e., `platform(main)`. Among other things, this access can result in erroneous writes to the entire DDR memory, i.e., `mem(main.ddr)`.

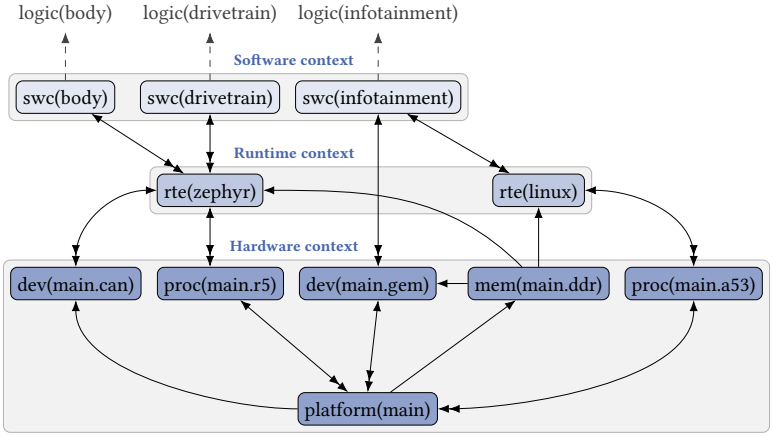


Figure 4.6: Visually enriched $G_\gamma[V_\beta]$ for the car server example. Connections to the application subgraph are reported using dashed arrows at the top edge; these connections are visual extensions and not part of the subgraph itself.

Since the Zephyr instance stores code and data in this memory module, the initial failure can propagate to `rte(zephyr)`. From there, it will potentially interfere with the implementation of the other two SWCs in the system, i.e., with `swc(body)` and `swc(drivetrain)`.

Context-to-application edges connect these two vertices to their respective logic block. In the case of `swc(body)`, for instance, there is an outgoing edge to `logic(body)`. As it was shown in the application subgraph, Figure 4.5, the actions taken by `logic(body)` influence `port(body.msg)`, which represents the safety-relevant body network.

4.1.2.2 APU configuration and process isolation

▹ Example 4.3: *Based on Example 4.1, consider the case that two isolation measures are specified: a request to generate APU configuration code for x_Z, and the declaration that r_Z is a runtime with process isolation. This affects $G_\gamma[V_\beta]$, which now has the structure visualized in Figure 4.7. Compared with the original context subgraph, five activity-based edges have been removed. The structure of the application subgraph remains unchanged; this means that context-to-application arrows at the upper edge of Figure 4.7 lead to the vertices in Figure 4.5, the application subgraph that is still valid.*

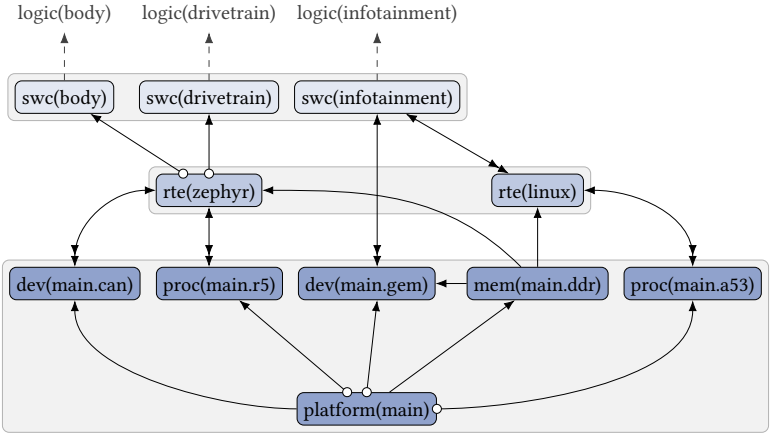


Figure 4.7: Visually enriched $G_\gamma[V_\beta]$ for the car server with two isolation measures. APU configuration code for `main` (x_Z) and process isolation for `zephyr` (r_Z) are reflected in the form of removed edges (each highlighted using a circle).

▹ Remark 4.3: *As documented in Figure 4.1b, edges removed due to an isolation measure are highlighted using a white circle (instead of the arrow tip).*

Considering the CF graph in Figure 4.7, we can now repeat the procedure from above and ask if a failure of the infotainment component will still propagate to the body network. This is not the case, since there is no longer a directed path from `swc(infotainment)` to `port(body.msg)`. From an implementation point of view, this improvement is due to the generated APU configuration code, which prevents `proc(main.a53)` from accessing the address space of `platform(main)` in an uncontrolled manner. Analogously, the process isolation of `rte(zephyr)` prevents `swc(body)` and `swc(drivetrain)` from interfering with the RTE itself.

4.1.2.3 Introduction of a seat adjustment feature

▹ Example 4.4: *Starting with the protected car server from Example 4.3, the system model is now refined to reflect Figure 4.8. This updated use case comprises a remote-controlled seat adjustment feature. To implement it, s_I is extended with an internal output $p_z^I \in P$, while s_B is extended with an internal input $p_z^B \in P$. When the vehicle is approached by a known driver, s_I receives a command to adjust the seat accordingly. Using its p_z^I output, it communicates the desired seat position*

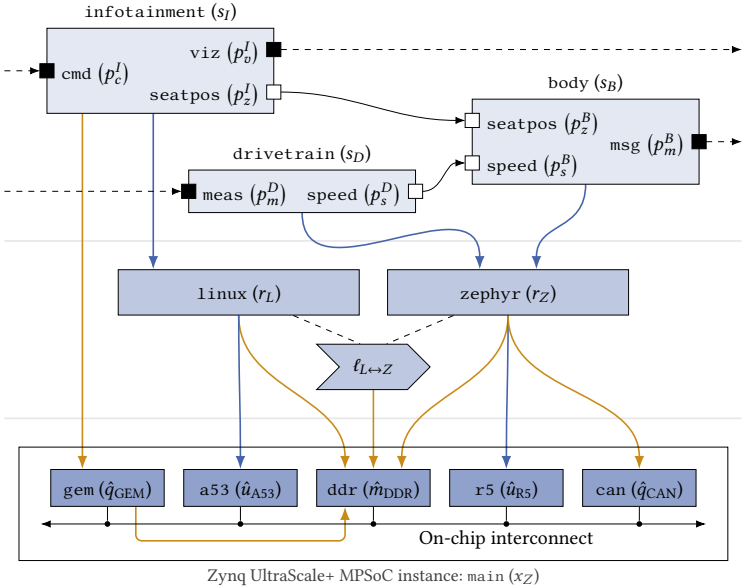


Figure 4.8: Central car server example with a remote-controlled seat adjustment feature. At the application level, there is now data flow from s_I to s_B.

to the p_z^B input of the body controller, i.e., there is now a channel $\langle p_z^I, p_z^B \rangle \in \phi_H$. Since this is an inter-RTE channel, it is now also necessary to introduce a local path between r_L and r_Z. As visualized in Figure 4.8, this local path ($\ell_{L\leftrightarrow Z} \in L$) is mapped to a DDR memory region.

The context subgraph for this scenario is visualized in Figure 4.9. Compared with the graph in Figure 4.7, it now comprises a `path(linux, zephyr)` *vertex representing the memory region that* `rte(zephyr)` *and* `rte(linux)` *use to communicate. Since the channel from* `port(infotainment.seatpos)` *to* `port(body.seatpos)` *is implemented via this channel, the path vertex has an outgoing edge to* `port(body.seatpos)` *in the application subgraph.*

With the introduction of the new seat adjustment feature, the application subgraph is affected as well. Under the assumption that no further isolation measures are specified, it is visualized in Figure 4.10.

The application subgraph in Figure 4.10 shows a key property of CF graphs: without further knowledge about the internals of SWC logic, it must be as-

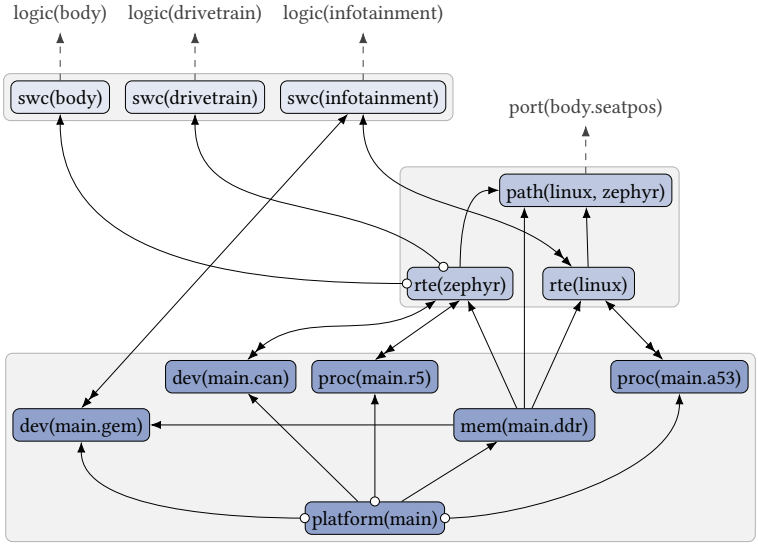


Figure 4.9: Visually enriched $G_Y[V_\beta]$ for the extended car server use case.

sumed that a failure of any SWC input causes all SWC outputs to fail. In the specific example, this means that failures of `port(infotainment.cmd)` can also propagate to `port(body.msg)`. Considering the possibly untrusted nature of infotainment commands and the safety-relevant impact of body network messages, this CF potential can develop into an issue during the safety assessment performed at a later time.

4.1.2.4 Application-level barriers

The following example assumes that the CF potential identified above is, in fact, a safety issue. Therefore, it introduces an application-level barrier.

▹ Example 4.5: *A safety assessment revealed that the remote-controlled seat adjustment can interfere with the driver's ability to operate the vehicle. Therefore, the feature must be disabled while the vehicle is in motion. The logic of s_B is extended to perform this check, and its nominal behavior is defined as follows:*

- *If* `speed` > 0*, written* `msg` *outputs are independent of* `seatpos` *inputs.*
- *Otherwise,* `msg` *outputs may trigger spurious seat adjustments.*

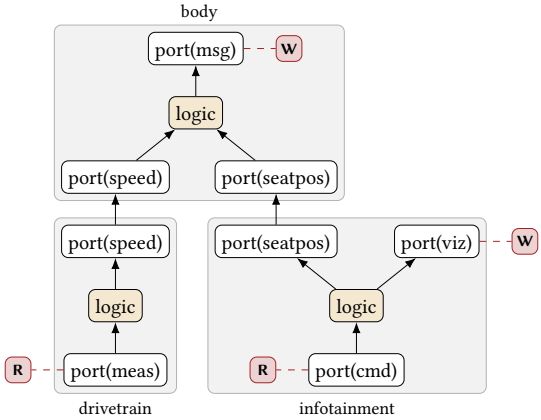


Figure 4.10: Visually enriched $G_\gamma[V_\alpha]$ for the extended car server use case.

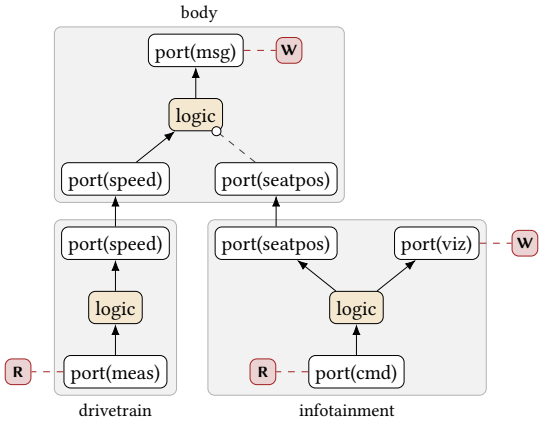


Figure 4.11: Visually enriched $G_\gamma[V_\alpha]$ for the extended car server use case with an application-level barrier between `port(body.seatpos)` and `logic(body)`.

Including the potential for spurious seat adjustments into the logic block specification allows future safety assessments to consider this aspect.

It also allows us to state that a failure of `port(body.seatpos)` *does no longer propagate to* `logic(body)`*: if* `speed` > 0*, erroneous* `seatpos` *inputs are ignored and will therefore not lead to a failure of* `logic(body)`*; if* `speed` $= 0$*,*

erroneous `seatpos` *inputs may trigger spurious seat adjustments, but these are covered in the specification of* `logic(body)` *and therefore not a failure.*

The updated application subgraph is shown in Figure 4.11. It captures the applied modifications as a removed incoming edge to `logic(body)`*.*

With the application-level barrier added to Figure 4.11, the CF potential from `port(infotainment.cmd)` to `port(body.msg)` is eliminated. This improvement comes at a cost, however: erroneous seat adjustments in *certain* situations are now a part of the body controller's specified behavior. In the safety assessment step, it will have to be shown that this circumstance does not violate a safety requirement (cf. Chapter 5).

4.2 Measures for logical isolation

Complementing the system model, a possibly empty set of isolation measures is the second input of the APU generation and CF determination step.

Isolation measures are means to limit logical interaction possibilities in the embedded software system. As described in Figure 3.3 on page 42, this work considers three types of isolation measures: APU configurations, process isolation, and application-level barriers. APU configuration is *requested* from the pattern, while the other two types are *communicated* to the pattern.

4.2.1 Isolation measure specification

Each isolation measure type is associated with one class of system elements:

- APU configurations are requested for execution platforms ($x \in X$).
- Process isolation is a binary attribute of deployed RTEs ($r \in R$).
- Application-level barriers are associated with logic blocks ($\lambda \in \Lambda^S$).

Recall from Section 4.1 that Λ^S is the set of all SWC logic blocks. It contains at least one logic block per SWC, but a more fine-grained specification of application behavior is possible through the *logic decomposition* concept.

Logic decomposition is an optional step and only relevant in combination with application-level barriers. If it is used, it becomes an implicit part of the isolation measures specified as part of the pattern input.

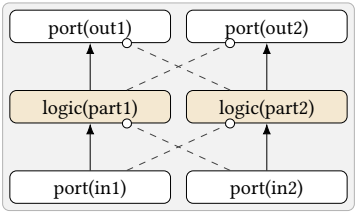


Figure 4.12: Application subgraph for a single SWC with two partial logic blocks, each associated with an input barrier (from Ω_I) and an output barrier (from Ω_O).

From the pattern user's perspective, logic decomposition is performed by populating the following tuple with values:

► **Definition 4.1:** Logic decomposition

A tuple $\langle \tilde{\Lambda}^S, \text{parent} \rangle$ is the *logic decomposition* of a given software architecture. It splits the logic of selected SWCs into different parts, each referred to as *partial logic block*, and it consists of two components:

1) $\tilde{\Lambda}^S \subseteq \Lambda^S$ is a finite set of partial logic blocks.
2) $\text{parent}\colon \tilde{\Lambda}^S \to S$ maps each partial logic block to its SWC.

▹ Example 4.6: *Logic decomposition makes it possible to model application-level behavior such as the one visualized in Figure 4.12. Here, the logic of a SWC was decomposed into two partial logic blocks,* `logic(part1)` *and* `logic(part2)`*. Each of them propagates failures from only one of the inputs and to only one of the outputs. Without logic decomposition, it would not be possible to model two separate CF potential paths within a single SWC.*

As described above, populating this tuple is optional. If a particular $s \in S$ is not associated with partial logic blocks (i.e., if $\nexists \lambda \in \tilde{\Lambda}^S : \text{parent}(\lambda) = s$), then a logic block $\lambda^s \in \Lambda^S$ is automatically introduced for s. This logic block is by definition not a partial one, i.e., $\lambda^s \notin \tilde{\Lambda}^S$ holds.

The auxiliary function 'partials$\colon S \to \tilde{\Lambda}^S$' allows us to refer to the partial logic blocks of the SWC passed as its argument:

$$\text{partials}(s) = \{\lambda \in \tilde{\Lambda}^S : \text{parent}(\lambda) = s\}.$$

The function 'logic: $S \to \Lambda^S$' returns all logic blocks of a SWC:

$$\text{logic}(s) = \begin{cases} \text{partials}(s), & |\text{partials}(s)| > 0, \\ \{\lambda^s\}, & \text{otherwise.} \end{cases}$$

Based on these functions, it is now possible to express Λ^S as

$$\Lambda^S = \bigcup_{s \in S} \text{logic}(s)$$

and define the isolation measure specification itself as follows:

► **Definition 4.2:** Isolation measure specification

The *isolation measure specification* is a tuple

$$\langle \Omega_X, \Omega_R, \Omega_I, \Omega_O \rangle$$

with the following components:

1) $\Omega_X \subseteq X$ is a set of APU configuration requests.
2) $\Omega_R \subseteq R$ is the set of RTEs implementing process isolation.
3) $\Omega_I \subseteq \{p \in P : \text{dir}(p) = \text{In}\} \times \Lambda^S$ is a set of application-level barriers between an input port and a logic block of the same SWC.
4) $\Omega_O \subseteq \Lambda^S \times \{p \in P : \text{dir}(p) = \text{Out}\}$ is a set of application-level barriers between a logic block an output port of the same SWC. For every output port, there must remain at least one logic block without an application-level barrier in between.

The pattern will generate APU configuration code for each $x \in \Omega_X$. The semantics of Ω_R, Ω_I, and Ω_O are described in Section 4.2.3. Roughly speaking, $r \in \Omega_R$ means that r protects itself from failures of its SWCs, elements of Ω_I suppress CFs from an input port to a logic block, and elements of Ω_O suppress CFs from a logic block to an output port.

▹ Example 4.5 (continued): *In the final version of the extended car server example from above, three isolation measures were applied: one requesting APU configuration code for* `platform(main)`*, one declaring process isolation for* `rte(zephyr)`*, and an input barrier between* `port(body.seatpos)` *and* `logic(body)`*. It was not necessary to decompose the logic of any SWC.*

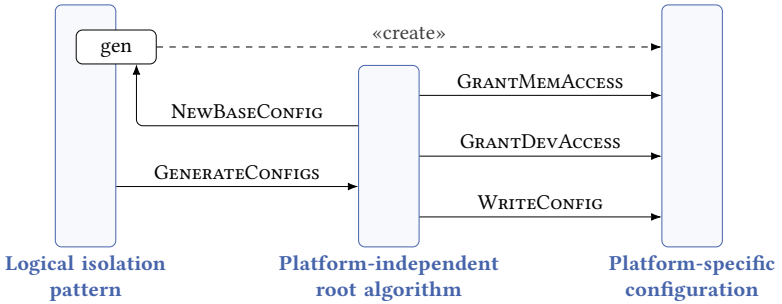


Figure 4.13: Interfaces between the pattern, the platform-independent root algorithm for logical isolation, and a platform-specific configuration.

Using the introduced modeling framework, the fact that no logic decomposition was applied is expressed by choosing $\tilde{\Lambda}^S = \varnothing$ *and* parent $= \varnothing$. *Setting*

$$\Omega_X = \{x_Z\}, \quad \Omega_R = \{r_Z\}, \quad \Omega_I = \left\{\langle p_z^B, \lambda^{S_B}\rangle\right\}, \quad \textit{and} \quad \Omega_O = \varnothing$$

formalizes the described isolation measure specification.

4.2.2 APU configuration for MPSoCs

To initiate an automatic APU configuration, the logical isolation pattern triggers a platform-independent root algorithm (cf. Figure 4.13).

This algorithm considers every APU configuration request ($x \in \Omega_X$) separately. For each such request, it consults the 'gen' collection of the pattern's platform library to obtain a mutable, platform-specific configuration object. To do so, it invokes the following function with $t = \text{type}(x)$:

$$\textsc{NewBaseConfig}(t \in T) : \textit{config}$$

The configuration object returned by this function (*config*) is expected to isolate all instantiated platform components from each other. However, accesses that involve internal on-chip components (i.e., components *not* exposed via the execution platform library) do not necessarily have to be prohibited. In fact, it is possible and often necessary to initialize the object with reasonable default permissions. Examples of such permissions include:

1) Accesses for internal power management functions.
2) External debug access to all on-chip components.

Among other things, they account for the fact that a bare minimum of transactions is generally required to keep a platform operational. The exact nature of these permissions is a property determined by the respective generator. At this stage, the configuration object is referred to as a *base* configuration. An example of a practical base configuration is given in Section 6.2.4.

The configuration object returned for each $x \in \Omega_X$ is evolved by repeatedly consulting the system model and granting required memory and peripheral device access permissions. This is achieved using two procedures of the mutable configuration object. The first procedure has the following interface:

$$\textsc{GrantMemAccess}(\mathit{config},\ \hat{c} \in (\hat{U} \cup \hat{Z}),\ \mu \in \mathbb{M})$$

Its first parameter (*config*) references the mutable configuration object to modify. The second parameter captures the instantiated platform component (from $\hat{U} \cup \hat{Z}$) whose access permissions shall be extended. The third and final parameter references the memory region (from $\mathbb{M}$) to grant read/write access to. Analogously, the second procedure has the following interface:

$$\textsc{GrantDevAccess}(\mathit{config},\ \hat{u} \in \hat{U},\ \hat{q} \in \hat{Q})$$

Here, the first parameter (*config*) is again a reference to the mutable configuration object. The two final parameters describe a processing unit (from $\hat{U}$) and the peripheral device (from $\hat{Q}$) that $\hat{u}$ shall obtain read/write access to.

To translate its state into APU configuration code, a platform-specific configuration object finally provides the following interface:

$$\textsc{WriteConfig}(\mathit{config})$$

The output artifacts generated by this procedure are platform-specific; they are entirely under the control of the platform-specific generator.

4.2.2.1 Platform-independent root algorithm

The platform-independent root algorithm is now described in more detail. As explained above, it handles all APU configuration requests:

Algorithm 4.1: Entry point into the APU configuration process

```
1 procedure GenerateConfigs()
2     // Process every APU configuration request:
3     for each x ∈ Ω_X do
4         GenerateConfig(x)
```

From line 4, it executes the following procedure for each $x \in \Omega_X$:

Algorithm 4.2: Logical isolation for an execution platform

```
1  procedure GenerateConfig(x ∈ X)
2      // Isolate instantiated platform components from each other:
3      config ← NewBaseConfig(type(x))
4      // Grant required access permissions:
5      for each ẑ ∈ Ẑ do
6          HandleMasterDevice(config, ẑ)
7      for each r ∈ R do
8          HandleRte(config, r)
9      for each ℓ ∈ L do
10         HandleLocalPath(config, ℓ)
11     for each s ∈ S do
12         HandleSwc(config, s)
13     // Generate and return APU configuration code:
14     WriteConfig(config)
```

▹ Remark 4.4: *In practice, a call to* NewBaseConfig *can fail with an error. This is the case if* 'gen' *does not contain a platform-specific generator for the respective execution platform type. For the sake of brevity, this alternative control flow path is not explicitly shown in the algorithm.*

In an incremental procedure, Algorithm 4.2 extends the mutable configuration object (*config*) to ensure that all access permissions required to implement individual parts of the system model are granted.

Starting at layer I of the system model, it first ensures that peripheral devices equipped with a master port ($\hat{z} \in \hat{Z}$) have full access to the memory regions allocated to them. This is achieved by the following procedure, which is called in line 6 of Algorithm 4.2:

Algorithm 4.3: Handling of allocations to master devices

```
1  procedure HandleMasterDevice(config, ẑ ∈ Ẑ)
2      // Handle memory region allocations:
3      for each μ ∈ malloc(ẑ) do
4          GrantMemAccess(config, ẑ, μ)
```

▹ Remark 4.5: *The fact that a specific peripheral device might not actually need full (i.e., read and write) memory region access is not captured by any pattern input. It can therefore not be considered by the APU configuration procedure.*

The procedure invoked in line 8 of Algorithm 4.2 deals with instantiated platform components allocated to an RTE:

Algorithm 4.4: Handling of allocations to an RTE

```
1  procedure HandleRte(config, r ∈ R)
2      // Handle memory region allocations:
3      for each μ ∈ malloc′(r) do
4          GrantMemAccess(config, proc(r), μ)
5      // Handle peripheral device allocations:
6      for each q̂ ∈ qalloc′(r) do
7          GrantDevAccess(config, proc(r), q̂)
8          if q̂ ∈ Ẑ then
9              // Handle indirect read/write permissions:
10             for each μ ∈ malloc(q̂) do
11                 GrantMemAccess(config, proc(r), μ)
12             for each μ ∈ malloc′(r) do
13                 GrantMemAccess(config, q̂, μ)
```

Note that for peripheral devices with a master port ($\hat{q} \in \hat{Z}$), the procedure also implements the two access permission rules from Section 3.2.3, i.e.:

- RTEs have access to memory owned by their peripheral devices.
- Peripheral devices have access to memory owned by their RTE.

▹ Remark 4.6: *At first glance, the second rule may seem unnecessarily lenient. However, it solves the practical challenge that OS kernels can allocate buffers to communicate with their peripheral devices anywhere in their memory. These locations may be dynamic or entirely unknown at design time.*

Afterward, line 10 of Algorithm 4.2 considers memory regions allocated to local paths. This is achieved by invoking the HandleLocalPath helper, which identifies the endpoints of a given local path and grants the underlying processing unit access to all owned memory regions:

Algorithm 4.5: Handling of memory regions for local paths

```
1  procedure HandleLocalPath(config, ℓ ∈ L)
2      // Grant memory region access to the processing unit of each endpoint:
3      for each r ∈ endpoints(ℓ) do
4          for each μ ∈ impl(ℓ) do
5              GrantMemAccess(config, proc(r), μ)
```

In line 12 of Algorithm 4.2, peripherals allocated to SWCs are translated to access permissions. This is achieved by the HandleSwc helper:

Algorithm 4.6: Handling of allocations to a SWC

```
1 procedure HandleSwc(config, s ∈ S)
2     // Handle peripheral device allocations:
3     for each q̂ ∈ qalloc''(s) do
4         GrantDevAccess(config, proc(rte(s)), q̂)
5         if q̂ ∈ Ẑ then
6             // Handle indirect read/write permissions:
7             for each μ ∈ malloc(q̂) do
8                 GrantMemAccess(config, proc(rte(s)), μ)
```

After all these steps, *config* is in its final state. To finalize the configuration process, line 14 of Algorithm 4.2 calls the WriteConfig interface. Doing so requests the platform-specific procedure from the 'gen' collection to generate executable APU configuration code from this state.

4.2.2.2 Code generator for the Zynq UltraScale+ MPSoC

To extend the 'gen' collection for a particular execution platform type, the following four functions/procedures need to be implemented for this type:

1) NewBaseConfig,
2) GrantMemAccess,
3) GrantDevAccess, and
4) WriteConfig.

Considering the Zynq UltraScale+ MPSoC as a representative example, this section describes one possible implementation of them. Note that it (also) serves as the theoretical foundation of Section 6.2.4, which covers the `zynqmp` generator developed as part of this thesis.

Zynq UltraScale+ MPSoCs are equipped with nine APUs: one XPPU and eight XMPUs. For clarity, XMPUs are suffixed with what they protect:

- XMPU/OCM: This unit monitors and controls accesses to the OCM, which is located in the LPD of the device.
- XMPU/DDR: Six of these units cover accesses to DDR memory, each responsible for accesses from a different inbound path.
- XMPU/FPD: This unit protects peripheral devices in the FPD of the platform, such as SLCRs or the GPU.

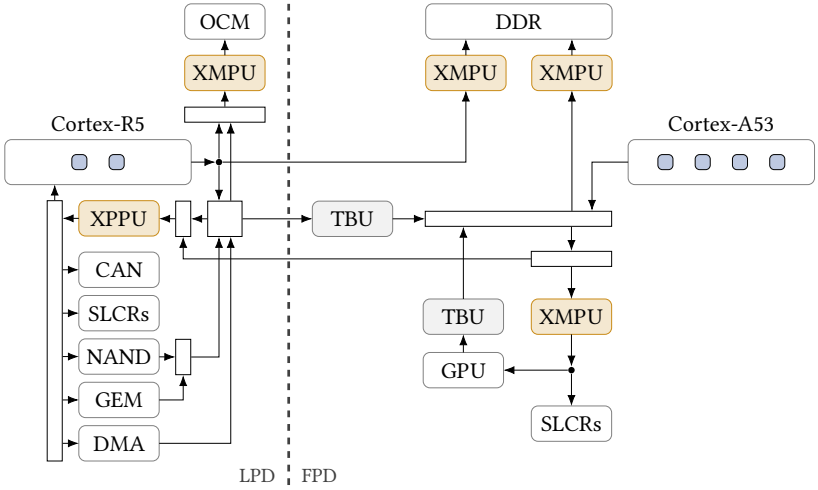


Figure 4.14: Selected components of the Zynq UltraScale+ MPSoC, including the XPPU and four of eight XMPUs. Schematic repeated from Figure 2.7 for convenience.

Figure 4.14 shows how the XPPU and four of the XMPUs are integrated into the platform itself. To restrict its complexity, the figure omits four of the XMPU/DDR instances. This is possible because, for the purposes of this work, all XMPU/DDR instances are treated equally.

From a configuration point of view, XPPUs and XMPUs constitute a sequence of memory-mapped registers.

XMPU configuration The address space of an XMPU contains global configuration and regional control registers. The regional control registers are organized into 16 blocks of 128 bit each. For the purposes of this work, each such block is referred to as a 'set' of regional control registers. The fields of a regional control register set are explained in Figure 4.15 and Figure 4.16, respectively. By populating any of the 16 sets, it is possible to specify precise access permissions of one or more masters, given that targeted slaves are actually protected by the respective XMPU. The entire XMPU can further be operated in a lenient or a strict mode; this mode determines how accesses *not* covered by a regional control register shall be handled. By prohibiting them, the XMPU operates in a whitelist fashion.

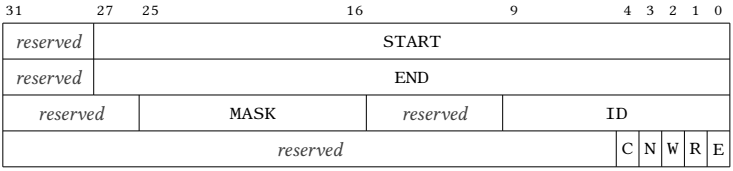


Figure 4.15: One of 16 regional control register sets in the single XMPU/FPD and all six XMPU/DDR instances. In addition to the slave address range (START – END) and master configuration fields (ID and MASK), it contains an enable (E), a read permission (R), a write permission (W) and two TrustZone (N/C) bits.

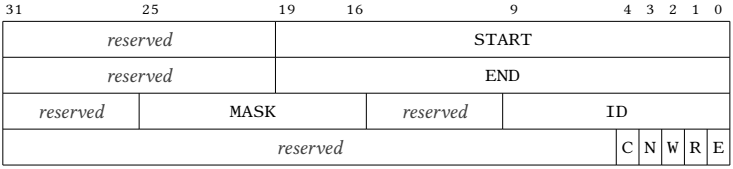


Figure 4.16: One of 16 regional control register sets in the XMPU/OCM instance. Except for a reduced START/END width, its structure is identical to that from Figure 4.15.

XPPU structure Although it pursues a similar objective, the XPPU has a different internal structure than the XMPU. The main difference is that the addresses to protect are predetermined and cannot be arbitrarily configured via START and END fields. With this, the XPPU is tailored to its main purpose: the protection of peripheral devices, whose address space must often be treated as an atomic block. Note, however, that some memory modules of the platform are protected by the XPPU rather than the XMPU. Most importantly, this applies to all TCM of the Cortex-R5.

XPPU configuration The configuration of an XPPU is determined by global control registers, 20 master profile registers, and 401 aperture configuration registers. The global control registers allow clients to enable and disable the XPPU protection. Figure 4.17 shows the fields of a master profile register. Each such

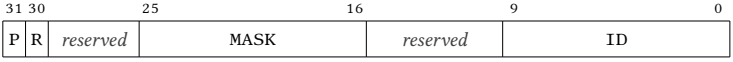


Figure 4.17: One of 20 master profile registers in the XPPU. The profile itself consists of two master configuration fields (ID and MASK) along with a read-only bit (R). The most significant bit (P) acts as an optional parity bit.

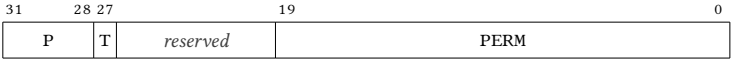


Figure 4.18: One of 401 aperture configuration registers in the XPPU. It specifies whether each of the 20 configured masters has access to the covered peripheral device (PERM). In addition, it comprises a TrustZone bit (T) and a parity field (P).

register can be used to represent one or more master components, and it encodes whether those master components are read-only masters. Based on this list, the 401 aperture configuration registers specify the access permissions of each master profile. Figure 4.18 shows the fields of one aperture configuration register; by setting its PERM field, clients specify which of the 20 master profiles shall have access to the respective slave.

Usage strategy To design a platform-specific generator, it is first necessary to decide how the available APUs shall be used. A possible usage strategy for the Zynq UltraScale+ MPSoC is as follows:

1) Use all APUs in a whitelist fashion, i.e., configure them to reject transactions by default. Therefore, initialize the PERM field of aperture configuration registers to zero and operate XMPUs in their strict mode.
2) Use each regional control register set (of an XMPU) and each master profile register (of the XPPU) to refer to no more than one bus master. Therefore, configure their MASK fields as narrowly as possible.
3) Grant only full (i.e., read and write) access permissions. Therefore, set both the R and the W bit when populating a set of regional control registers, and do *not* set the R bit of master profile registers.
4) Do not apply any TrustZone restrictions, i.e., keep generated APU configurations orthogonal to the TrustZone mechanism.
5) Generate exactly one XMPU/DDR configuration and mirror it to all six XMPU/DDR instances.

This strategy is the one that will be explained in the remainder of this section. It is also the strategy that was used in the reference implementation of the logical isolation pattern and is therefore covered further in Section 6.2.4.

Base configuration A possible approach to implement NewBaseConfig is to generate a data structure hosting four mutable configurations: one for the six XMPU/DDRs, and one dedicated configuration for each of the three other APUs. Then, suitable default permissions need to be applied, and the

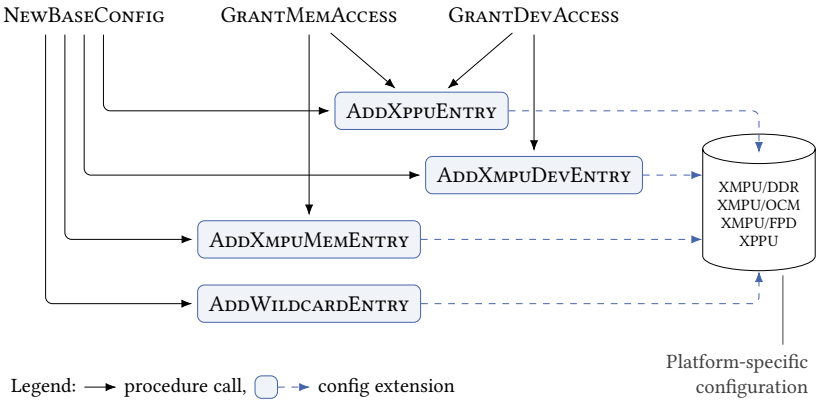


Figure 4.19: Permission granting procedures of a platform-specific generator for the Zynq UltraScale+ MPSoC. Solid arrows indicate how they are invoked.

data structure is returned as the mutable '*config*' object. This object can then be refined by the platform-independent root algorithm.

Permission granting procedures APU access permissions are granted by NewBaseConfig, GrantMemAccess, and GrantDevAccess. Each of them needs to be able to influence *all* configurations of the mutable configuration object.[1] Therefore, it is reasonable to introduce procedures to extend the mutable configuration object. These procedures (and how they are invoked) are visualized in Figure 4.19. All of them operate on symbolic names ($\mathbb{S}$) to identify a master component, a slave component, or both.

$$\text{AddXppuEntry}(\textit{config},\ \textit{master} \in \mathbb{S},\ \textit{aperture} \in \mathbb{S})$$

extends the current XPPU configuration with permissions that grant '*master*' full access to '*aperture*'. The procedure

$$\text{AddXmpuMemEntry}(\textit{config},\ \textit{target} \in \{\text{DDR}, \text{OCM}\},\ \textit{master} \in \mathbb{S},\ \mu \in \mathbb{M})$$

adds memory access permissions to either XMPU/DDR or XMPU/OCM. Which of these XMPUs to configure is determined by the '*target*' parameter. By

[1] For example, if the GrantMemAccess procedure is used to grant TCM access to a master, this needs to be reflected in the XPPU instead of an XMPU.

adding a suitable configuration entry, the procedure grants '*master*' full access to memory region μ. Similarly, the procedure

$$\textsc{AddXmpuDevEntry}(\mathit{config},\ \mathit{master} \in \mathbb{S},\ \mathit{slave} \in \mathbb{S})$$

extends the XMPU/FPD configuration to grant '*master*' full access to the '*slave*' module. Finally, the procedure

$$\textsc{AddWildcardEntry}(\mathit{config},\ \mathit{master} \in \mathbb{S})$$

is a helper that grants '*master*' full access to all possible slaves, including all memory regions. It extends the configuration of all APUs, and its usage is limited to the NewBaseConfig function.

Memory access handler Components that the system model treats as memory modules ($\hat{m} \in \hat{M}$) are protected by an XMPU/DDR, by the XMPU/OCM, or the XPPU. This is used in the following sample procedure:

Algorithm 4.7: Memory access handler for Zynq UltraScale+ MPSoCs

```
1  procedure GrantMemAccess(config, ĉ ∈ (Û ∪ Ẑ), μ ∈ 𝕄)
2      x ← pf(ĉ)
3      t ← type(x)
4      // Find the underlying master component:
5      if ĉ ∈ Û then
6          master ← (symbolic name of u ∈ U(t) with u^x = ĉ)
7      else if ĉ ∈ Ẑ then
8          master ← (symbolic name of z ∈ Z(t) with z^x = ĉ)
9      // Find the underlying memory module:
10     memory ← (m ∈ M(t) with μ ⊆ mmap(t)(m))
11     // Using the corresponding protection mechanism, add permissions:
12     if memory = m_DDR then
13         AddXmpuMemEntry(config, DDR, master, μ)
14     else if memory = m_OCM then
15         AddXmpuMemEntry(config, OCM, master, μ)
16     else if memory = m_TCM then
17         for each aperture ∈ (XPPU apertures covering μ) do
18             AddXppuEntry(config, master, aperture)
```

The procedure first derives the symbolic name that identifies the master to grant access to (line 6 or line 8), and it determines the generic memory module that the supplied memory region μ is part of (in line 10). Recall that

elements from $U(t)$, $Q(t)$, and $M(t)$ describe generic platform components; in contrast to elements from $\hat{U}$, $\hat{Q}$, and $\hat{M}$, they do not designate specific component instances. With this knowledge, Algorithm 4.7 calls the respective permission granting procedure(s). Note that each of these calls will fail if the remaining APU configuration resources are insufficient: AddXmpuMemEntry fails if the 16 regional control register sets of the relevant XMPU are already occupied; AddXppuEntry fails if all 20 master profile registers are occupied and it would be necessary to introduce an additional one.

Device access handler Zynq UltraScale+ MPSoC components that the system model treats as peripheral device ($\hat{q} \in \hat{Q}$) are either protected by the XPPU or the XMPU/FPD instance. A valid implementation of GrantDevAccess needs to take this detail into account, for example as follows:

Algorithm 4.8: Device access handler for Zynq UltraScale+ MPSoCs

```
1  procedure GrantDevAccess(config, û ∈ Û, q̂ ∈ Q̂)
2      x ← pf(û)
3      t ← type(x)
4      // Identify the underlying component:
5      master ← (symbolic name of u ∈ U(t) with u^x = û)
6      slave ← (symbolic name of q ∈ Q(t) with q^x = q̂)
7      // Using the corresponding protection mechanism, add permissions:
8      if slave is protected by XPPU then
9          AddXppuEntry(config, master, slave)
10     else if slave is protected by XMPU/FPD then
11         AddXmpuDevEntry(config, master, slave)
```

Like in the memory access case, this procedure first identifies the underlying platform components (in line 5 and line 6, respectively). Afterward, it uses the permission granting procedures from above to modify the mutable configuration object. Note that AddXmpuDevEntry reads the START and the END address to populate the regional control registers with from an internal lookup table. As before, this call may fail if the 16 regional control register sets of XMPU/FPD are already occupied.

▹ Remark 4.7: *With respect to the number of occupied configuration registers, the strategy described in this section is not necessarily optimal. In practice, optimizations can be applied to realize the same configuration with a reduced number of occupied XPPU or XMPU registers.*

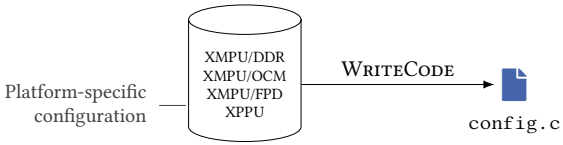


Figure 4.20: Symbolic representation of a WriteCode procedure that generates executable C code for the Zynq UltraScale+ MPSoC.

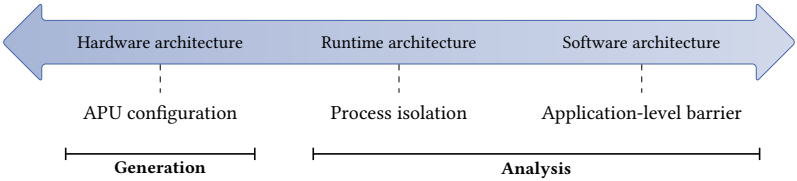


Figure 4.21: Isolation measures applicable to each system model layer and how they are considered by the pattern: either through generation or through analysis.

Code generation After all GrantMemAccess and GrantDevAccess calls have been executed, the mutable configuration object is in its final state. At this point, the platform-independent root algorithm calls WriteConfig to translate this state into executable APU configuration code. For the considered Zynq UltraScale+ MPSoC, this procedure can generate a lightweight C program performing writes to XPPU and XMPU registers (cf. Figure 4.20). This code may be self-contained or supported by a platform-specific configuration library. The reference implementation presented in Chapter 6 makes use of the latter option; it generates C code that depends on the library documented in Section A.2 of the appendix.

4.2.3 Semantics of barrier declarations

The procedure from Section 4.2.2 controls how on-chip interconnects of selected MPSoCs behave. As shown in Figure 4.21, the APU configuration code it returns operates at the hardware architecture level and is actively generated by the pattern. From a pattern user's point of view, the potential to misuse this isolation measure is limited: as long as generated APU configuration code is actually applied to the system under consideration, it is the pattern's responsibility to ensure that possible on-chip transactions are correctly represented in the CF graph. This is not true for the two other types of isolation measures,

i.e., process isolation and the application-level barrier. As it was explained in Section 3.1.1, these measures are only *declared* to and then *analyzed* by the pattern. Their use is completely optional. If they are used, however, it is the user's responsibility to ensure that conditions associated with each barrier declaration are actually met by a system.

4.2.3.1 Process isolation

In simplified terms, $r \in \Omega_R$ means that $r \in R$ constrains failures of its SWCs. More specifically, the meaning of process isolation is as follows:

$$r \in \Omega_R \Rightarrow \Big(\forall s \in \text{swcs}(r) : \text{a failure of } s \text{ cannot directly cause } r \text{ or } \text{proc}(r) \text{ to fail}\Big),$$

where *directly* refers to the direct interaction between a SWC and its RTE or, alternatively, between a SWC and its processing unit. Indirect effects (due to the failure of other system elements) do not have to be considered to decide whether $r \in \Omega_R$ holds for a particular $r \in R$.

Recall from Section 3.3.1 that in CF graphs, $r \in R$ represents not only the RTE itself. It also represents the memory region(s) allocated to this RTE. Selected portions of this memory region will typically be used to store kernel code, store kernel data, or they will be delegated to a particular SWC.

RTEs that execute SWCs without limiting their access to such memory regions are highly susceptible to CFs: failed SWCs might be able to override memory used by the kernel, for instance. In this case, it would *not* be justified to add this RTE to Ω_R. Limiting the memory access permissions of every SWC to the regions that are delegated to this SWC is a necessary condition that an $r \in \Omega_R$ has to meet. A similar statement can be made about peripheral devices that the RTE has access to: ensuring that SWCs are only able to access peripheral devices delegated to them is another necessary condition for $r \in \Omega_R$. In practice, these requirements will often be enforced through local MMU or MPU configurations managed by r. Examples of operating systems that are able to apply such configurations are Linux and Zephyr.

MMU/MPU protection is not sufficient to declare process isolation for a particular RTE, however. Even with this protection in place, a SWC might be able to overwrite an important processor control register. Other possible interference sources are means for inter-process communication; the following snippet shows how such mechanisms may cause an RTE to fail.

▹ Example 4.7: *Consider an RTE based on Linux, where every SWC is spawned as a user-space process. Although these processes will not be able to access each other's memory spaces, POSIX signals allow them to terminate another. This mechanism is triggered by the following C code, for instance:*

Listing 4.1: SWC interference via POSIX signals

```c
#include <signal.h>

int main(void) {
  pid_t target_pid = /* PID of another SWC binary */;
  return kill(target_pid, SIGKILL);
}
```

From the perspective of the logical isolation pattern, this would primarily be a failure of the RTE, because this RTE fails to provide the expected execution environment to the terminated SWC. The termination of the SWC is a second failure that is an immediate consequence of the first one.

State-of-the-art software for OS-level virtualization on Linux, such as Docker or LXC, is a possible way to solve this particular issue.

4.2.3.2 Application-level barriers

Application-level barriers capture external knowledge about CF potential between SWC input ports, SWC logic blocks, and SWC output ports. They allow the user to either ignore the impact of an input port on a logic block (Ω_I) or the impact of a logic block on an output port (Ω_O). Application-level barriers may be declared only after careful evaluation of a SWC.

Formally, elements of Ω_I have the following meaning:

$$\langle p, \lambda \rangle \in \Omega_I \Rightarrow \Big(\text{a failure of SWC input } p \text{ cannot directly cause logic block } \lambda \text{ to fail}\Big),$$

where *directly* means through direct interactions within the SWC. Indirect effects (via a cyclic channel structure in the software architecture) do not have to be considered to decide whether such a barrier is justified. Another way to express this necessary condition is to say that the correctness of p does not have a direct impact on the correctness of λ. Elements of Ω_I will also be referred to as *input barriers* in the following.

Analogously, elements of Ω_O have the following meaning:

$$\langle \lambda,\, p \rangle \in \Omega_O \Rightarrow \Big(\text{a failure of logic block } \lambda \text{ cannot directly cause SWC output } p \text{ to fail}\Big),$$

where *directly* has the same meaning as above. In other words: the correctness of λ does not have a direct impact on the correctness of p. Such barriers will also be referred to as *output barriers* in the following.

▹ Remark 4.8: *When tuples are added to Ω_O, it is essential to ensure that every SWC output port remains associated with at least one SWC logic block. This invariant is stipulated by Definition 4.2 and will be leveraged in Section 4.3.2.*

As it was illustrated in Example 4.5, application-level barriers can be used to reflect SWC logic that performs a safety check before processing data from a particular input. The feature can also be used to model that a SWC logic block does not read from or write to a particular SWC port, i.e., that there is no nominal information flow from or to a SWC port.

4.3 CF determination procedure

This step is concerned with the creation of a CF graph. Complementing generated APU configurations, this graph is the second and final result of the first safety pattern step. According to Definition 3.5, it is an exact representation of the CF potential in a considered system.

4.3.1 Formal foundation

The algorithm to create a CF graph is based on one global premise and three theorems. The global premise is the following:

► **Premise 4.1:** Absence of hidden failure propagation

CFs propagate only from system element to system element.

In other words, the pattern assumes that other than system elements, no physical or logical system parts are relevant to reason about CFs. When the set of system elements was formalized in Section 4.1.1, the definition of when an

instantiated platform component is 'in use' was postponed. Since it is relevant to ensure that Premise 4.1 is met, it is given here:

► **Definition 4.3:** Use of instantiated platform components

An instantiated platform component $\hat{c} \in \hat{C}$ is *in use* if and only if the following conditions are met:

$$\hat{c} \in \hat{U} \Rightarrow \left(\exists r \in R : \text{proc}(r) = \hat{c}\right),$$
$$\hat{c} \in \hat{M} \Rightarrow \left(\exists v \in (\hat{Z} \cup R \cup L) : \hat{c} \in \text{mems}'(v)\right), \text{ and}$$
$$\hat{c} \in \hat{Q} \Rightarrow \left(\hat{c} \in \hat{Z} \wedge \text{malloc}(\hat{c}) \neq \varnothing\right) \vee \left(\exists v \in (R \cup S) : \hat{c} \in \text{devs}'(v)\right) \vee \left(\exists b \in B : \hat{c} \in \text{controllers}(b)\right).$$

With this, the set of system elements V is now unambiguously defined. To decide whether a failure propagates from one system element to another, the procedure makes use of the following definition:

► **Definition 4.4:** CF potential transfer

The relation $\rightsquigarrow \subseteq V \times V$ describes *CF potential transfers* from one to another system element. $v_1 \rightsquigarrow v_2$ is true if and only if a failure of v_1 has the potential to *directly* cause a failure of v_2.

Here, the term *directly* means that the CF is due to the direct interaction between the two involved system elements. Indirect effects due to a failure of one or more other, intermediate system elements are deliberately *not* captured by the definition of CF potential transfer. Intuitively speaking, values of '$\rightsquigarrow$' capture how CF potential 'transfers' between system elements. They can be used to determine the complete CF potential:

► **Theorem 4.1:** Derivation of CF potential

CF potential from $v \in V$ leads to $v' \in V$ if and only if there are distinct vertices $w_1, \ldots, w_k \in V$ such that $w_1 = v$, $w_k = v'$, and $w_i \rightsquigarrow w_{i+1}$ is true for all $i = 1, \ldots, k-1$.

As shown in the following proof, this theorem holds if Premise 4.1, i.e., the absence of hidden failure propagation, is fulfilled.

▹ Proof: We first rewrite the left-hand side of the theorem:

$$\begin{aligned} & \text{There is CF potential from } v \in V \text{ to } v' \in V. \\ \overset{\text{(Def. 3.1)}}{\Longleftrightarrow} \; & \text{There is the possibility of a CF that is caused} \\ & \text{by } v \text{ and leads to a failure of } v'. \end{aligned}$$

Due to Premise 4.1, this is equivalent to the following statement: There is an ordered sequence of distinct system elements, where v is the first, v' is the last, and a failure of each system element has the potential to cause a direct failure of the next system element. Through the application of Definition 4.4, this can be rewritten as the right-hand side of Theorem 4.1 above. □

For the creation of CF graphs, Theorem 4.1 is an important intermediate result. It reduces the problem of finding the transitive CF potential to finding only CF potential transfers, which is a non-transitive relation. With knowledge about (1) the feasibility of interactions in an embedded system and (2) applied isolation measures, finding this non-transitive relation is significantly easier than arguing about the complete CF potential.

Based on this observation, a possible strategy to construct a CF graph is to introduce a directed edge $\langle v_1, v_2 \rangle \in E$ if and only if there is CF potential transfer from system element v_1 to system element v_2. This strategy leads to a CF graph in the sense of Definition 3.5:

► Theorem 4.2: Creation of CF graphs

If the set $E \subseteq V \times V$ fulfills the condition

$$\langle v_1, v_2 \rangle \in E \Leftrightarrow (v_1 \rightsquigarrow v_2),$$

then $G_\gamma = \langle V, E \rangle$ is a CF graph.

▹ Proof: By definition, G_γ must contain a directed path from $v \in V$ to $v' \in V$ if and only if there is CF potential from v to v'.

According to Section 2.2.2, a directed graph $\langle V, E \rangle$ contains a directed path from $v \in V$ to $v' \in V$ if and only if there are distinct vertices $w_1, \ldots, w_k \in V$ such that $w_1 = v$, $w_k = v'$, and $\langle w_i, w_{i+1} \rangle \in E$ for all $i = 1, \ldots, k-1$. Using the condition from Theorem 4.2 itself, the requirement that $\langle w_i, w_{i+1} \rangle \in E$ can be replaced by $w_i \rightsquigarrow w_{i+1}$ for all $i = 1, \ldots, k-1$. According to Theorem 4.1, this is exactly the definition of CF potential from v to v'. □

This is not the only approach for CF graph construction, however. As it was already mentioned in Section 3.1.1.2, it is possible to drop edges from a CF graph and obtain a CF graph semantically equivalent to the initial one. The following edge creation strategy makes use of this observation:

► **Theorem 4.3:** Creation of reduced CF graphs

If the set $E \subseteq V \times V$ fulfills the conditions

1) $\langle v_1, v_2 \rangle \in E \Rightarrow (v_1 \rightsquigarrow v_2)$ and
2) $(v_1 \rightsquigarrow v_2) \Rightarrow \langle V, E \rangle$ contains a directed path from v_1 to v_2,

then $G_\gamma = \langle V, E \rangle$ is a CF graph.

▹ Proof: To show that $\langle V, E \rangle$ is a CF graph, we decompose the biconditional from Definition 3.5 into a necessary and a sufficient condition.

First, the following statement must be proven: ($\langle V, E \rangle$ contains a directed path from $v \in V$ to $v' \in V$) $\Rightarrow$ (CF potential leads from v to v'). Using item 1 from the theorem itself, this can be achieved by repeating the proof of Theorem 4.2 and replacing the final equivalence with (only) an implication.

Into the opposite direction, it must be shown that the following statement is true: (CF potential leads from $v \in V$ to $v' \in V$) $\Rightarrow$ ($\langle V, E \rangle$ contains a directed path from v to v'). According to Theorem 4.1, the left-hand side is equivalent to the statement that there are distinct vertices $w_1, \ldots, w_k \in V$ such that $w_1 = v$, $w_k = v'$, and $w_i \rightsquigarrow w_{i+1}$ holds for all $i = 1, \ldots, k-1$. Using item 2 from the theorem, this implies that there are distinct vertices $w'_1, \ldots, w'_k \in V$ such that $w'_1 = v$, $w'_k = v'$, and $\langle V, E \rangle$ contains a directed path from w'_i to w'_{i+1} for all $i = 1, \ldots, k-1$. This, in turn, implies that, vertex v' is reachable from v and, consequently, that there is a directed path from v to v'. □

4.3.2 CF potential transfers

This section describes an approach to derive CF potential transfers from (1) a system model and (2) a specification of applied isolation measures. As part of the description, it will be argued why it is reasonable to assume that there *is* or, alternatively, that there is *not* a CF potential transfer between a pair of system elements. In the latter case, the decision is occasionally based on additional premises whose fulfillment has to be ensured by the user of the safety pattern. Such premises will be explicitly highlighted.

		CF potential transfer	
	Scope	Predicate	Impact
D01)	$\forall \hat{c}$	$\hat{c} \in \hat{C}_+$	$\mathrm{pf}(\hat{c}) \rightsquigarrow \hat{c}$
D02)	$\forall r$	$r \in R$	$\mathrm{proc}(r) \rightsquigarrow r$
D03)	$\forall s$	$s \in S$	$\mathrm{proc}(\mathrm{rte}(s)) \rightsquigarrow s$
D04)	$\forall v, \hat{m}$	$v \in (\hat{Z} \cup L \cup R), \hat{m} \in \mathrm{mems}'(v)$	$\hat{m} \rightsquigarrow v$
D05)	$\forall v, \hat{q}$	$v \in (S \cup R), \hat{q} \in \mathrm{devs}'(v)$	$\hat{q} \rightsquigarrow v$
D06)	$\forall g, b$	$g \in G, b \in \mathrm{bus}(g)$	$b \rightsquigarrow g$
D07)	$\forall s$	$s \in S$	$\mathrm{rte}(s) \rightsquigarrow s$
D08)	$\forall y, r$	$y \in (L \cup G), r \in \mathrm{endpoints}(y)$	$r \rightsquigarrow y$
D09)	$\forall y, p$	$y \in (L \cup G), p \in \mathrm{sinks}(y)$	$y \rightsquigarrow p$
D10)	$\forall s, \lambda$	$s \in S, \lambda \in \mathrm{logic}(s)$	$s \rightsquigarrow \lambda$
D11)	$\forall s, \lambda, p$	$s \in S, \lambda \in \mathrm{logic}(s), p \in \mathrm{inputs}(s), \langle p, \lambda \rangle \notin \Omega_I$	$p \rightsquigarrow \lambda$
D12)	$\forall s, \lambda, p$	$s \in S, \lambda \in \mathrm{logic}(s), p \in \mathrm{outputs}(s), \langle \lambda, p \rangle \notin \Omega_O$	$\lambda \rightsquigarrow p$
D13)	$\forall p, p'$	$\langle p, p' \rangle \in \phi_H$	$p \rightsquigarrow p'$

Table 4.1: CF potential transfers due to the dependency on system elements. From top to bottom, the four groups cluster CF potential transfers originating from the hardware context, runtime context, software context, and application vertices, respectively. The consideration of isolation measures is highlighted with blue background.

▹ Remark 4.9: *Based on other premises, it is possible to argue different strategies for the determination of '$\rightsquigarrow$' values. The strategy presented below is the one that the author considers most practical for the given context.*

To structure the description of the approach, we split CF potential transfers into two classes: dependency-based and activity-based ones. They are analogous to dependency-based and activity-based CF graph edges, respectively. Each type is now described in a dedicated subsection. Together, both subsections describe the '$\rightsquigarrow$' relation fully (closed-world assumption).

4.3.2.1 Dependency-based transfers

Dependency-based transfers capture cases in which a failure of software element v_1 might directly cause a failure of software element v_2, because v_2 is implemented by or implemented using v_1. Table 4.1 presents thirteen rules to derive them. Each rule is labeled with a unique identifier, e.g., rule D01. In each identifier, the 'D' emphasizes that this rule describes dependency-based trans-

fers. The table has a bottom-up nature in the sense that it is first concerned with the hardware context, then with the runtime context, and so on.

Every rule in Table 4.1 encodes an expression in first-order logic, where the 'Scope' specifies one or more bound variables. For all possible combinations ($\forall$) of these variables, if the associated 'Predicate' holds true, then the CF potential transfer given as 'Impact' is implied. For example, rule D13 is equivalent to the following expression:

$$\underbrace{\forall p \, \forall p'}_{\text{Scope}} \big(\underbrace{(\langle p, p' \rangle \in \phi_H)}_{\text{Predicate}} \Rightarrow \underbrace{(p \rightsquigarrow p')}_{\text{Impact}} \big).$$

In the following, it is reasoned why the table contains exactly the thirteen presented rules. The fundamental question that repeatedly has to be answered is the following: For a given system element v, which other system elements directly depend on v and, therefore, can fail if v fails?

Execution platforms (D01) An execution platform hosts and interconnects instantiated platform components. If it fails, it might no longer be able to provide these components with the environment they require. For example, if the platform is no longer able to power a memory module, this memory module ceases to deliver its intended functionality. If its on-chip interconnect stalls, does not achieve the required throughput, forwards transactions between components that should not communicate, or falsifies transactions, then all instantiated platform components might be affected. Since unused components are no system elements, they are not relevant for the purposes of CF determination. For each used component ($\hat{c} \in \hat{C}_+$), however, there is CF potential transfer from $\mathrm{pf}(\hat{c})$ to $\hat{c}$. This is captured by rule D01. No other system elements depend directly on an execution platform.

Processing units (D02, D03) Processing units are responsible for the execution of instructions, primarily those issued by an RTE. Temporarily, an RTE might delegate the capabilities of its processing unit to a SWC. If the processing unit is affected by a soft error, for instance, an instruction from kernel code might be falsified during its execution. More generally, an RTE and all of its SWCs depend on the functionality of their underlying processing unit. Formally, this is reflected by rule D02 and rule D03. Apart from those two, no other system elements depend directly on a processing unit.

Memory modules and peripheral devices (D04, D05) As it was described in Section 3.3.1, there are three system elements that also represent the memory region(s) allocated to them: peripheral devices with a master port ($\hat{z} \in \hat{Z}$), RTEs ($r \in R$), and local paths ($\ell \in L$). Each such memory region must be realized by a specific memory module. If this memory module fails, for example due to a radiation-induced soft error, a failure of any system element that stores data in the memory module is possible. This is captured by rule D04. A similar argument can be made about peripheral devices allocated to an RTE ($r \in R$) or a SWC ($s \in S$). For example, a peripheral device might be an on-chip timer that an RTE uses to schedule tasks. Without further knowledge, it must be assumed that the owner of a peripheral device depends on the correct functionality of this device. Formally, rule D05 describes this assumption. No other system elements depend directly on the correct functionality of a memory module or a peripheral device.

Off-chip interconnects (D06) Off-chip interconnects are physical media used to implement global paths ($g \in G$). If an off-chip interconnect stalls, does not achieve the required throughput, forwards transactions between I/O controllers that should not communicate, or falsifies transactions, then the global path lacks a reliable foundation. Therefore, as described by rule D06, a global path depends on the correct operation of its off-chip interconnect. The only other system element type that can be in direct contact with an off-chip interconnect is the peripheral device. If a peripheral device is used as an I/O controller, the functionality it provides—in the given context—is implemented using the attached off-chip interconnect. Regarding the potential of CFs, however, it is often reasonable to assume that a failure of the off-chip interconnect does not lead to a misconfiguration of the I/O controller itself:

► **Premise 4.2:** Self-protection of I/O controllers

Peripheral devices used as I/O controllers do not fail due to a malfunction of the off-chip interconnect attached to them.

If this invariant is enforced by the pattern user, it is not necessary to add a dependency-based CF potential transfer from an off-chip interconnect to I/O controllers attached to it. Other direct dependencies on the correct operation of an off-chip interconnect do not exist.

RTEs (D07, D08) RTEs form the foundation of the SWCs they execute, as well as the communication paths for which they serve as an endpoint. If the RTE fails to schedule its processes, a SWC might no longer be provided with its expected execution environment. If the kernel code that manages inter-RTE communication is erroneous, associated communication paths can fail. These aspects are reflected in rule D07 and rule D08, respectively. No other system elements depend directly on the behavior of RTEs.

Local and global paths (D09) A local path represents memory regions that two local RTEs use to communicate. Analogously, a global path is a virtual channel that two RTEs from *different* execution platforms use to communicate. In both cases, this communication mechanism is primarily used to transmit data from SWC outputs generated on one RTE to SWC inputs on another RTE. In both cases, failures of a path can have an effect on all the channels ($\langle p,\ p' \rangle \in \phi_H$) that are implemented using this system element. By definition (cf. Section 3.2.4), the failure cannot affect output ports writing to a channel. However, it *can* affect the values that are received by input ports reading data from corresponding channels. Formally, this relationship is captured in rule D09. Furthermore, it must generally be assumed that the failure of a path affects the connected RTEs by themselves. Since these RTEs have full control over how they use the local or global path that connects them, however, it is possible to add the following premise:

► **Premise 4.3:** Absence of path-induced RTE failures

The correct operation of an RTE does not depend on the correct operation of local or global paths associated with this RTE.

With this premise in place, no system element other than the SWC input port depends on the correct operation of a path.

SWC implementations (D10) These system elements represent the implementation of the logic to be performed by every SWC. If this implementation is erroneous, e.g., because an instruction in the 'text' section of the SWC's memory region became corrupted, it must be assumed that the SWC logic deviates from its intended behavior. This dependency is described by rule D10. No other system element directly depends on the SWC implementation.

SWC input ports (D11) Data received via a SWC input port is provided to SWC logic blocks. It might be erroneous, e.g., due to rule D09. If this is the case, SWC logic blocks operating on this data might fail to deliver their intended behavior. After examination of a SWC, it might be possible to argue that such a dependency does not exist in certain cases. Given the following premise, this can be represented as an input barrier ($\langle p, \lambda \rangle \in \Omega_I$):

► **Premise 4.4:** Correctness of the input barrier set

Every element of Ω_I fulfills the specification from Section 4.2.3.2.

Without an input barrier, the CF potential transfer shown in rule D11 must be assumed. Other direct dependencies on SWC input ports do not exist.

SWC logic blocks (D12) Without further knowledge about the behavior of a logic block, it must be assumed that its failure propagates to all output ports of the respective SWC. As in the previous case, an application-level barrier can be specified to suppress this assumption ($\langle \lambda, p \rangle \in \Omega_O$). As before, such barriers are associated with an invariant that must be enforced:

► **Premise 4.5:** Correctness of the output barrier set

Every element of Ω_O fulfills the specification from Section 4.2.3.2.

Without an output barrier, the CF potential transfer shown in rule D12 must be assumed. Other dependencies on a logic block do not exist.

SWC output ports (D13) Outputs with environment scope affect only the environment and, therefore, cannot give rise to dependency-based transfers. Outputs with internal scope, however, can be connected to other ports via a channel ($\langle p, p' \rangle \in \phi_H$). If incorrect values are written to an output, ports at the receiving end of a channel connected to this output read incorrect values, i.e., they fail. This relationship is captured by rule D13. Other dependencies on SWC output ports do not exist.

		CF potential transfer	
	Scope	Predicate	Impact
A01)	$\forall s$	$s \in S$, $\text{rte}(s) \notin \Omega_R$	$s \rightsquigarrow \text{rte}(s)$
A02)	$\forall s$	$s \in S$, $\text{rte}(s) \notin \Omega_R$	$s \rightsquigarrow \text{proc}(\text{rte}(s))$
A03)	$\forall s, \hat{q}$	$s \in S$, $\hat{q} \in \text{qalloc}''(s)$	$s \rightsquigarrow \hat{q}$
A04)	$\forall r$	$r \in R$	$r \rightsquigarrow \text{proc}(r)$
A05)	$\forall s, \hat{q}$	$s \in S$, $\hat{q} \in \text{qalloc}''(s)$	$\text{rte}(s) \rightsquigarrow \hat{q}$
A06)	$\forall r, \hat{q}$	$r \in R$, $\hat{q} \in \text{qalloc}'(r)$	$r \rightsquigarrow \hat{q}$
A07)	$\forall v$	$v \in (\hat{U} \cup \hat{Z})$, v in use, $\text{pf}(v) \notin \Omega_X$	$v \rightsquigarrow \text{pf}(v)$
A08)	$\forall s, \hat{q}$	$s \in S$, $\hat{q} \in \text{qalloc}''(s)$	$\text{proc}(\text{rte}(s)) \rightsquigarrow \hat{q}$
A09)	$\forall r, \hat{q}$	$r \in R$, $\hat{q} \in \text{qalloc}'(r)$	$\text{proc}(r) \rightsquigarrow \hat{q}$
A10)	$\forall x, u, m$	$x \in X$, $\langle u, m \rangle \in \Gamma_1(\text{type}(x))$, u^x and m^x in use	$u^x \rightsquigarrow m^x$
A11)	$\forall x, q, c$	$x \in X$, $\langle q, c \rangle \in \Gamma_2(\text{type}(x))$, q^x and c^x in use	$q^x \rightsquigarrow c^x$
A12)	$\forall x, q$	$x \in X$, $q \in \Gamma_3(\text{type}(x))$, q^x in use	$q^x \rightsquigarrow x$
A13)	$\forall b, \hat{q}$	$b \in B$, $\hat{q} \in \text{controllers}(b)$	$\hat{q} \rightsquigarrow b$

Table 4.2: CF potential transfers due to activities performed by system elements. From top to bottom, the three groups cluster CF potential transfers originating from the software context, runtime context, and hardware context, respectively. The consideration of isolation measures is highlighted with blue background.

4.3.2.2 Activity-based transfers

Activity-based transfers capture cases in which a failure of system element v_1 might directly cause a failure of system element v_2, because v_1 is able to perform activities affecting v_2. Like in the previous section, Table 4.2 presents the rules to derive them. This time, rule identifiers carry an 'A' prefix to emphasize the activity-based focus, and the table has a top-down nature: it is first concerned with the transfers originating from the software context, then from the runtime context, and finally from the hardware context.

With respect to temporal isolation, activities triggered by failed system elements are a difficult topic. For example, a failed SWC might access a peripheral device allocated to this SWC so often that the on-chip interconnect is overloaded, leading to increased memory access latencies of an unrelated RTE. A similar argument can be made about logically valid accesses to a memory region that cause unexpected contention at a memory module. As outlined in Chapter 1, APUs are means for spatial isolation and have only limited control over timing aspects. Depending on the timing requirements of applications, it

might be necessary to address timing interferences in addition to performing spatial isolation. If this is the case, the logical isolation pattern regards this as the user's responsibility. The argumentation in the remainder of this section is based on and only valid if the following premise is fulfilled:

► **Premise 4.6:** Sufficient temporal isolation

If a system runs applications with timing requirements, anomalous activities directed toward (1) an execution platform or (2) an instantiated platform component will not cause this system element to deliver a temporal performance insufficient for applications that depend on it.

▹ Remark 4.10: *Dependency-based transfers of CF potential, which are described in Table 4.1, can support the user in checking this premise. If there is no chain of dependency-based CF potential transfers from a hardware vertex to an application vertex, timing issues of the former cannot cause a failure of the latter.*

SWC implementations (A01, A02, A03) If a SWC implementation fails, the activities it is able to perform depend heavily on whether its RTE attempts to contain the failure. Executed on an RTE without process isolation ($r \notin \Omega_R$), the SWC might write to the memory region that its RTE uses to store kernel code. This must be classified as a failure of the RTE. Without process isolation, the SWC might also be able to cause the underlying processor to fail, for example by writing to a CPU-specific configuration register. Both of these activities are reflected in rule A01 and rule A02, respectively. If a SWC is the owner of a peripheral device, it is responsible for the configuration of the peripheral device. Naturally, its failure might lead to an erroneous configuration, which must be regarded as failure of the peripheral device. This relationship is represented by rule A03. Due to Premise 4.6, activities triggered by the failure of a SWC implementation cannot cause a failure of any other system element. Furthermore, if process isolation is used, the validity of the above argumentation depends on the correctness of Ω_R:

► **Premise 4.7:** Correctness of the process isolation set

Every element of Ω_R fulfills the specification from Section 4.2.3.1.

RTEs (A04, A05, A06) Executed on a processing unit, each RTE is able to trigger activities that compromise this processing unit. For example, its failure can lead to the corruption of important processor registers or initiate an on-chip transaction that the processing unit should not execute. This relationship is captured by rule A04. Memory regions allocated to the RTE are represented by the RTE vertex itself (cf. Section 3.3.1) and therefore do not warrant outgoing CF potential transfers. With respect to peripheral devices, an RTE has access to its own and the peripheral devices of its SWCs. Its failure may cause a misconfiguration of these peripheral devices, as reflected by rule A05 and rule A06. RTEs are unable to perform other activities that result in the direct failure of another system element.

Processing units and peripheral devices (A07 – A13) Instantiated processing units ($\hat{u} \in \hat{U}$) and peripheral devices with a master port ($\hat{z} \in \hat{Z}$) have access to the on-chip interconnect of their execution platform. Their failure might therefore cause an on-chip transaction directed at instantiated platform components that should not be accessible to this master component. Without applied APU configuration code, activity-based CF potential transfer that corresponds to such interactions must be assumed. APUs configured according to Section 4.2.2 are able to eliminate this type of CF potential transfer. This conditional relationship is described by rule A07. This rule is valid only if generated APU configuration code is correctly applied to the target:

► **Premise 4.8:** Correct usage of APU configuration code

APU configuration code generated for each $x \in \Omega_X$ is deployed to and successfully executed on this execution platform.

As it was described in Section 4.2.2, an APU configuration allows master components to issue certain transactions via the on-chip interconnect. A consequence of this is that a processing unit will always be able to access peripheral devices that are allocated to its RTE or any of its SWCs. A failure of a processing unit might therefore give rise to activities that result in the misconfiguration of these peripheral devices, which explains rule A08 and rule A09. Furthermore, a processing unit will always have access to the memory region of its RTE. This means that the failure of a processing unit might give rise to activities that corrupt this memory region. Since a dependency-based CF potential transfer into this direction is already in place (cf. rule D02), however, this scenario does not

have to be represented as an activity-based transfer. Peripheral devices from $\hat{Z}$ will always have access to memory regions allocated to them. However, these memory regions are represented by the peripheral device vertex itself; a CF potential transfer is therefore not applicable. Indirect effects the execution platform library captures as Γ_1, Γ_2, or Γ_3 are translated into activity-based transfers by rule A10, rule A11, and rule A12, respectively. Finally, rule A13 captures the fact that a peripheral device will always be able to trigger transactions that traverse the off-chip interconnect attached to this peripheral device.

Another type of activity that processing units and peripheral devices are generally able to initiate are interrupt requests. The occurrence of an interrupt request can influence the control flow executed by a processing unit and, therefore, lead to its failure. From the perspective of each processing unit, however, interrupt handling is an opt-in mechanism. A processing unit is usually able to mask the interrupts that shall not have an influence on its execution. For all specifically activated (i.e., unmasked) interrupts, it is reasonable to assume that their occurrence cannot cause the processing unit to fail. Therefore, the pattern user is expected to enforce the following invariant:

► **Premise 4.9:** Absence of interrupt-induced failures

The correct operation of a processing unit does not depend on the correctness of interrupt requests directed to it.

Under this assumption, it is not necessary to add interrupt-related entries to the activity-based list of rules. In addition to the seven relationships described above, no further activities triggered by a processing unit or a peripheral device are able to cause a direct failure of another system element.

Other system elements The thirteen types of activity-based CF potential transfer have now been described as originating from four kinds of system elements: SWC implementations, RTEs, processing units, and peripheral devices. Failures of the other seven kinds of system elements (such as SWC logic blocks or local paths) cannot trigger activities that cause another system element to fail. This means that they are not the source of activity-based CF potential transfers and, therefore, do not appear on the left-hand side of an 'Impact' column entry in Table 4.2.

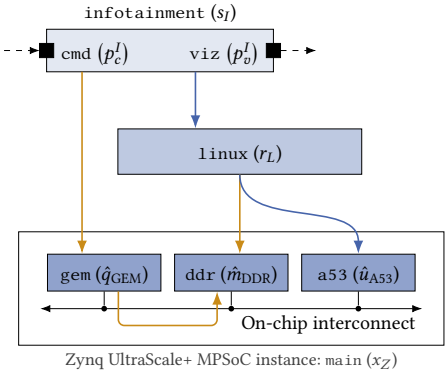


Figure 4.22: Excerpt of the introductory car server example, deliberately simplified by dropping all system model entities related to the Zephyr instance.

4.3.3 CF graph creation

Using the 26 formal rules that were derived in Section 4.3.2, the following algorithm generates a CF graph according to Definition 3.5:

Algorithm 4.9: Algorithm for the creation of a CF graph

```
1 procedure BuildGraph()
2     // Populate vertex sets:
3     V_α ← (Λ^S ∪ P)
4     V_β ← (X ∪ Ĉ_+ ∪ B ∪ R ∪ L ∪ G ∪ S)
5     // Build suitable edge sets:
6     E_δ ← DependencyBasedEdges()
7     E_η ← ActivityBasedEdges()
8     // Construct the final CF graph:
9     G_γ ← ⟨V_α ∪ V_β, E_δ ∪ E_η⟩
```

First, in line 3, the set of application vertices is created by unifying all SWC logic blocks and all SWC ports. Then, in line 4, the context vertices are built by unifying all remaining system elements. In combination, these lines implement the specification from Section 4.1.1.1.

The function DependencyBasedEdges, which is called in line 6, returns the set of dependency-based edges to add to the CF graph. Analogously, the call to ActivityBasedEdges in line 7 returns all activity-based edges added

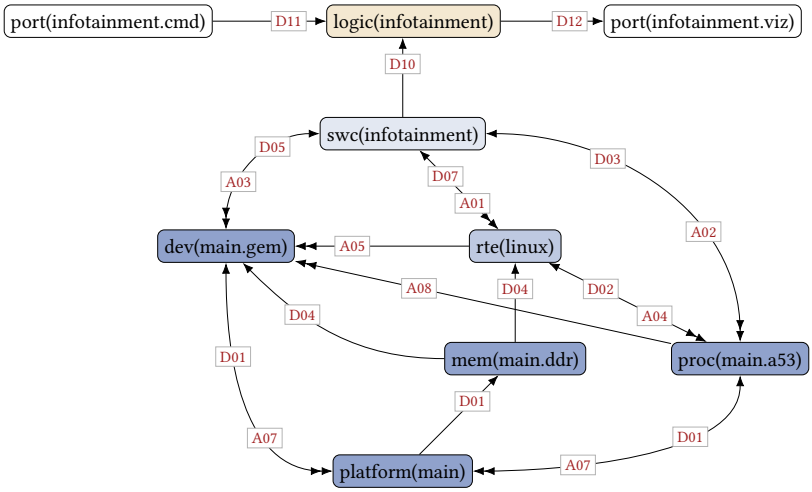


Figure 4.23: CF graph constructed via the direct application of Theorem 4.2. Superimposed rectangles with three-character labels name the CF potential transfer rule that the closest arrow of an edge was created by; they are not part of the graph itself.

to the CF graph. Therefore, the union of both returned sets serves as the edge set of G_γ. As described above, there is some flexibility with respect to what DependencyBasedEdges and ActivityBasedEdges return. Based on Theorem 4.2, the following implementation strategy is feasible:

- For each dependency-based CF potential transfer (from Table 4.1): translate $v \rightsquigarrow v'$ into a dependency-based edge $\langle v, v' \rangle \in E_\delta$.
- For each activity-based CF potential transfer (from Table 4.2): translate $v \rightsquigarrow v'$ into an activity-based edge $\langle v, v' \rangle \in E_\eta$.

▹ Example 4.8: *The system model in Figure 4.22 was generated by dropping all drivetrain and the body control functions from the central car server example. Following the discussion from Section 4.1.1.1, it consists of nine system elements—three from the application and six from the context subgraph. By applying the above edge creation strategy, one obtains the CF graph visualized in Figure 4.23. Rectangles superimposed to graph edges name the rule that is responsible for the existence of a particular arrow. For example, the dependency-based edge from* `swc(infotainment)` *to* `logic(infotainment)` *is due to rule D10, which demands that* $\forall s \, \forall \lambda \, ((s \in S \wedge \lambda \in \text{logic}(s)) \Rightarrow (s \rightsquigarrow \lambda))$.

Adding an edge for every CF potential transfer can result in graphs that are difficult to visualize to the human observer. Even for the comparably simple system model in Example 4.8, the strategy generated a graph with intersecting edges. It is therefore desirable to reduce the number of edges while keeping the semantics of a graph unchanged. Theorem 4.3 can be used to achieve this: if it is possible to show that a CF potential transfer is already covered by a directed path in G_γ, it is not necessary to introduce a dedicated edge to reflect it. Using this observation, it is possible to argue that rule D03 can be neglected by the DependencyBasedEdges function:

1) CF potential transfers described by rule D03 will always be covered by a directed path originating from rule D02 and rule D07.

Analogously, it is possible to show that rule A02, rule A05, rule A08, and rule A09 can be neglected by the ActivityBasedEdges function:

2) CF potential transfers described by rule A02 will always be covered by a directed path originating from rule A01 and rule A04.
3) CF potential transfers described by rule A05 will always be covered by a directed path originating from rule D07 and rule A03.
4) CF potential transfers described by rule A08 will always be covered by a directed path originating from rule D02, rule D07, and rule A03.
5) CF potential transfers described by rule A09 will always be covered by a directed path originating from rule D02 and rule A06.

With this observation, the edge selection functions called in Algorithm 4.9 can be simplified by choosing them as follows:

Algorithm 4.10: Selection of CF graph edges

```
1  function DependencyBasedEdges()
2      E_δ ← ∅
3      for each (v ⇝ v′) due to (rule Dxx with xx ∉ {03}) do
4          E_δ ← E_δ ∪ {⟨v, v′⟩}
5      return E_δ

6  function ActivityBasedEdges()
7      E_η ← ∅
8      for each (v ⇝ v′) due to (rule Axx with xx ∉ {02, 05, 08, 09}) do
9          E_η ← E_η ∪ {⟨v, v′⟩}
10     return E_η
```

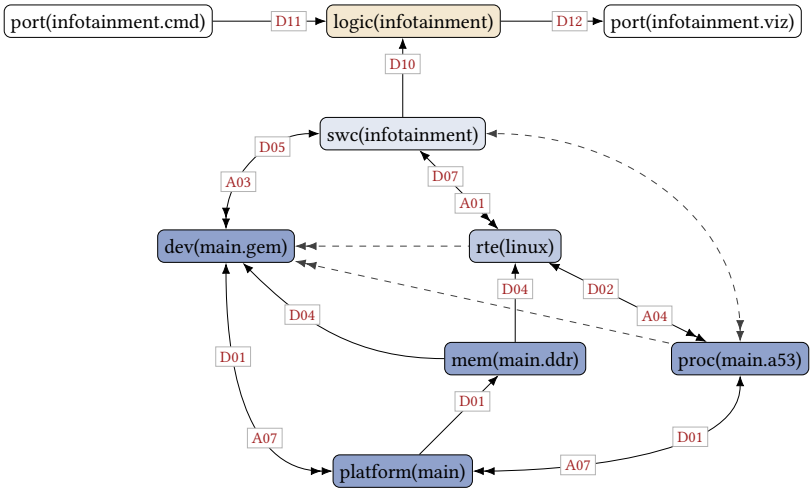


Figure 4.24: CF graph resulting from the omission of rules D03, A02, A05, and A08. Edges removed due to this strategy are shown using dashed arrows. Arrows of remaining edges are again labeled with the underlying CF potential transfer rule.

These definitions lead to the BuildGraph algorithm that is used in the remainder of this thesis. For illustration purposes, this algorithm is now applied to the simplified car server example from before.

▹ Example 4.8 (continued): *Applying the final version of the* BuildGraph *algorithm to the system shown in Figure 4.22 leads to the CF graph in Figure 4.24. Compared with the previous version of this CF graph, a total of four edges have been removed. They were due to rule D03, rule A02, rule A05, and rule A08.*

This algorithm was also used to create the introductory CF graph examples in Section 4.1. It can be insightful to revisit them, for instance, by considering Example 4.1 along with the CF graph in Figure 4.4 on page 77.

▹ Example 4.1 (continued): *In the CF graph, an edge from* `proc(main.a53)` *to* `swc(infotainment)` *does not exist. This is despite the fact that a failure of the Cortex-A53 can clearly have an impact on the SWC implementation, potentially causing it to deviate from its indented behavior. While this CF potential is covered by rule D03, it is not necessary to add a dedicated edge for it: it is already covered by a directed path via* `rte(linux)`.

With this, the CF determination procedure is fully described. Together with the APU configuration approach from Section 4.2.2, this concludes the description of the first safety pattern step. The following section closes this chapter with a brief discussion of selected properties exhibited by the outputs of this step, i.e., by APU configuration code and CF graphs.

4.4 Closing remarks

It is essential to understand that CF graphs capture the *possibility* of a CF between system elements. If a CF graph contains a path from $v \in V$ to $v' \in V$, it is assumed that during runtime, a failure of v will be able to impact v' in an undesired manner. In other words: a directed path from $v \in V$ to $v' \in V$ means that the logical isolation pattern was unable to show that a malfunction of v does not lead to a failure of v'. This is *not* equivalent to the statement that a malfunction of v can actually lead to a failure of v'. In fact, auto-generated CF graphs contain a degree of pessimism that could potentially be reduced with more precise system knowledge.

▹ Example 4.8 (continued): *The auto-generated CF graph in Figure 4.24 contains an edge from* `dev(main.gem)` *to* `swc(infotainment)` *and, therefore, also a directed path from the former to the latter. This path is due to rule D05, which assumes that whenever a SWC or an RTE allocates a peripheral device, its correct functionality depends on the correct functionality of the peripheral device.*

With detailed knowledge about how the SWC uses the Ethernet controller, one might be able to argue that the SWC implementation itself remains unaffected by the controller's failure. The safety pattern is unable to reason about this, however. It therefore employs the worst-case assumption and introduces CF potential transfer. Future work could introduce another type of barrier declaration to capture this (currently unavailable) knowledge.

Another aspect that needs to be emphasized is that CF graphs do not capture information flow. The lack of a directed path from $v \in V$ to $v' \in V$, for example, does *not* justify the statement that v will not have an impact on v'. Among other things, this means that the approach from this chapter cannot be used to reason about the confidentiality of data in an embedded software system. It can be used to reason about the integrity of data, however.

Finally, note that the APU configuration procedure from Section 4.2.2 gives master components either complete or no access to a particular memory region or peripheral device. It does *not* give masters read-only access by default. Technically, doing so would be possible on most commercially available MPSoCs.

Since read-only permissions do not have a logical impact on addressed slave modules, doing so would not even require a modification of the algorithm to create CF graphs. Still, the strategy from Section 4.2.2 does not grant these default permissions; this decision is based on two observations:

- Restrictive transaction filtering at the on-chip interconnect level can support pattern users in the enforcement of Premise 4.6, i.e., in achieving a sufficient degree of temporal isolation at certain interfaces.
- Despite their negligible impact, read transactions directed to incorrect targets are still an anomaly. A restrictive APU configuration facilitates the identification of such anomalies.

In theory, however, it is conceivable to refine the APU configuration procedure in such a way that read transactions are always possible.

Chapter 5

Safety assessment framework

This chapter presents a systematic approach to verify that an embedded software system meets its safety requirements. The application of this approach is the safety assessment, i.e., the second and final step of the pattern.

Safety assessment can be fully automated, just like the combined APU configuration and CF determination procedure from Chapter 4. The value of underlying concepts goes beyond an automated check of whether a system meets its safety requirements, however. As it will be shown in Section 5.4, for example, they can also be used for a safety-aware exploration of the design space. The term safety assessment *framework* is used to emphasize the broad applicability of concepts and ideas presented in this chapter.

According to Premise 1.1 on page 5, this work assumes that the intended functionality of all system elements is safe. In other words: if every system element delivers its intended functionality, the system under consideration must not cause physical harm. Clients of the logical isolation pattern are expected to ensure this, especially for SWC logic blocks. This can be achieved by meeting the requirements of ISO 21448 [25] or similar standards.

With this targeted restriction of scope, the only remaining cause of physical harm is the circumstance that a system element deviates from its intended behavior, i.e., exhibits a failure. To reduce the risk of failure-induced harm to an acceptable level is the topic of functional safety. Achieving functional safety involves the consideration of various topics, such as hazard analysis, risk assessment, fault avoidance, or fault tolerance. Another important topic is how failures are able to propagate from system element to system element. This propagation is exactly what this thesis refers to as CF potential. The

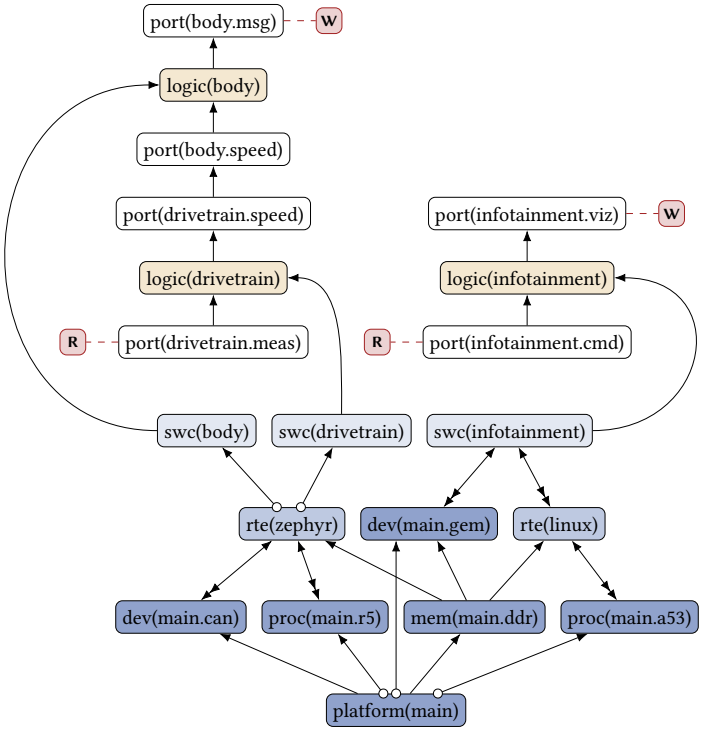


Figure 5.1: CF graph for the car server use case from Example 4.3, here enriched with R/W pins that highlight interactions at environment ports.

procedure from Section 4.2.2 captures CF potential in the form of a CF graph. Given a CF graph, safety assessment seeks to answer the question if the reflected CF potential poses an unreasonable safety risk.

▹ Example 5.1: *In the final version of the partitioned car server use case, two isolation measures were in place: a request to generate APU configuration code for x_Z and the declaration that r_Z implements process isolation (cf. Example 4.3). The corresponding CF graph is shown in Figure 5.1.*

CF graphs do not capture the safety impact that the failure of a particular system element can have. They are also agnostic to the probability that a particular system element fails. They are only concerned with failure propagation.

5.1 Safety impact of CF potential

CF potential does not necessarily constitute a safety issue. For a CF to jeopardize safety, the following chain of events must be fulfilled (cf. Section 3.1.2):

1) The manifestation of a fault leads to the failure of a system element.
2) In the form of a CF, this failure propagates to another system element.
3) This system element affects the environment in a hazardous manner.

While the consideration of item 2 is the responsibility of CF graphs, knowledge about item 1 and item 3 needs to be provided by the pattern user. To create a formal framework that allows them to do so, it is first necessary to specify the system elements that are relevant with respect to these kinds of events.

5.1.1 Safety-relevant system elements

In this thesis, such system elements are referred to as *safety-relevant* system elements. To prepare for their definition, we introduce the abbreviations

$$\tilde{P}_I = \{p \in P : \text{scope}(p) = \text{Env} \wedge \text{dir}(p) = \text{In}\} \text{ and}$$
$$\tilde{P}_O = \{p \in P : \text{scope}(p) = \text{Env} \wedge \text{dir}(p) = \text{Out}\}.$$

As given by the formal statements above, $\tilde{P}_I$ refers to all environment inputs of SWCs, while $\tilde{P}_O$ refers to all environment outputs of SWCs.

In line with the fault model from Section 3.3, only a subset of system elements is susceptible to faults. This subset is defined as follows:

► **Definition 5.1:** Susceptible system elements

The set of *susceptible system elements*,

$$V_s = (X \cup \hat{C}_+ \cup B) \cup (R) \cup (S \cup \Lambda^S \cup \tilde{P}_I),$$

contains all system elements that may be subject to random or systematic faults, regardless of the likelihood that a fault exists or manifests.

An example of a random fault affecting an execution platform, $x \in X$, is an ionizing particle that falsifies a transaction traversing the on-chip interconnect of x. An example of a systematic fault of environment input $p \in \tilde{P}_I$ is the

circumstance that p reads from an unreliable sensor and, therefore, might introduce erroneous data into the system.

Physical harm can only be caused by system elements that affect the environment. They are referred to as *environment writers* and, following the underlying system model, defined as follows:

> ► **Definition 5.2:** Environment writers
>
> The set of *environment writers*,
>
> $$V_w = \tilde{P}_O,$$
>
> contains all environment output ports, regardless of the physical harm that can actually be caused by these interfaces.

One example of an environment writer is a SWC output port that controls the door locks of a road vehicle using multiple actuators.

▹ Remark 5.1: *At this point, it is important to emphasize the virtual nature of elements in $\tilde{P}_I$ and $\tilde{P}_O$. Like all SWC ports, they are logical entities that allow the pattern user to capture relevant SWC behavior. Physical components that allow an environment input (or output) to actually read (or write) values are represented by other system elements, including SWC implementations, underlying RTEs, and utilized I/O controllers. In the CF graph, there will always be a directed path from these system elements to logic blocks reading environment inputs and writing environment outputs, respectively.*

5.1.2 Fault manifestation and physical harm

V_s and V_w are disjoint sets. This means that a system element susceptible to faults (V_s) is not an environment writer (V_w) and, therefore, unable to cause physical harm. The converse is also true: an environment writer (V_w) is not a susceptible system element (V_s) and will therefore, by itself, not be affected by a fault. This leads to two important observations:

1) Physical harm can only be the result of a CF that eventually causes the failure of an environment writer, i.e., an element of V_w.
2) The root cause (i.e., the fault) that gives rise to a CF can only originate from a susceptible system element, i.e., an element of V_s.

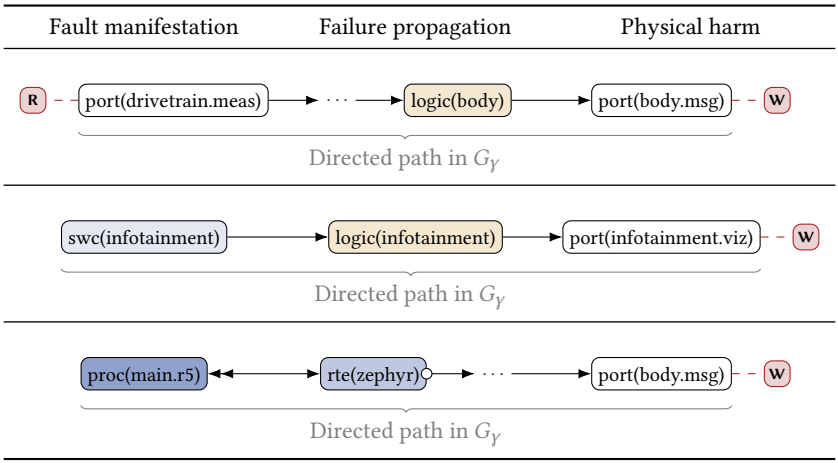


Table 5.1: Sample paths in G_γ that lead from fault manifestation (at a susceptible system element) to physical harm (at an environment output of a SWC).

If we combine these two observations, it is possible to state the following: physical harm can only result from a directed path in G_γ that starts at a susceptible system element and reaches an environment writer.

▹ Example 5.1 (continued): *In the partitioned car server use case, there are two environment writers:* p^B_m*, which is also labeled as* `port(body.msg)`*, and* p^I_v*, which is also labeled as* `port(infotainment.viz)`*.*

Table 5.1 shows three paths from the corresponding CF graph. The first one starts at `port(drivetrain.meas)` *and ends at* `port(body.msg)`*. Whether this CF potential leads to an unreasonable safety risk depends on (1) the integrity of values introduced by* `port(drivetrain.meas)` *and (2) the criticality of outputs generated by* `port(body.msg)`*. For the purposes of this example, we assume that these two factors are in line with each other. Therefore, the discussed CF potential is not a reason to reject the system design.*

The other two paths visualized in Table 5.1 originate from a SWC implementation and a processing unit, respectively. In this case, to assess the presence or absence of unreasonable safety risk, it needs to be answered (1) how the integrity of `swc(infotainment)` *relates to the criticality of* `port(infotainment.viz)`*, and (2) how the integrity provided by* `port(main.r5)` *relates to the criticality of* `port(body.msg)`*. Under the assumption that these properties are again pairwise compatible, the two CF potentials can be accepted.*

When the criticality of environment writers is assessed, state-of-the-art methods such as a Hazard Analysis and Risk Assessment (HARA) can be applied. Analogously, when the integrity of susceptible system elements is determined, the application of fault avoidance or fault tolerance mechanisms should be considered. For example, a SWC implementation performed using a memory-safe programming language might be assigned a higher integrity than a SWC implementation based on a programming language that does not provide memory safety. Another example is lockstep functionality implemented by a processing unit: it might be justified to consider the integrity provided by such a processing unit higher than that of an unprotected one.

These relationships can be *captured* by the logical isolation pattern, but their determination is the responsibility of the pattern user. To allow users to communicate them in an unambiguous manner, the pattern accepts them in the form of a formal *safety requirements* specification. As introduced in Section 3.1.2, this work proposes two alternative specification approaches: the interference whitelist and a lattice-based integrity assignment concept. Each of them is associated with a corresponding assessment procedure and will now be described in a dedicated section.

5.2 Interference whitelist approach

This approach is based on an enumeration of acceptable CF potential between susceptible system elements and environment writers. It is referred to as an *interference whitelist*, where *interference* is used as a synonym for CF.

5.2.1 Safety requirements specification

Formally, interference whitelists are binary relations:

► **Definition 5.3:** Interference whitelist

The *interference whitelist* is a binary relation

$$\Sigma \subseteq V_s \times V_w,$$

where $\langle v, v' \rangle \in \Sigma$ implies ($\Rightarrow$) that CF potential from v to v' does not pose an unreasonable safety risk.

To decide whether a system element pair can be added to Σ, the corresponding CF potential needs to be considered on its own. This is an important contributor to the applicability of the approach, since it allows pattern users to focus on *one pair* of system elements at a time.

5.2.2 Assessment algorithm

To ensure that a system design is free from unreasonable safety risk, the assessment procedure traverses the CF graph from every susceptible system element, identifies all reachable environment writers, and compares the resulting CF potential to the interference whitelist:

Algorithm 5.1: Safety assessment based on an interference whitelist

```
1 function SafetyAssessment()
2     for each v ∈ V_s do
3         for each v′ ∈ DepthFirstSearch(G_Y, v) do
4             if (v′ ∈ V_w) ∧ (⟨v, v′⟩ ∉ Σ) then
5                 return false
6     return true
```

The binary result returned by Algorithm 5.1 emphasizes that safety assessment is a decision problem, not a search or optimization problem. In the algorithm, a return value of 'true' means that the system design is accepted; a return value of 'false' communicates the rejection of the system design.

Time complexity The loop in line 2 is executed up to $|V_s|$ times. For each iteration of this outer loop, line 3 applies Depth-First Search (DFS) to iterate over each system element reachable from v; under the assumption that edges are stored in the form of an adjacency list, each DFS has a worst-case performance of $O(|V| + |E|)$. Based on the DFS result, in each iteration of the outer loop, line 4 is executed up to $|V|$ times. Under the assumption that this line can be executed in constant time, we obtain a total worst-case time complexity of $O(|V_s|(|V| + |E| + |V|)) = O(|V|^2 + |V||E|)$.

Space complexity In addition to memory used by relevant inputs, the worst-case space complexity of Algorithm 5.1 is $O(|V|)$. This is due to the application of DFS, which needs to maintain a vertex stack.

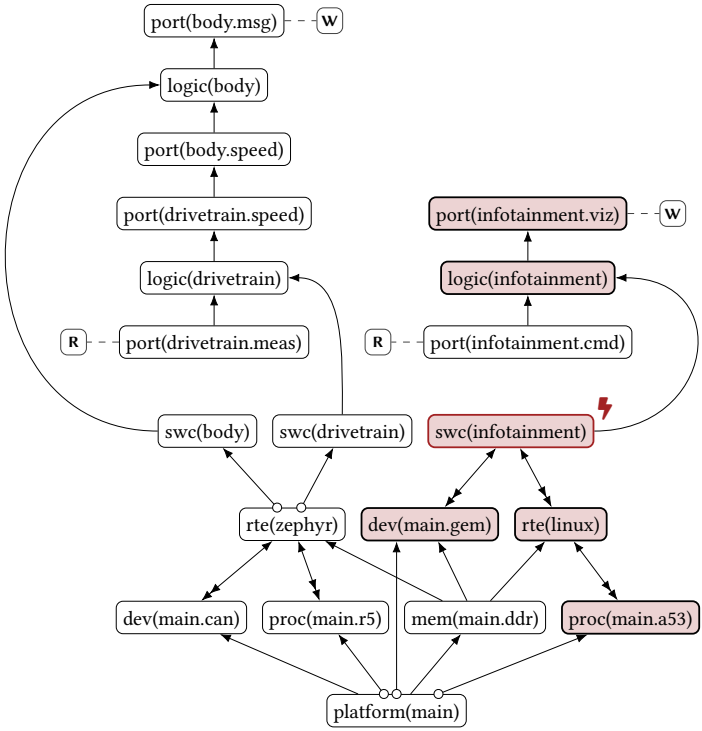


Figure 5.2: CF potential spawned by `swc(infotainment)` in the protected car server use case. A lightning bolt highlights this arbitrarily selected source vertex. Vertices reachable from it are outlined black and shaded red.

▹ Example 5.2: *For illustration purposes, we now determine an interference whitelist for which a safety assessment of the partitioned car server use case (cf. Example 5.1) succeeds. Therefore, it is first necessary to identify all CF potential between the 16 susceptible system elements and the two environment writers. Starting from* `swc(infotainment)`*, for instance, the CF graph contains a directed path only to* `port(infotainment.viz)`*. This is graphically emphasized in Figure 5.2 and documented in a row of Table 5.2.*

Repeating this process for the other $v \in V_s$ completes Table 5.2. It captures the entire CF potential that is relevant from a safety assessment perspective.

For the safety assessment to succeed, Σ needs to contain all system element pairs for which Table 5.2 reports a 'reachable (●)' entry. To minimize the cardi-

Susceptible system element	Environment writer[a]	
	`port(infotainment.viz)`	`port(body.msg)`
`platform(main)`	●	●
`mem(main.ddr)`	●	●
`proc(main.a53)`	●	○
`dev(main.gem)`	●	○
`rte(linux)`	●	○
`swc(infotainment)`	●	○
`logic(infotainment)`	●	○
`port(infotainment.cmd)`	●	○
`proc(main.r5)`	○	●
`dev(main.can)`	○	●
`rte(zephyr)`	○	●
`swc(body)`	○	●
`swc(drivetrain)`	○	●
`logic(body)`	○	●
`logic(drivetrain)`	○	●
`port(drivetrain.meas)`	○	●

[a] Legend: ● reachable (⇒ there is CF potential from the susceptible system element on the left to this environment writer), ○ unreachable (⇒ there is no such CF potential).

Table 5.2: Complete enumeration of CF potential between susceptible system elements and environment writers in the car server use case.

nality of $|\Sigma|$*, we do not add any further system element pairs and obtain a* Σ *value populated with 18 elements:*

$$\Sigma = \Big\{\langle x_Z,\, p_v^I\rangle,\, \langle \hat{m}_{DDR},\, p_v^I\rangle,\, \langle \hat{u}_{A53},\, p_v^I\rangle,\, \langle \hat{q}_{GEM},\, p_v^I\rangle,\, \ldots,$$
$$\langle x_Z,\, p_m^B\rangle,\, \langle \hat{m}_{DDR},\, p_m^B\rangle,\, \langle \hat{u}_{R5},\, p_m^B\rangle,\, \langle \hat{q}_{CAN},\, p_m^B\rangle,\, \ldots\Big\}.$$

If the pattern user declares these safety requirements, Algorithm 5.1 returns a positive (true) result. At the same time, it is the user's responsibility to ensure that the integrities and criticalities of each listed system element pair are in line with each other. For instance, $\langle \hat{u}_{R5},\, p_m^B\rangle \in \Sigma$ *communicates to the pattern that CF potential from the Cortex-R5 to the body network is acceptable. This statement can only be made if the Cortex-R5 has sufficient integrity to control the body network, and its validity needs to be enforced by the user.*

5.3 Integrity assignment procedure

The interference whitelist is a flexible approach to specify safety requirements in a fine-grained manner. From a usability perspective, however, defining this whitelist for designs with numerous system elements can be difficult. To specify Σ, safety considerations for a total of $|V_s||V_w|$ system elements pairs need to be performed. In other words: to specify an interference whitelist, the pattern user is expected to take $O(|V|^2)$ decisions.

This section presents a safety assessment procedure that attempts to solve this usability issue. It is inspired by the concept of lattice-based information flow tracking, but it reasons about CF potential instead of general information flow. The procedure was initially contributed by the author to [4] and is now described in a tailored and extended version.

5.3.1 Safety requirements specification

This specification strategy allows pattern users to define an arbitrary set of *integrity levels*, where each level quantifies the 'certainty' that no fault manifestation will affect a susceptible system element during runtime. These integrity levels must be partially ordered and form a meet-semilattice.

Based on this framework, susceptible system elements are labeled with the integrity level they *provide*, while environment writers are labeled with the integrity level they *require* to avoid unreasonable safety risk:

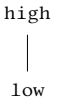
► **Definition 5.4:** Integrity assignment

The *integrity assignment* is a tuple

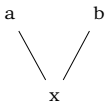
$$\langle I, \leq, \text{int}, \text{ireq} \rangle$$

in which:

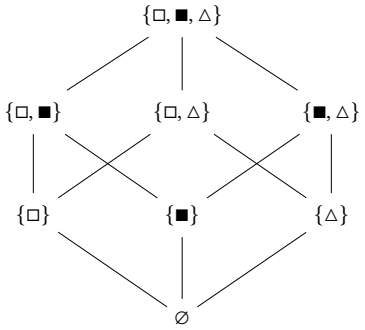
1) I is a finite set of *integrity levels*,
2) the partial order $\leq$ turns $\langle I, \leq \rangle$ into a meet-semilattice,
3) $\text{int}\colon V_s \to I$ specifies the *inherent integrity* of susceptible system elements, i.e., the integrity level they provide by themselves, and
4) $\text{ireq}\colon V_w \to I$ maps every $w \in V_w$ to a *required integrity* such that for each $\langle v, w \rangle \in V_s \times V_w$, if $\text{ireq}(w) \leq \text{int}(v)$, then CF potential from v to w does not pose an unreasonable safety risk.



(a) $I = \{\texttt{low}, \texttt{high}\}$ with $\texttt{low} \leq \texttt{high}$, $\texttt{low} \leq \texttt{low}$, and $\texttt{high} \leq \texttt{high}$.



(b) $I = \{\texttt{x}, \texttt{a}, \texttt{b}\}$ with $\texttt{x} \leq \texttt{a}$, $\texttt{x} \leq \texttt{b}$, $\texttt{a} \leq \texttt{a}$, $\texttt{b} \leq \texttt{b}$, and $\texttt{x} \leq \texttt{x}$.



(c) $I = \mathscr{P}(\{\square, \blacksquare, \triangle\})$, where $\leq$ represents set containment ($\subseteq$).

Figure 5.3: Hasse diagrams of three arbitrarily chosen integrity lattices. The first (a) constitutes a total order, while the third (c) leverages concepts from set theory.

For the sake of brevity, each $\langle I, \leq \rangle$ tuple that meets Definition 5.4 will be referred to as an *integrity lattice* in the following, although this tuple does not necessarily have to be a lattice in the mathematical sense. As stated above, it is sufficient that $\langle I, \leq \rangle$ is a meet-semilattice; it is not necessary that there is a least upper bound $\sup\{i_1, i_2\}$ for all $i_1, i_2 \in I$.

▹ Example 5.3: *Three possible integrity lattices are shown in Figure 5.3.*

The integrity lattice in Figure 5.3a is built from a total order on $\{\texttt{low}, \texttt{high}\}$. *These integrity levels represent low and high integrity, respectively. Due to the fact that* $\texttt{low} \leq \texttt{high}$, *a susceptible system element with an inherent integrity of* high *is less likely to fail due to an inherent root cause than one with* low.

The definition in Figure 5.3b consists of three elements, where a *and* b *describe different, incomparable fault manifestation characteristics. However, susceptible system elements labeled with* x *are more likely to fail due to an inherent root cause than one labeled with either* a *or* b.

Finally, the integrity lattice in Figure 5.3c is based on three 'beneficial' properties symbolized by □, ■, *and* △, *respectively. Semantically, each property expresses robustness against a certain type of fault manifestations. If* □ *is used to express the absence from systematic programming errors, and* ■ *signifies robustness against single-event upsets, then an inherent integrity of* {□, ■} *means that a susceptible system element exhibits both of these properties.*

With respect to usability, it is again interesting to discuss the effort that pattern users need to invest for the specification of an integrity assignment. Under the assumption that the integrity lattice is already defined, integrity levels need to be assigned to $|V_s| + |V_w|$ system elements. For each such assignment, the pattern user needs to find the suitable 'int' or 'ireq' value among $|I|$ options. Therefore, to specify an integrity assignment, the pattern user is expected to take $O(|V||I|)$ decisions.

Before moving on to the assessment procedure that decides if inherent and required integrities are actually in line with each other, we briefly discuss how the integrity assignment approach relates to the interference whitelist method. In fact, the two specification approaches have the same expressivity:

► **Theorem 5.1:** Expressivity of specification approaches

Interference whitelists (according to Definition 5.3) and integrity assignments (according to Definition 5.4) are interchangeable.

▹ Proof: It is necessary to show that every interference whitelist can be transformed into a semantically equivalent integrity assignment, and vice versa.

Starting with an interference whitelist Σ, we set I to $\mathscr{P}(V_w)$ and define $\leq$ as set containment ($\subseteq$) over I. With this, $\langle I, \leq \rangle$ is a meet-semilattice, each integrity level is a set of environment writers, and the greatest lower bound of $i_1 \in I$ and $i_2 \in I$ is $i_1 \cap i_2$. We choose 'int' such that

$$\forall v \in V_s\colon\ \mathrm{int}(v) = \{w \in V_w : \langle v, w \rangle \in \Sigma\}.$$

After this assignment, per Definition 5.3, it is possible to state the following: for all $\langle v, w \rangle \in V_s \times V_w$, if $w \in \mathrm{int}(v)$, then CF potential from v to w does not pose an unreasonable safety risk. Since $\leq$ represents set containment, this statement can be rewritten as follows: for all $\langle v, w \rangle \in V_s \times V_w$, if $\{w\} \leq \mathrm{int}(v)$, then CF potential from v to w does not pose an unreasonable safety risk. We complete the integrity assignment specification by defining 'ireq' as follows:

$$\forall w \in V_w\colon\ \mathrm{ireq}(w) = \{w\}.$$

Substituting this into the previous intermediate result, we obtain precisely the constraint from item 4 of Definition 5.4. Therefore, the derived integrity assignment $\langle I, \leq, \text{int}, \text{ireq}\rangle$ is semantically equivalent to Σ.

Into the opposite direction, we start with $\langle I, \leq, \text{int}, \text{ireq}\rangle$ and need to show that this safety requirements specification can be transformed into an integrity whitelist. From item 4 of Definition 5.4, it is obvious that for a precisely defined subset of $V_s \times V_w$, the integrity assignment argues that CF potential does not pose an unreasonable safety risk. This is exactly the subset that needs to be assigned to Σ to achieve semantic equivalence. □

▹ Remark 5.2: *The proof of Theorem 5.1 utilizes the fact then when an integrity assignment is created, it is possible to choose both I and $\leq$ arbitrarily. In practice, it is often more manageable to limit I to a few elements. If such a constraint is enforced, it may no longer necessarily possible to transform an interference whitelist into an integrity assignment.*

The integrity assignment can be seen as a generalization of the interference whitelist: in principle, it is as powerful as the interference whitelist (cf. Theorem 5.1), but this expressivity can be reduced to make the safety requirements specification more manageable.

5.3.2 Assessment algorithm

To argue that CF potential from G_γ does not pose an unreasonable safety risk, it is necessary to make use of 'int' and 'ireq' values. Intuitively speaking, the assessment algorithm applies the following strategy:

1) Initialize the integrity level of each $v \in V_s$ with $\text{int}(v)$.
2) Propagate integrity levels along the edges of G_γ; when integrity levels converge, calculate and forward their greatest lower bound.
3) Ensure that the integrity level of each $w \in V_w$ is at least $\text{ireq}(w)$, i.e., the required integrity of the environment writer.

Therefore, the algorithm first extends $\langle I, \leq\rangle$ with a greatest element $\top$, also referred to as the 'top' element. This is achieved by first setting

$$I_\top = I \cup \{\top\}$$

and declaring $i \leq \top$ for all $i \in I_\top$. This results in an extended integrity lattice $\langle I_\top, \leq\rangle$, in which $\top$ is the meet ($\wedge$) of the empty set. All system elements that are *not* part of V_s can be initialized with this value to ensure that they do not interfere with the integrity propagation process.

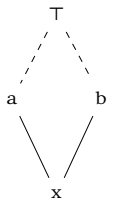


Figure 5.4: Semilattice from Figure 5.3b extended with a top ($\top$) element, which leads to $I_\top = \{\mathrm{x}, \mathrm{a}, \mathrm{b}, \top\}$. With this, $\top$ is the meet of the empty set.

▹ Example 5.4: *An extension of the integrity lattice from Figure 5.3b leads to the Hasse diagram shown in Figure 5.4. As shown in the diagram, the top element ($\top$) is greater than or equal to every $i \in \{\mathrm{x}, \mathrm{a}, \mathrm{b}, \top\}$.*

Based on $I_\top$, the following auxiliary concept is introduced:

► Definition 5.5: Effective integrity

The *effective integrity* of $v_j \in V$ is

$$i_{j,0} \wedge \left(\bigwedge_{v^* \in V_v^*} \mathrm{int}(v^*) \right) \in I_\top,$$

where V_v^* are the susceptible system elements from which there is CF potential to v_j, and $i_{j,0}$ is the *initial integrity* of v_j. If v_j is a susceptible system element, then $i_{j,0}$ is $\mathrm{int}(v_j)$; otherwise, $i_{j,0}$ is $\top$.

▹ Remark 5.3: *If a system element $v_j \in V$ has no incoming CF potential from any susceptible system element, its effective integrity is its initial integrity.*

Effective integrities of environment writers are exactly the values that need to be compared with 'ireq' values in order to reason about the absence of unreasonable safety risk. From a dynamic programming perspective, their calculation is facilitated by the effective integrity of all $v \in V \setminus V_w$.

Based on this observation, the integrity propagation strategy can be reduced to the determination of an effective integrity for each system element.

The following algorithm performs this determination and, finally, compares the effective integrities of all environment writers to their requirements:

Algorithm 5.2: Safety assessment based on an integrity lattice

```
1  function SafetyAssessment()
2      // Preallocate a map for effective integrities:
3      for each v ∈ V do
4          levels[v] ← ⊤
5      // Populate the 'levels' map and keep track of modified values:
6      for each v ∈ V_s do
7          levels[v] ← int(v)
8      dirtySet ← V_s
9      // Propagate effective integrities until there are no further changes:
10     while dirtySet ≠ ∅ do
11         // Get a system element whose effective integrity is not propagated:
12         v ← (arbitrary element from dirtySet)
13         dirtySet ← dirtySet \ {v}
14         // Propagate it via all outgoing edges:
15         for each w ∈ V : ⟨v, w⟩ ∈ E do
16             prev ← levels[w]
17             levels[w] ← (levels[v] ∧ levels[w])
18             if levels[w] ≠ prev then
19                 dirtySet ← dirtySet ∪ {w}
20     // Ensure that required integrities are met:
21     for each w ∈ V_w do
22         if ireq(w) ⋢ levels[w] then
23             return false
24     return true
```

To determine the effective integrity of all system elements, the algorithm performs an iterative procedure that operates on two data structures:

1) *levels* is a map that finally stores the effective integrity of each system element; while the algorithm is in progress, it stores values that are temporarily assumed to be effective integrities of system elements.
2) *dirtySet* is a set holding every system element whose current *levels* value has not yet been propagated via its outgoing edges.

After the loops in line 3 and line 6, the *levels* map is initialized as follows:

$$\forall v \in V\colon \mathit{levels}[v] = \begin{cases} \mathrm{int}(v), & v \in V_s, \\ \top, & \text{otherwise.} \end{cases}$$

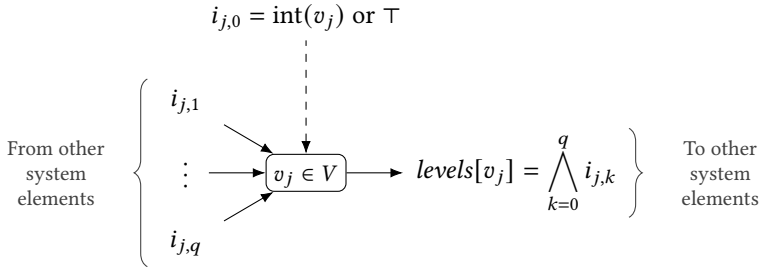


Figure 5.5: Local invariant to maintain for each system element. If $v_j \in V_s$, then $i_{j,0}$ is the inherent integrity of v_j; otherwise, it is the top element $\top$.

Then, during its iterative procedure, Algorithm 5.2 maintains the following invariant: for each $v_j \in V$, $levels[v_j]$ is the greatest lower bound of its initial value $i_{j,0}$ and $i_{j,1}, \ldots, i_{j,q} \in I_\top$. Here, q is the number of CF graph edges that lead to v_j, and $i_{j,k}$ has the following value:

1) If the system element that edge k originates from is *not* contained in *dirtySet*, then $i_{j,k}$ is the *levels* value of this system element.
2) If the system element that edge k originates from *is* in *dirtySet*, then $i_{j,k}$ is greater than or equal to the *levels* value of this system element.

These relationships are illustrated in Figure 5.5. *dirtySet* allows us to reduce a *levels* value at any time: as long as the respective system element is also added to *dirtySet*, the invariant will be maintained. To remove a system element v from *dirtySet*, it needs to be ensured that the *levels* values of its successors are (individually) reduced to at most $levels[v]$. When *dirtySet* is empty, the invariant simplifies to the following: for each $v \in V$, $levels[v]$ is the greatest lower bound of its initial integrity and the *levels* values of all predecessors of v. At this point, $levels[v]$ is exactly the effective integrity of $v \in V$.

These concepts are now mapped to Algorithm 5.2. After the loop in line 3, every *levels* value is $\top$, and the invariant is trivially met. The loop in line 6 reduces the *levels* value of each susceptible system element. By simultaneously adding these system elements to *dirtySet* in line 8, the invariant is maintained. The loop in line 10 iterates until the *dirtySet* is empty. In each iteration, an arbitrarily selected *dirtySet* element is removed. To ensure that the invariant is maintained, all outgoing edges of this $v \in V$ need to be considered, and the *levels* value of each reachable system element needs to be reduced to at most $levels[v]$. This is achieved by the loop in line 15 and, particularly, the

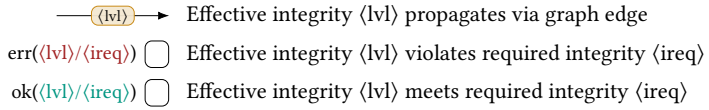


Figure 5.6: Notation used for the annotation of lattice-based safety assessment results. Elements in angular brackets are placeholders that represent a particular $i \in I_\top$.

assignment in line 17. If this assignment leads to a reduction of a *levels* value, the affected system element is again added to *dirtySet* in line 18. Eventually, this procedure will result in a steady state, which means that $dirtySet = \emptyset$. When this is the case, the loop in line 10 terminates, and the effective integrity of each system element v is available as $levels[v]$.

The final task of Algorithm 5.2 is to compare effective integrities of environment writers to 'ireq' values. This is performed by the loop in line 21:

1) An effective integrity of $levels[w] = \top$ means that there is no CF potential from a susceptible system element to $w \in V_w$. Since the top element $\top$ is at least as high as every $i \in I$, this case will never lead to a rejection of the design, i.e., it will not lead to a 'false' return value.
2) An effective integrity of $levels[w] \neq \top$ means that there is at least one susceptible system element with CF potential leading to $w \in V_w$. In this case, per Definition 5.5, $levels[w]$ is the greatest lower bound of all inherent integrities assigned to these system elements. If $\text{ireq}(w) \leq levels[w]$, per Definition 5.4, it is possible to infer the absence of an unreasonable safety risk. Therefore, this condition does also not lead to the rejection of a design. If $\text{ireq}(w) \not\leq levels[w]$, however, then there is CF potential from a susceptible system element v^* for which $\text{ireq}(w) \not\leq \text{int}(v^*)$. In this case, absence of unreasonable safety risk can no longer be inferred, and the design will be rejected in line 23.

Algorithm 5.2 returns its decision as a binary result. To provide active support during the design process, it can further be insightful to return an annotated version of a CF graph that provides pattern users with more details about the assessment result. The legend in Figure 5.6 shows a notation that this thesis uses for such a visual annotation of safety assessment results.

To analyze the computational complexity of Algorithm 5.2, the following paragraphs assume that the meet operation is precomputed (for constant-time lookups), that each $v \in V$ is associated with a unique index from 0 to $|V| - 1$, that *levels* is stored as an array using these indices, and that *dirtySet* combines

two arrays to achieve constant-time addition and removal of set elements. To facilitate this, addition to *dirtySet* is limited to elements from V, while removal from *dirtySet* returns and deletes an arbitrary (random) element.

Time complexity The time complexity is dominated by the iterative integrity level propagation. The outer loop in line 10 is executed for as long as elements can be removed from *dirtySet*. Over the course of the algorithm, a system element is added to *dirtySet* whenever its *levels* value is reduced. In the worst case, this occurs up to $|I|$ times per system element. Therefore, in total, the following two segments are executed up to $|V||I|$ times:

1) The set operations in line 12 and line 13, which can be executed in constant time. For all iterations of the outer loop, they therefore have a worst-case time complexity of $O(|V||I|)$.
2) The inner loop in line 15, whose body can again be executed in constant time. It iterates over all vertices adjacent to the respective $v \in V$. In all iterations of the outer loop combined, it visits each $e \in E$ up to $|I|$ times, which leads to a worst-case time complexity of $O(|E||I|)$.

In total, we obtain a worst-case time complexity of $O(|V||I| + |E||I|)$.

Space complexity In addition to external memory, which also holds the precomputed meet operation table, the algorithm requires $O(|V|)$ space to store *levels* and *dirtySet* using suitable data structures.

▷ Example 5.5: *Based on the partitioned car server use case (cf. Example 5.1), we now discuss an integrity assignment that reflects relevant safety requirements.*

The use case consists of two environment writers: p_m^B, which controls the safety-critical body network, and p_v^I, which is a convenience feature without the potential to cause physical harm. Therefore, it makes sense to introduce the following integrity lattice: $I = \{\texttt{low}, \texttt{high}\}$ with $\texttt{low} \leq \texttt{high}$, $\texttt{low} \leq \texttt{low}$, and $\texttt{high} \leq \texttt{high}$. Here, $\texttt{high}$ is the inherent integrity that susceptible system elements with CF potential to p_m^B need. Susceptible system elements labeled with $\texttt{low}$ have an unknown or unpredictable fault manifestation behavior.

A possible assignment of 'int' and 'ireq' values is shown in Table 5.3. In line with the above statements, it defines $\text{ireq}(p_m^B) = \texttt{high}$ and $\text{ireq}(p_v^I) = \texttt{low}$. It further specifies, e.g., that r_L has an inherent integrity of only $\texttt{low}$; in other words: a failure of this RTE is considered more likely than it would have to be to accept CF potential from r_L to p_m^B or, more generally, to accept CF potential to any environment writer with a required integrity of $\texttt{high}$.

System element $v \in V$	Integrity assignment int(v)	ireq(v)
`platform(main)`	`high`	–
`mem(main.ddr)`	`high`	–
`proc(main.a53)`	`low`	–
`proc(main.r5)`	`high`	–
`dev(main.gem)`	`low`	–
`dev(main.can)`	`high`	–
`rte(linux)`	`low`	–
`rte(zephyr)`	`high`	–
`swc(infotainment)`	`low`	–
`swc(body)`	`high`	–
`swc(drivetrain)`	`high`	–
`logic(infotainment)`	`low`	–
`logic(body)`	`high`	–
`logic(drivetrain)`	`high`	–
`port(infotainment.cmd)`	`low`	–
`port(drivetrain.meas)`	`high`	–
`port(drivetrain.speed)`	–	–
`port(body.speed)`	–	–
`port(infotainment.viz)`	–	`low`
`port(body.msg)`	–	`high`

Table 5.3: Sample assignment of inherent and required integrities for the partitioned car server use case. Two of the 20 system elements are neither in V_s nor in V_w; these system elements are not associated with any integrity level.

For this integrity assignment, the safety assignment returns a positive result, i.e., it argues the absence of unreasonable safety risk. The effective integrities that propagate via CF graph edges are shown in Figure 5.7.

The annotated graph also visualizes how the effective integrity of each environment writer relates to the required one: p_m^B requires an effective integrity of at least `high`*, for example, and has an effective integrity of* `high`*.*

▹ Remark 5.4: *Although effective integrities are a property of system elements, the notation from Figure 5.6 annotates them to outgoing edges of vertices. This is an attempt to simplify the human observer's understanding of how effective integrity levels propagate through the CF graph. For bidirectional edges, the effective integrity that propagates into both directions is shown only once.*

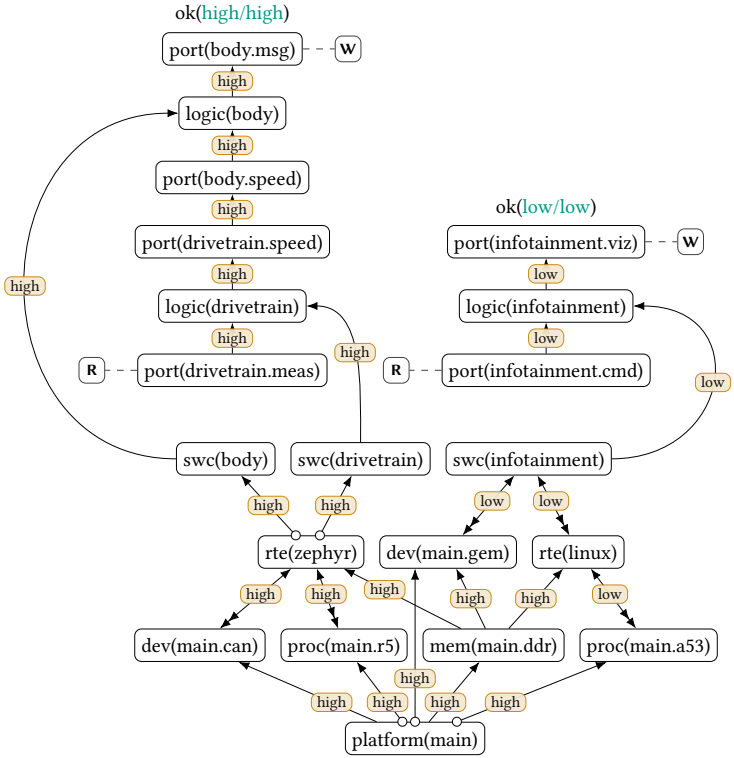


Figure 5.7: CF graph of the partitioned car server use case with safety assessment result annotations. The assessment is based on $I = \{\texttt{low}, \texttt{high}\}$ with $\texttt{low} \leq \texttt{high}$, $\texttt{low} \leq \texttt{low}$, and $\texttt{high} \leq \texttt{high}$. It leads to a positive result, i.e., the acceptance of the design.

▹ Example 5.5 (continued): *Without an APU protection of x_Z, the safety assessment would not lead to a positive result. In this case, the integrity '*$\texttt{low}$*' would propagate from $\hat{u}_{A53}$ via x_Z to the safety-critical environment writer p_m^B.*

5.4 Safety-aware system design using ILP

By the logical isolation pattern itself, safety assessment is treated as a decision problem. This section leverages ILP to show how the underlying concept can be used to solve search or optimization problems. The approach is *guided* by

specified safety requirements and is meant to support pattern users in taking certain design decisions. It is applicable to all pattern inputs where:

1) Safety requirements are specified in the form of an integrity lattice.
2) The binary relation $\leq$ is a total order.

In other words: using $\leq$, any two integrity levels from I must now be comparable. This is a restriction on arbitrary integrity lattices, which only require that any two integrity levels have a greatest lower bound.

This allows us to associate each integrity level with a unique integer such that greater integers correspond to greater integrity levels. Formally, this is achieved by defining a one-to-one function $\pi\colon I \to \mathbb{Z}$ such that for all $i_j, i_k \in I$, $i_j \leq i_k \Leftrightarrow \pi(i_j) \leq \pi(i_k)$. In the following and without loss of generality, the integer set $\{0, \ldots, |I| - 1\} \subseteq \mathbb{Z}$ will be used as the codomain of π.

▹ Example 5.5 (continued): $I = \{\texttt{low}, \texttt{high}\}$ *with* $\texttt{low} \leq \texttt{high}$, $\texttt{low} \leq \texttt{low}$, *and* $\texttt{high} \leq \texttt{high}$ *is a totally ordered set. Following the convention from above, each integrity level must be mapped to a unique integer from* $\{0, 1\}$. *We choose* $\pi(\texttt{low}) = 0$ *and* $\pi(\texttt{high}) = 1$ *to satisfy* $\texttt{low} \leq \texttt{high} \Leftrightarrow 0 \leq 1$.

5.4.1 LP formulation of the safety assessment

As foundation, we first discuss how the decision problem can be formulated using ILP. Therefore, we introduce a vector $\boldsymbol{\sigma} \in \mathbb{Z}^n$, where $n = |V|$, and every element of $\boldsymbol{\sigma}$ represents an integrity level associated with one particular system element. In the following, we use subscript indices to communicate the mapping between a system element and its associated integrity level: the integrity level associated with $v_j \in V$, $j = 1, \ldots, n$, is referred to as σ_j.

A positive safety assessment can be argued if and only if there is a vector $\boldsymbol{\sigma} \in \mathbb{Z}^n$ that fulfills the following constraints:

$$\forall \langle v_j, v_k \rangle \in E\colon \quad \sigma_k - \sigma_j \leq 0, \tag{5.1}$$

$$\forall v_j \in V_s\colon \quad \sigma_j \leq \mathrm{int}'(v_j), \tag{5.2}$$

$$\forall v_j \in V_w\colon \quad -\sigma_j \leq -\mathrm{ireq}'(v_j). \tag{5.3}$$

Here and in the following, the auxiliary function $\mathrm{int}'\colon V_s \to \mathbb{Z}$ maps each susceptible system element $v \in V_s$ to $\pi(\mathrm{int}(v))$, i.e., to the unique integer associated with $\mathrm{int}(v)$. Analogously, $\mathrm{ireq}'\colon V_w \to \mathbb{Z}$ maps each environment writer $v \in V_w$ to $\pi(\mathrm{ireq}(v))$, i.e., to the unique integer associated with $\mathrm{ireq}(v)$. For a given integrity assignment, the returned integers are constants.

Mapping (part 1/2)	Mapping (part 2/2)
v_1 = platform(main)	v_{11} = swc(body)
v_2 = proc(main.a53)	v_{12} = port(infotainment.cmd)
v_3 = proc(main.r5)	v_{13} = logic(infotainment)
v_4 = mem(main.ddr)	v_{14} = port(infotainment.viz)
v_5 = dev(main.gem)	v_{15} = port(drivetrain.meas)
v_6 = dev(main.can)	v_{16} = logic(drivetrain)
v_7 = rte(linux)	v_{17} = port(drivetrain.speed)
v_8 = rte(zephyr)	v_{18} = port(body.speed)
v_9 = swc(infotainment)	v_{19} = logic(body)
v_{10} = swc(drivetrain)	v_{20} = port(body.msg)

Table 5.4: Association of vertex indices for the car server example.

In general, these constraints lead to $m = |E| + |V_s| + |V_w|$ linear inequalities, and each inequality depends on the integrity level assigned to one or two system elements as its only variable(s).

The feasibility of the system with m linear inequalities can be determined by checking if the following ILP problem has a solution:

$$\begin{aligned} \underset{\boldsymbol{\sigma} \in \mathbb{Z}^n}{\text{maximize}} \quad & 0 \\ \text{subject to} \quad & A\boldsymbol{\sigma} \leq \boldsymbol{d}, \\ & \boldsymbol{\sigma} \geq \mathbf{0}, \end{aligned}$$

where $A \in \{-1, 0, 1\}^{m \times n}$ is a matrix and $\boldsymbol{d} \in \mathbb{Z}^m$ is a vector, both chosen according to Equation 5.1, Equation 5.2, and Equation 5.3.

▹ Example 5.5 (continued): *To perform the safety assessment of the partitioned car server use case via ILP, we first give each system element a numeric index. These indices are arbitrarily chosen and visualized in Table 5.4.*

Elements of $\boldsymbol{\sigma}$ are interpreted according to this mapping. This means that σ_1 refers to the integrity level associated with `platform(main)`*, σ_2 refers to the integrity level associated with* `proc(main.a53)`*, and so on.*

Since there is a CF graph edge from `platform(main)` *to* `proc(main.a53)`*, for example, the constraints from above include the following inequality:*

$$\sigma_2 - \sigma_1 \leq 0.$$

Intuitively speaking, it captures that the integrity level assigned to v_2 cannot be higher than the one assigned to v_1. The inherent integrity specification

of $\mathrm{int}(v_1) = \texttt{high}$ *is translated into*

$$\sigma_1 \leq 1,$$

which describes that the integrity level assigned to v_1 *cannot exceed* `high`*. The specification that* $\mathrm{ireq}(v_{20}) = \texttt{high}$ *is translated into the following inequality:*

$$-\sigma_{20} \leq -1.$$

It means that the integrity level assigned to the safety-critical body message output needs to be at least `high`*. Following this strategy, a total of*

$$m = |E| + |V_s| + |V_w| = 30 + 16 + 2 = 48$$

inequalities are specified and lead to the following matrix A *and vector* $\boldsymbol{d}$*:*

$$A = \begin{array}{c} \begin{matrix} \sigma_1 & \sigma_2 & \sigma_3 & \sigma_4 & \sigma_5 & \sigma_6 & \sigma_7 & \sigma_8 & \cdots & \sigma_{20} \end{matrix} \\ \begin{pmatrix} -1 & +1 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ -1 & 0 & +1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ -1 & 0 & 0 & +1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ -1 & 0 & 0 & 0 & +1 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ +1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & +1 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & -1 \end{pmatrix} \end{array}, \quad \boldsymbol{d} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ +1 \\ 0 \\ \vdots \\ -1 \end{pmatrix}.$$

Using the lp_solve software, it can be shown that this system of inequalities has a solution. This means that the constraints can be satisfied and, therefore, that the safety assessment has a positive result. It confirms the positive result that was already obtained from the application of Algorithm 5.2.

From a mathematical point of view, A matrices have an interesting property: it is possible to show that an A derived from the above constraints is totally unimodular (cf. Section A.1 in the appendix). Furthermore, entries of $\boldsymbol{d}$ will always be integers. Based on Section 2.2.5, this is sufficient to argue that the following LP problem has only integral solutions:

$$\begin{aligned} \underset{\boldsymbol{\sigma} \in \mathbb{R}^n}{\text{maximize}} \quad & 0 \\ \text{subject to} \quad & A\boldsymbol{\sigma} \leq \boldsymbol{d}, \\ & \boldsymbol{\sigma} \geq \mathbf{0}. \end{aligned}$$

In other words: instead of solving the ILP formulation for $\boldsymbol{\sigma} \in \mathbb{Z}^n$, it is sufficient to solve the LP formulation for $\boldsymbol{\sigma} \in \mathbb{R}^n$. This turns the problem into one with a polynomial worst-case time complexity.

5.4.2 ILP-based search and optimization framework

By extending the ILP formulation of the decision problem, it is possible to solve a search problem such as the following:

> Given a system model and an integrity assignment, find a combination of isolation measures that leads to the acceptance of the design, or determine that this is infeasible.

It is also possible to solve the corresponding optimization problem:

> Given a system model and an integrity assignment, find the best combination of isolation measures that leads to the acceptance of the design, or determine that this is infeasible.

Without loss of generality, this section presents an ILP approach to solve the optimization version under the following two assumptions:

1) A given subset of possible isolation measures is actually applicable.
2) The best solution uses a minimum number of isolation measures.

The approach is based on a vector $\boldsymbol{\omega} \in \{0, 1\}^q$ in which every entry represents an applicable isolation measure. Whether an isolation measure is applicable is decided by the designer using this approach. For example, the designer might consider it feasible to apply APU configurations to each execution platform but infeasible to specify barrier declarations. In this case, each entry of $\boldsymbol{\omega}$ would correspond to exactly one $x \in X$, and $q = |X|$.

The goal of the ILP approach is to find a specific $\boldsymbol{\omega}$. If a particular $\omega_j = 1$, then the isolation measure represented by entry j is applied. If $\omega_j = 0$, the isolation measure represented by entry j is not applied.

The procedure to find this $\boldsymbol{\omega}$ operates on a CF graph created *without* the application of these q isolation measures. As before, based on this CF graph, a system of linear inequalities is constructed. Instead of Equation 5.1 from above, however, the following rule is used:

$$\forall \langle v_j, v_k \rangle \in E\colon \quad \begin{cases} \sigma_k - \sigma_j - \beta \cdot \omega_{(f_{jk})} & \leq 0, \quad f_{jk} > 0, \\ \sigma_k - \sigma_j & \leq 0, \quad \text{otherwise.} \end{cases} \tag{5.4}$$

Here, $\beta \geq |I| - 1$ is a constant that holds at least the maximum difference between two numeric integrity level values. Subtracting it from $\sigma_k - \sigma_j$ ensures that the inequality is always met. Intuitively speaking, it deactivates the edge-induced constraint under a certain condition: the application of a particular isolation measure. Which isolation measure leads to this 'deactivation' is determined by $F \in \{0, 1, \ldots, q\}^{n \times n}$, a matrix that is yet to be determined. For all $j, k = 1, \ldots, n$, matrix element f_{jk} needs to be chosen as follows:

$$f_{jk} = \begin{cases} \alpha, & \text{applicable isolation measure } \alpha \text{ eliminates } v_j \rightsquigarrow v_k, \\ 0, & \text{no applicable isolation measure eliminates } v_j \rightsquigarrow v_k, \end{cases}$$

where $\alpha \in \{1, \ldots, q\}$ is the (numeric) index of the $\boldsymbol{\omega}$ entry that represents the respective isolation measure. The other two constraint rules from the decision problem (Equation 5.2 and Equation 5.3) remain valid:

$$\forall v_j \in V_s \colon \quad \sigma_j \leq \mathrm{int}'(v_j), \tag{5.5}$$

$$\forall v_j \in V_w \colon -\sigma_j \leq -\mathrm{ireq}'(v_j). \tag{5.6}$$

With the goal to minimize the number of applied isolation measures, i.e., the number of ones in $\boldsymbol{\omega}$, this leads to the following ILP formulation:

$$\begin{aligned} \underset{\boldsymbol{\sigma} \in \mathbb{Z}^n,\, \boldsymbol{\omega} \in \mathbb{Z}^q}{\text{maximize}} \quad & -\mathbf{1}^T \boldsymbol{\omega} \\ \text{subject to} \quad & A' \begin{pmatrix} \boldsymbol{\sigma} \\ \boldsymbol{\omega} \end{pmatrix} \leq \boldsymbol{d}', \\ & \mathbf{0} \leq \boldsymbol{\omega} \leq \mathbf{1}, \\ & \boldsymbol{\sigma} \geq \mathbf{0}, \end{aligned}$$

where matrix $A' \in \{-1, 0, 1\}^{m \times (n+q)}$ and vector $\boldsymbol{d}' \in \mathbb{Z}^m$ are chosen according to Equation 5.4, Equation 5.5, and Equation 5.6.

▹ Remark 5.5: *A' from this extended formulation does not have to be totally unimodular. For an integrity lattice with $|I| = 3$, for example, β must be set to at least 2. A totally unimodular matrix can only contain entries from $\{-1, 0, 1\}$, which might be violated in this case. A reduction to an LP problem, as in Section 5.4.1, can no longer be argued.*

▹ Example 5.5 (continued): *To finalize this example, we apply the ILP approach to minimize $|\Omega_X| + |\Omega_R|$ in such a way that the design is still regarded as safe. At the hardware layer, there is only the possibility to apply APU configuration*

code to `platform(main)`. *At the runtime layer, process isolation can be declared for* `rte(linux)` *and* `rte(zephyr)`. *We introduce* ω_1, ω_2, *and* ω_3 *to refer to these applicable isolation measures in the given order. Based on this assignment, the matrix* F *has non-zero entries at exactly six positions:*

$$
\begin{array}{c}
 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \\ v_8 \\ v_9 \\ v_{10} \\ v_{11} \\ \vdots
\end{array}
\begin{array}{c}
\begin{array}{cccccccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 & v_9 & \cdots \end{array} \\
\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & \cdots & & & \\
1 & 0 & 0 & 0 & 0 & \cdots & & & & \\
1 & 0 & 0 & 0 & \cdots & & & & & \\
0 & 0 & 0 & \cdots & & & & & & \\
1 & 0 & \cdots & & & & & & & \\
0 & \cdots & & & & & & & & \\
\cdots & & & & & & & & & \\
 & & & & & & & & & \\
 & & & & & & 2 & & & \\
 & & & & & & & 3 & & \\
 & & & & & & & 3 & & \\
 & & & & & & & & &
\end{pmatrix}
\end{array}
$$

With $\beta = 1$, *this leads to the following* A' *and* $\boldsymbol{d}'$ *values:*

$$
A' = \begin{array}{c}
\begin{array}{cccccccccc} \sigma_1 & \sigma_2 & \sigma_3 & \sigma_4 & \sigma_5 & \cdots & \sigma_{20} & \omega_1 & \omega_2 & \omega_3 \end{array} \\
\begin{pmatrix}
-1 & +1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\
+1 & -1 & 0 & 0 & 0 & \cdots & 0 & -1 & 0 & 0 \\
+1 & 0 & -1 & 0 & 0 & \cdots & 0 & -1 & 0 & 0 \\
+1 & 0 & 0 & 0 & -1 & \cdots & 0 & -1 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\
+1 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\
0 & +1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & 0 & 0 & \cdots & -1 & 0 & 0 & 0
\end{pmatrix}
\end{array}
, \quad \boldsymbol{d}' = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 0 \\ 0 \\ \vdots \\ +1 \\ 0 \\ \vdots \\ -1 \end{pmatrix}
$$

For this ILP problem, lp_solve returns an optimum solution of $\boldsymbol{\omega} = (1, 0, 0)$. *With this solution, the cost function evaluates to* -1. *This means that:*

1) *To meet the safety requirements from Table 5.3, it is sufficient to apply APU configuration code to* `platform(main)`. *Process isolation is not required for either* `rte(linux)` *or* `rte(zephyr)`.

2) *The minimum number of isolation measures required to meet the safety requirements from Table 5.3 is one.*

This concludes the description of the optimization procedure. Note that the strategy of 'selectively disabling' the effect of CF graph edges, which is how the ILP approach considers isolation measures, is not limited to the problem statement from above. It is a flexible approach applicable to various related problems. One such problem is the safety-aware allocation of SWCs to RTEs. Its solution goes beyond the scope of this thesis, however, and remains a possible direction for future research.

Chapter 6

Implementation and evaluation

As part of this thesis, the logical isolation pattern was fully implemented. The resulting implementation is a seamless integration of the described APU configuration, CF determination, and safety assessment procedures. It is available in the form of a command-line utility and expects pattern users to formulate inputs using a textual notation. The notation is a Domain-Specific Language (DSL) specifically developed for the logical isolation pattern.[1]

To evaluate the usability of the pattern, its implementation was then used for (1) the safety-aware deployment of an Ethernet-based software application to the Zynq UltraScale+ MPSoC and (2) the safety assessment of a cost-efficient fault tolerance approach.

6.1 Language specification

In the remainder of this thesis, the developed DSL will be referred to as Logical Isolation Pattern (LIP) syntax. Its context-free grammar is now specified using Wirth syntax notation [149]—with one useful extension: in a sequence of alternatives, the '…' symbol is used to represent omitted terms.[2] This means that literals are surrounded by double quotes, alternatives are separated using |, groups are wrapped into a () pair, repetition is denoted by a {} pair, and optional expressions are surrounded by a [] pair.

[1] As described in Chapter 7, the author contributed this notation to the XANDAR project. Therefore, it is also aligned with the XbCgen syntax accepted by the XANDAR toolchain.

[2] This approach is also used in the Go language specification [150], for example.

6.1.1 Lexical elements and top-level grammar

The lexical structure of LIP syntax is based on five token types: keyword, punctuation, identifier, identifier path, and hexadecimal number.

Keyword and punctuation tokens are the literals that appear in any production rule from Section 6.1.2 or Section 6.1.3. If such a literal contains at least one alphabetical character, such as `platform` or `gen_apu`, it is a keyword. If it contains no such character, such as '`>=`', it is a punctuation token. The remaining three token types are defined as follows:

```
      ident = letter {letter | decimal_digit} .
 ident_path = ident {"." ident} .
hex_number = "0x" hex_digit {hex_digit} .
```

The underlying non-terminals are defined as follows:

```
decimal_digit = "0" ... "9" .
    hex_digit = "0" ... "9" | "a" ... "f" | "A" ... "F" .
       letter = "a" ... "z" | "_" .
```

Based on this, an example of an identifier (`ident`) is `slcr_fpd`. An example of an identifier path (`ident_path`) is `main.a53`. An example of a hexadecimal number (`hex_number`) is `0x1000`.

Line comments are introduced by two forward slashes (`//`); they instruct the lexer to ignore the remainder of a respective line. Furthermore, the lexer ignores any kind of whitespace between tokens.

The start symbol, which describes a complete LIP syntax file, is an arbitrary sequence of items drawn from seven alternative constructs:

```
file = {platform | bus | rte | path | swc | channel | pattern} .
```

The first six constructs (from platform to channel) describe a system model. The final construct allows the user to 'annotate' the pattern itself and, by doing so, describe isolation measures and safety requirements.

6.1.2 Grammar for system model entities

The foundation of a system model is the system's hardware architecture (cf. Section 3.2.2). At this layer, the LIP syntax allows users to instantiate execution platforms, specify the device configuration, and connect I/O controllers via

off-chip interconnects. The device configuration is based on the following rules, which make it possible to express a set of memory regions:

```
mem_region  = hex_number "-" hex_number .
mem_regions = mem_region {"," mem_region} .
```

For each memory region, the start and the end address is separated by a hyphen. Based on this, the hardware architecture syntax is as follows:

```
  platform = "platform" ident ":" ident (";" | "{" {dev_entry} "}") .
 dev_entry = "alloc" "(" mem_regions ")" "to" ident ";" .
       bus = "bus" ident "=" ident_path {"," ident_path} ";" .
```

The first rule instantiates an execution platform ($x \in X$) with a type from the underlying library ($t \in T$). The identifier following the `platform` keyword is the *name* of this execution platform, while the identifier following the ':' is a reference to the *type* of the platform. Afterward, it is possible to add device configuration entries for the instantiated platform. Each such entry is a list of memory regions allocated to a peripheral device, and this peripheral device is specified by the identifier following the `to` token. The third rule introduces an off-chip interconnect ($b \in B$). The identifier following the `bus` keyword becomes the name of this interconnect, and the comma-separated list of identifier paths reference all I/O controllers that are attached to this interconnect (formally the 'controllers' function from Definition 3.12).

▹ Example 6.1: *The partitioned car server use case from Example 3.6 on page 58 makes use of a Zynq UltraScale+ MPSoC instance. This instance was created from the simplified execution platform type from Example 3.5. Assuming that this execution platform type is named* `zynqmp_demo`*, we can use the following LIP code to describe the hardware architecture of the use case:*

Listing 6.1: Hardware architecture of the partitioned car server

```
platform main : zynqmp_demo {
  alloc (0x3C00000 - 0x43FFFFF) to gem;
}
```

This syntax allocates a DDR memory region of the underlying MPSoC to the Ethernet controller (`gem`*). Off-chip interconnects are not declared in this case.*

▹ Example 6.2: *To connect two* `zynqmp_demo` *instances via a CAN network, the following hardware architecture can be used:*

Listing 6.2: Zynq UltraScale+ MPSoC instances connected via CAN

```
platform primary : zynqmp_demo;
platform secondary : zynqmp_demo;
bus network = primary.can, secondary.can;
```

For the definition of an RTE ($r \in R$), including its mapping to a processing unit and its allocations, LIP syntax accepts the following statements:

```
        rte = "rte" ident ":" ident_path "{" {rte_allocs} "}" .
rte_allocs = "alloc" rte_alloc {"," rte_alloc} ";" .
 rte_alloc = mem_region | ident .
```

The identifier following the `rte` keyword is the name of the RTE. The identifier path following the ':' is the path of the processing unit executing this RTE. Surrounded by curly braces, an arbitrary sequence of allocation statements can be specified. Each statement contains a comma-separated list of allocated memory regions and peripheral devices. In the formal system model, they correspond to malloc′ and qalloc′, respectively.

▹ Example 6.1 (continued): *To continue the LIP formulation of the partitioned car server use case, we introduce two RTEs:* `linux` *mapped to the Cortex-A53 processor and* `zephyr` *mapped to the Cortex-R5 processor. Allocations are specified as it was described in Example 3.6:*

Listing 6.3: Runtime architecture of the partitioned car server

```
rte linux : main.a53 {
  alloc 0x0 - 0x17FFFFF;
}

rte zephyr : main.r5 {
  alloc 0x2C00000 - 0x37FFFFF, can;
}
```

Local and global paths ($y \in L \cup G$) are specified by referencing their RTEs in curly braces and describing their realization:

```
path = "path" "{" ident "," ident "}" "=" (mem_regions | ident) ";" .
```

In case of a local path, the realization is a list of memory regions (specified using the 'mem_regions' rule). In case of a global path, it is a reference to the name of an off-chip interconnect (specified using the 'ident' rule).

The declaration of a SWC ($s \in S$) contains its name, its executing RTE, and a sequence of port ($p \in P$) and allocation statements:

```
      swc = "swc" ident ":" ident "{" {swc_ports | swc_allocs} "}" .
swc_ports = swc_port {"," swc_port} ";" .
swc_allocs = "alloc" ident {"," ident} ";" .
 swc_port = ("out" | "in") ["~"] ident .
```

The name of a SWC follows the `swc` keyword, while the RTE reference follows the ':' punctuation. Multiple port declarations can be merged into a comma-separated list. Each element of this list uses a keyword to declare the port direction (`out` or `in`), uses an (optional) tilde to flag environment ports, and ends with the name of the port. In line with Definition 3.18, a SWC may allocate only peripheral devices. These allocations are expressed in the same way as for RTEs, but they populate qalloc'' instead of qalloc'.

▹ Example 6.1 (continued): *To describe the SWC implementing infotainment functions, the following LIP syntax can be used:*

Listing 6.4: SWC specification from the partitioned car server

```
swc infotainment : linux {
  in ~cmd, out ~viz;
  alloc gem;
}
```

Channels are specified by explicitly stating the source and the sink port they connect, each using an identifier path:

```
channel = "channel" ident_path "->" ident_path ";" .
```

▹ Example 6.1 (continued): *The remaining two SWCs from the partitioned car server example can be declared and connected as follows:*

Listing 6.5: Software architecture of the partitioned car server

```
swc drivetrain : zephyr {
  in ~meas, out speed;
}

swc body : zephyr {
  in speed, out ~msg;
}

channel drivetrain.speed -> body.speed;
```

In combination with the LIP syntax in previous parts of this example, this concludes the description of the system model from Example 3.6.

6.1.3 Grammar for pattern-specific annotations

In comparison to the system model, isolation measures and safety requirements are less universal. They are specific to the logical isolation pattern and therefore part of a designated 'pattern' annotation:

```
pattern = "pattern" [ident] ":" "isolation"
            (";" | "{" [int_lattice] {isolation_item} "}") .
```

Following the `pattern` keyword, this rule begins with an optional identifier, which can be used to assign a name to the pattern annotation. In curly braces, it is then possible to specify the integrity lattice as a collection of integrity level sequences. Collectively, all sequences specify a dominance relation between integrity levels, each given using an identifier:

```
    int_lattice = "lattice" "{" {level_sequence} "}" .
level_sequence = ident {">=" ident} ";" .
```

Referenced identifiers implicitly define I. The partial order $\leq$ (according to Definition 5.4) is derived from the specified dominance relation. Therefore, the order of integrity levels listed in each level sequence is inverted, and a $\leq$ entry is derived from each specified level to the next. Finally, the reflexive transitive closure of all these entries yields $\leq$, i.e., the partial order.

▹ Example 6.3: *Consider the set* $I = \{\texttt{low}, \texttt{mid}, \texttt{high}\}$ *with the following partial order:* $\texttt{low} \leq \texttt{low}$, $\texttt{mid} \leq \texttt{mid}$, $\texttt{high} \leq \texttt{high}$, $\texttt{low} \leq \texttt{mid}$, $\texttt{mid} \leq \texttt{high}$, *and* $\texttt{low} \leq \texttt{high}$. *Two ways to describe this in LIP syntax are given below:*

Listing 6.6: Semantically equivalent integrity lattices

```
lattice {
  high >= mid;
  mid >= low;
}

lattice {
  high >= mid >= low;
}
```

Logic decomposition, APU configuration requests, barrier declarations, and the assignment of 'int' and 'ireq' values can all be specified between the optional integrity lattice and the closing brace:

```
isolation_item  = context_isolation | logic_block | int_assignment .
context_isolation = ("gen_apu" | "prot_rte!") ident ["," ident] ";" .
logic_block     = "logic" "(" ident_path ")" ["=" ident]
                    (";" | "{" {app_barrier} "}") .
app_barrier     = ("no_in!" | "no_out!") ident ["," ident] ";" .
int_assignment  = int_obj "(" ident_path ")" ("=" | ">=") ident ";" .
int_obj         = ("swc" | "port" | "platform" | "bus" | "rte" |
                   "proc" | "mem" | "dev") .
```

Isolation measures targeting context vertices are specified using the 'context_isolation' syntax: APU configuration is requested via the `gen_apu` keyword, while process isolation is declared via the `prot_rte!` keyword.

Using the 'logic_block' syntax, it is possible to (1) perform logic decomposition, (2) assign the inherent integrity of each logic block, and (3) specify application-level barriers of a logic block. The identifier path following a `logic` statement is either the name of a SWC or, alternatively, the name of a SWC suffixed with a dot and the name of the partial logic block to create. Within curly braces, the `no_in!` keyword can be used to declare an input barrier, while the `no_out!` keyword introduces an output barrier. Optionally, following the '`=`' token, the inherent integrity of a logic block is specified. If unspecified, it defaults to the minimum integrity level from $\langle I, \leq \rangle$.

Using the 'int_assignment' syntax, it is possible to specify the inherent integrity of all other susceptible system elements. This is achieved using the '=' variant of this rule. At the same time, the '>=' variant of the rule can be used to specify required integrities of environment writers. As before, all unspecified integrities default to the minimum integrity level from $\langle I, \leq \rangle$.

▹ Example 6.1 (continued): *To conclude the LIP formulation of the partitioned car server example, we use the textual syntax to apply the logical isolation pattern:*

Listing 6.7: Pattern applied to the partitioned car server example

```
pattern : isolation {
  lattice { high >= low; }
  prot_rte! zephyr;
  gen_apu main;

  platform(main) = high;
  proc(main.r5) = high;
  mem(main.ddr) = high;
  dev(main.can) = high;
  rte(zephyr) = high;
  swc(body) = high;
  swc(drivetrain) = high;
  port(drivetrain.meas) = high;
  logic(drivetrain) = high;
  logic(body) = high;
  port(body.msg) >= high;
}
```

▹ Example 6.4: *In an extended version of the car server use case (cf. Example 4.5), the body control SWC was extended with a* `seatpos` *input, and an application-level barrier between this input and the body control logic was introduced. Using the DSL, this barrier is represented as follows:*

Listing 6.8: Selected excerpt of the extended car server model

```
logic(body) = high {
  no_in! seatpos;
}
```

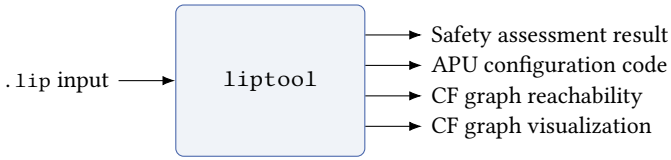


Figure 6.1: Command-line tool implementing the logical isolation pattern. In addition to APU configuration and integrity-based safety assessment, it provides functions for the automatic analysis and visualization of CF graphs.

6.2 Reference implementation

The presented implementation is referred to as 'LIP tool' and was developed using version 1.22 of the Go programming language [150]. In the following, calls to this utility are indicated by invocations of the `liptool` binary. As shown in Figure 6.1, it operates on one single LIP syntax file, which by convention has a `.lip` extension. The tool first parses and resolves the provided input. It then executes one or more of the following tasks:

1) Conduct an integrity-based safety assessment (cf. Section 5.3) and return the assessment result as a binary success value.
2) Generate and output APU configuration code for each $x \in \Omega_X$ (or determine and return that this is not possible for some reason).
3) Using the internal CF graph, determine and output all environment writers that are reachable from a given $v \in V_s$.
4) Output a Ti*k*Z visualization of the internal CF graph, potentially with annotations such as effective integrity levels.

▹ Remark 6.1: *A safety assessment based on interference whitelists (cf. Section 5.2) is not directly supported. Based on Theorem 5.1, however, it can be achieved by transforming the interference whitelist into an integrity assignment.*

The logical isolation pattern, as it was introduced in Section 3.1, needs to perform the safety assessment and the generation of APU configuration code. This is achieved by the features in item 1 and item 2 above. The other features are useful extensions to give pattern users more insights into the procedure. The feature in item 4, for example, was used to generate all the CF graph visualizations shown in this thesis.

Option	Description
`-verify`	Perform a safety assessment according to Section 5.3.
`-gen` ⟨`dir`⟩	Run code generation and write output(s) to ⟨`dir`⟩.
`-sinks` ⟨`source`⟩	Determine environment writers reachable from ⟨`source`⟩.
`-visualize` ⟨`dir`⟩	Write a Ti*k*Z visualization of CF graphs to ⟨`dir`⟩.
`-transfers`	Enforce the visualization of all CF potential transfers.
`-rules`	Annotate edges of CF graphs with rule identifiers.

Table 6.1: Options accepted by the implemented command-line tool. Terms surrounded by angular brackets are parameters processed by the application.

6.2.1 Command-line interface

As its (only) mandatory argument, `liptool` expects a path to the `.lip` file to process. The specific tasks to perform using this input are determined by the options in Table 6.1. These options can be arbitrarily combined.

For example, a call to '`liptool -gen configs -verify model.lip`' writes APU configuration code to '`configs`' and returns the binary acceptance value. Calling '`liptool -visualize graphs model.lip`' generates visualizations such as the ones in Figure 4.7 or Figure 5.1. Extending this call with '`-transfers`' and '`-rules`' leads to a visualization such as the one in Figure 4.23. If the '`-verify`' option is combined with '`-visualize`', the generated Ti*k*Z output annotates CF graph edges with effective integrities and environment writers with their safety conformance; an example of such a visualization was shown in Figure 5.7.

6.2.2 Execution platform types

To show that the pattern concept is applicable to various MPSoCs, execution platform types for two platforms were developed: the Zynq UltraScale+ MPSoC and the i.MX 8M (cf. Example 2.4). For these commercially available platforms, the library was populated with a total of three entries:

1) `zynqmp_demo` is a simplified model of the Zynq UltraScale+ MPSoC; it considers only a small subset of available on-chip components.
2) `zynqmp` is the full version of the Zynq UltraScale+ MPSoC model; it considers all on-chip components required for the evaluation.
3) `imx8m` is a proof-of-concept model of the i.MX 8M; it was used to show that RDC configurations can be generated successfully.

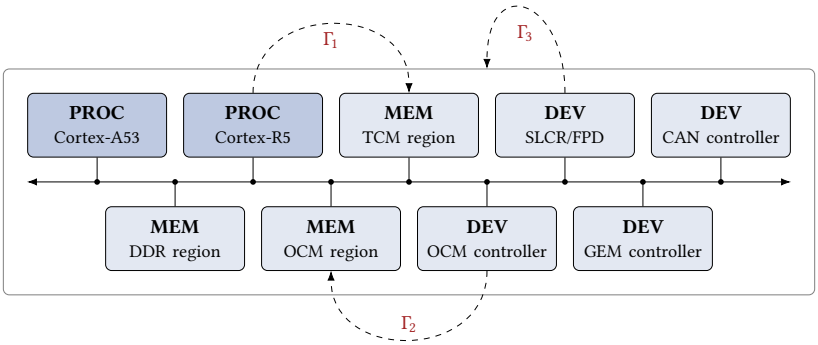


Figure 6.2: Simplified execution platform type of a Zynq UltraScale+ MPSoC, formally covered in Example 3.5 and here repeated for convenience.

In all three cases, the address space origin ($\hat{a}$) is set to the Cortex-A53 processor of the respective execution platform type.

The first type, `zynqmp_demo`, realizes the simplified model that was derived in Example 3.5 and is repeated in Figure 6.2 for convenience. The implementation of this execution platform type is the following Go code:

Listing 6.9: Go formulation of Example 3.5

```
ddr := Memory{Name: "ddr", Start: 0x00000000, End: 0x7FFFFFFF}
ocm := Memory{Name: "ocm", Start: 0xFFFC0000, End: 0xFFFFFFFF}
tcm := Memory{Name: "tcm", Start: 0xFFE00000, End: 0xFFE1FFFF}
r5 := Processor{Name: "r5", AccessibleMems: []*Memory{&tcm}}
a53 := Processor{Name: "a53"}
library["zynqmp_demo"] = PlatformType{
  Processors: []*Processor{&a53, &r5},
  Memories:   []*Memory{&ddr, &tcm, &ocm},
  Devices: []*Device{
    {Name: "ocm_ctrl", AffectedMems: []*Memory{&ocm}},
    {Name: "slcr_fpd", HasPlatformImpact: true},
    {Name: "gem", IsMaster: true},
    {Name: "can"},
  },
}
```

This model was created for visualization purposes and is too limited for a practical application. It disregards the fact that there are multiple instances of the GEM and CAN controller, for instance. The second type, `zynqmp`, is intended for a practical application and therefore more accurate.

6.2.3 Input model resolution

After parsing a `.lip` file, the implementation resolves all references made by identifiers or identifier paths. As part of this, the input file is validated and possible errors are reported back to the user. This process is independent of the command-line options passed to the utility.

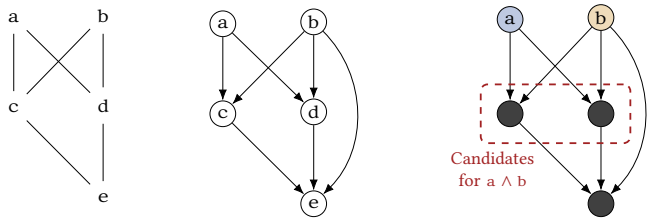
The resolution process is now illustrated using the `lattice` statement, which defines a dominance relation that needs to be translated into $\leq$. This translation is achieved by attempting to calculate the meet of all integrity level pairs ($i_1 \wedge i_2$ for all $i_1, i_2 \in I$). If two integrity levels do not have a meet, $\langle I, \leq \rangle$ is not a meet-semilattice, and the `.lip` file is invalid.

Algorithm 6.1: Meet calculation for integrity levels

```
1  function CalculateMeetTable()
2      // Iterate over all (unordered) pairs from I = {i_1, i_2, ...}:
3      for each j = 1, ..., |I| do
4          for each k = 1, ..., j do
5              // Get black vertices not reachable from another black node:
6              candidates = MeetCandidates(i_j, i_k)
7              // If this vertex is unique, add it to the meet table:
8              if |candidates| ≠ 1 then
9                  return error (i_j ∧ i_k does not exit)
10             meet ← (the integrity level in candidates)
11             meetTable[i_j][i_k] = meet
12             meetTable[i_k][i_j] = meet
13     return meetTable
```

For all $j = 1, \ldots, |I|$ and $k = 1, \ldots, j$, it calls MeetCandidates(i_j, i_k) to obtain the meet value candidates of i_j and i_k. Following one of the approaches from [151], these candidates are determined as follows:

1) Represent the dominance relation (`>=`) as a directed graph.
2) If this graph contains cycles, return the empty set.
3) Color vertices reachable from both i_j and i_k black.
4) Return all black vertices not reachable from another black node.



(a) Hasse diagram (b) Dominance relation (c) MeetCandidates(a, b)

Figure 6.3: Detection of an invalid dominance relation (supposed to define ≤). For each combination of integrity levels, MeetCandidates colors vertices reachable from both levels black. A subset of colored vertices is the set of meet candidates.

If there is exactly one such candidate, it is the meet value. Otherwise, there is no greatest lower bound, which means that the described structure is not an integrity lattice; in this case, the input is rejected.

▹ Example 6.5: *To represent the partial order that is visualized in Figure 6.3a, the following LIP formulation can be used:*

Listing 6.10: Semantically invalid system model

```
// invalid.lip
pattern : isolation {
  lattice {
    a >= c >= e;
    b >= d >= e;
    a >= d;
    b >= c;
    b >= e;
  }
}
```

Figure 6.3b illustrates this dominance relation in the form of a directed graph. During the model resolution procedure, `MeetCandidates` *traverses this graph to calculate the meet of all integrity level pairs. As shown in Figure 6.3c, there are two candidates for* a ∧ b. *This means that none of them is the greatest one and, therefore, the described structure is not a meet-semilattice.*

When this input model is supplied to the `liptool` *utility, regardless of specified command-line options, the following error is reported:*

Listing 6.11: Error reporting for an invalid integrity lattice

```
$ liptool invalid.lip
error: invalid.lip:3: ambiguous meet of a and b
```

6.2.4 APU configuration procedures

An execution platform library according to Definition 3.9 consists of two components: T (the set of execution platform types) and gen (a collection of APU configuration procedures for selected $t \in T$). The reference implementation provides APU configuration procedures for two of the three execution platform types: `zynqmp` and `imx8m`. In line with Section 4.2.2, each of them provides the platform-independent root algorithm with four capabilities:

- When triggered by NewBaseConfig, it creates a mutable configuration object that isolates all instantiated on-chip components from each other. As described in Section 4.2.2.1, if applicable, the initial state of this object may be configured to accept a certain set of transactions.
- It extends this configuration object when a call to GrantMemAccess or to GrantDevAccess makes it necessary to do so.
- Finally, when triggered by WriteCode, it generates C code that applies the final state of this configuration object to one or more APUs of the targeted execution platform.

Like the platform-independent root algorithm itself, both implementations were developed in version 1.22 of the Go programming language.

For the Zynq UltraScale+ MPSoC model (`zynqmp`), the implementation makes use of the exact algorithms described in Section 4.2.2.2. The initial state of its configuration object permits the following transactions:

1) Debug Access Port (DAP) → all possible slave components
2) Configuration Security Unit (CSU) → Platform Management Unit (PMU)
3) Cortex-A53 → Generic Interrupt Controller (GIC)
4) Cortex-A53 → Global system counter

Granting these access permissions by default served useful during the case studies (cf. Section 6.3). Allowing the DAP to access all possible slave components facilitated debugging tasks, for instance. At the same time, it does not

violate requirements of the root algorithm: DAP is not modeled as a platform component in the sense of Definition 3.8 and, therefore, does not need to be isolated. The two slave components made accessible from the Cortex-A53 are highly related to this CPU; since they are not modeled as instantiated on-chip components, these default permissions are again possible.

▹ Remark 6.2: *The four default permissions specified for the* `zynqmp` *type should be understood as an example for how the initial state of configuration objects can be chosen. Depending on the particular use case, it is possible to argue that it should contain more, less, or even no default permissions.*

It is also conceivable to populate the execution platform library with multiple execution platforms types for the same MPSoC—each with its own set of default permissions and therefore tailored to a particular way of using this MPSoC.

Default permissions are merged with those required by the respective system model. In the `zynqmp` case, this leads to C code that configures all XMPUs and the XPPU of a Zynq UltraScale+ MPSoC. To support this process, a static configuration library was developed; Section A.2 in the appendix describes the interface and implementation of this library in more detail.

In addition, platform-specific generators have the opportunity to output additional artifacts, such as reports that communicate more knowledge about derived APU configuration code. The `zynqmp` generator, for example, complements generated C code with such a report.

▹ Example 6.6: *The following LIP formulation contains a simple system model and a request to generate APU configuration code for the* `main` *platform:*

Listing 6.12: LIP formulation for a simple system model

```
platform main : zynqmp;
rte a53 : main.a53 {
  alloc 0x1000000 - 0x1FFFFFF, uart0;
}

swc app : a53 {
  alloc can0;
}

pattern : isolation {
  gen_apu main;
}
```

Applying the `liptool -gen` *command to this input triggers the* `zynqmp` *generator and results in the following APU configuration code:*

Listing 6.13: APU configuration code (partially truncated)

```
#include "apulib.h"

void apu_apply(void) {
  xmpu_clear_configs();
  xmpu_set_ddr_region(0, 0x0UL, 0xFFFFFFFFUL, MREG_DAP);
  xmpu_set_ocm_region(0, 0x0UL, 0xFFFFFFFFUL, MREG_DAP);
  xmpu_set_fpd_region(0, 0x0UL, 0xFFFFFFFFUL, MREG_DAP);
  xmpu_set_fpd_region(1, 0xF9000000UL, 0xF90FFFFFUL, MREG_A53);
  xmpu_set_ddr_region(1, 0x1000000UL, 0x1FFFFFFUL, MREG_A53);
  xmpu_finalize_configs();

  xppu_clear_config();
  xppu_set_master_profile(0, MREG_DAP);
  xppu_set_master_profile(1, MREG_A53);
  xppu_set_master_profile(2, MREG_CSU);
  xppu_set_permissions(XPPU_APER_IOU_SCNTR, 0x3UL);
  xppu_set_permissions(XPPU_APER_IOU_SCNTRS, 0x3UL);
  xppu_set_permissions(XPPU_APER_PMU_GLOBAL, 0x5UL);
  xppu_set_permissions(XPPU_APER_UART0, 0x3UL);
  xppu_set_permissions(XPPU_APER_UART1, 0x1UL);
  xppu_set_permissions(XPPU_APER_UART2, 0x1UL);
  xppu_set_permissions(XPPU_APER_UART3, 0x1UL);
  xppu_set_permissions(XPPU_APER_CAN0, 0x3UL);
  // ...
  xppu_finalize_config();
}
```

The code executes a sequence of function calls, each targeting a function from the library described in Section A.2. Successfully executing this code on any processor of the `main` *platform leads to the fulfillment of Premise 4.8.*

In addition to the APU configuration itself, the generator also creates the report shown in Figure 6.4. Among other aspects, this report documents the resource utilization caused by generated APU configuration code. It states, for example, that two of the 16 XMPU/DDR regions are occupied: one granting the DAP module full DDR access (which is a default permission) and one granting the Cortex-A53 processor full access to the memory region owned by its RTE.

APU configuration report for ‘`main`’

Creation time: 2024-06-02, 07:49 pm (CEST)
Code generator: `zynqmp`

Resource utilization

XPPU	XMPU/FPD	XMPU/DDR	XMPU/OCM
3/20 master profiles	2/16 regions	2/16 regions	1/16 regions

Access permissions

Master	Accessible addresses (per APU module)			
	XPPU	XMPU/FPD	XMPU/DDR	XMPU/OCM
`dap`	$(*)^{\triangle}$	$(*)^{\triangle}$	$[0x0, 0xFFFFFFFF]^{\triangle}$	$[0x0, 0xFFFFFFFF]^{\triangle}$
`a53`	`iou_scntr`$^{\triangle}$	`gic`$^{\triangle}$	$[0x1000000, 0x1FFFFFF]$	–
	`iou_scntrs`$^{\triangle}$	–	–	–
	`uart0`	–	–	–
	`can0`	–	–	–
`csu`	`pmu_global`$^{\triangle}$	–	–	–

Legend: $[x, y]$ = memory region from x to y (inclusive); $(*)$ = any address (wildcard); $\triangle$ = default permissions

Figure 6.4: APU configuration report complementing the code from Listing 6.13. Automatically created by the `zynqmp` generator as a Portable Document Format (PDF) file.

Code generated for `imx8m` platforms has a structure similar to the code shown in Listing 6.13. However, it targets the RDC of the i.MX 8M instead of XPPU and XMPU modules.

6.2.5 Automatic visualization of CF graphs

Inputs supplied to `liptool` are translated into three graphs: the full CF graph, its application subgraph, and its context subgraph. Each of them is stored in the form of an adjacency list. During their execution, tasks are able to extend these graphs with annotations. While the ‘`-verify`’ task is executed, for example, edges and vertices are labeled with integrity levels.

Using the ‘`-visualize`’ option, the final state of each graph can be written out in the form of Ti*k*Z code. This code employs the `graphdrawing` library to position and connect vertices. All graphs are generated using

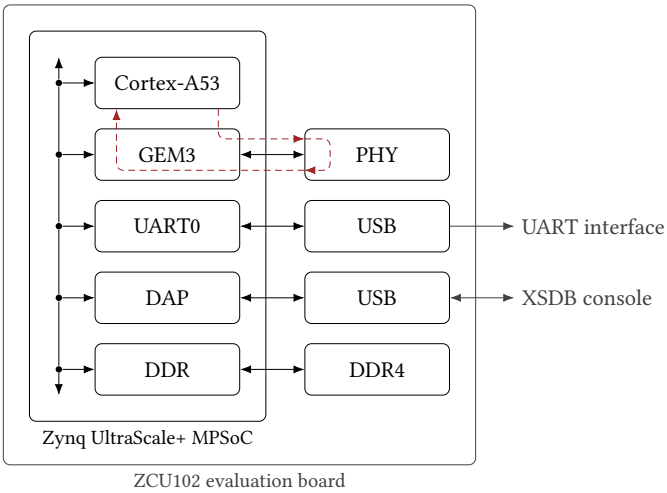


Figure 6.5: Physical hardware setup used for the first case study. Dashed arrows from the Cortex-A53 to PHY and back describe the loopback test to run.

the `layered` drawing algorithm from [152]. To achieve the characteristic bottom-up structure of CF graph visualizations, edges are processed as follows:

1) Dependency-based edges (from E_δ) are directly supplied to the algorithm.
2) Activity-based edges (from E_η) are visually reversed: if there is an antiparallel edge from E_δ, they are merged into the antiparallel one; otherwise, the algorithm is instructed to draw them into the opposite direction.

Compiling generated code using LuaTEX, optionally with a custom style sheet, leads to a vector graphic representation of each graph. Manual postprocessing of the Ti*k*Z code can be applied to increase the readability of a visualization, for example by bending a particular edge. Such postprocessing was applied to obtain the CF graphs visualized in this thesis.

6.3 Case study: Ethernet controller access

To demonstrate the applicability of the logical isolation pattern, the reference implementation was first used to generate an APU configuration for the Zynq UltraScale+ MPSoC in a particular test scenario. Figure 6.5 shows the

hardware setup of this scenario; it comprised a ZCU102 evaluation board [153] connected to a host machine via a Universal Asynchronous Receiver/Transmitter (UART) and the Xilinx System Debugger (XSDB) interface.

The primary goal of the case study was to demonstrate that using generated APU configuration code, the Cortex-A53 remained able to configure and use the GEM3 controller. Usage of this Ethernet controller requires correct interrupt handling and involves a DMA function in which the controller appears as a master of the on-chip interconnect. Demonstrating that the logical isolation pattern supports this use case is therefore representative for various other peripheral devices of the MPSoC. To achieve this, the loopback test illustrated in Figure 6.5 was executed. More specifically, an interrupt-driven demo application shipped with the `emacps` driver by AMD/Xilinx was executed on the Cortex-A53. The success status (i.e., whether the transmitted Ethernet frames were later received correctly) was reported to the host machine via UART0. The memory region used by the test application was stored in DDR memory; this region also stored the buffers that GEM3 used for DMA-driven communication with the Cortex-A53.

As a secondary goal, the case study aimed to demonstrate that CF graphs are accurate with respect to CF potential that was present in the actual hardware setup. As part of this, certain safety assessment problems were defined, evaluated by a `liptool` invocation, and finally verified by manual attempts to trigger a particular CF using XSDB access to the MPSoC.

Since there was only one SWC to be executed by the Cortex-A53, the RTE was a simple bare-metal environment directly provided by AMD/Xilinx. It comprised a bootloader that spawned one binary (the SWC) as the only executed application. Any memory region allocated to the RTE was automatically forwarded to this SWC. Additional management functions (e.g., for inter-SWC communication) or a scheduler were not present.

6.3.1 Pattern-aware memory partitioning

To exchange Ethernet frame data, GEM controllers depend on receive and transmit buffers. Along with associated descriptors, these buffers are stored in memory that needs to be (read and write) accessible from both the application that uses a GEM controller and the controller itself. The test application allocates this memory region as part of its own data storage. This kind of allocation is not directly compatible with the logical isolation pattern, however. SWC data is stored in a memory region of the underlying RTE, but according to Definition 3.18, the pattern does not permit to allocate the same region

Memory region	Description
$\mu_{\text{A53}} = [\texttt{0x0}, \texttt{0x7FFFFF}]_{\mathbb{A}}$	DDR memory region that contains (1) all code of the test application and (2) all data that is exclusive to this application. Owned by the RTE of the Cortex-A53 processor.
$\mu_{\text{GEM3}} = [\texttt{0x800000}, \texttt{0xFFFFFF}]_{\mathbb{A}}$	DDR memory region that holds (1) receive and transmit buffers as well as (2) associated buffer descriptors. Owned by GEM3.

Table 6.2: Partitioning of the initial application memory to achieve compliance with requirements of the logical isolation pattern.

also to an I/O controller at the same time. Instead, it expects the pattern user to allocate the memory region to *only* the I/O controller. Whichever system entity is granted access to this controller will then automatically obtain read and write access to the controller's memory.

The first step of the case study was therefore to partition the test application memory into two portions. The resulting memory regions are given and described in Table 6.2. From an implementation point of view, this partitioning was facilitated by a `.gem` section added to the application's linker script:

Listing 6.14: Linker script modifications for memory partitioning

```
MEMORY {
  psu_ddr_0_MEM_0 : ORIGIN = 0x0, LENGTH = 0x800000
  psu_ddr_0_MEM_1 : ORIGIN = 0x800000, LENGTH = 0x800000
}

SECTIONS {
  // .text, .init, .fini, ...
  .gem (NOLOAD) : {
    . = ALIGN(64);
    *(.gem)
    . = ALIGN(64);
  } > psu_ddr_0_MEM_1
}
```

In this listing, modifications are highlighted in orange, while the majority of unchanged lines (`.text`, `.init`, ...) is hidden.

Step	Master	Memory region		Peripheral device			
		μ_{A53}	μ_{GEM3}	gem3	uart0	csu	crl
Initialization	Cortex-A53	●	●	●	○	●	●
	GEM3	○	○	○	○	○	○
Loopback test	Cortex-A53	●	●	●	●	○	○
	GEM3	○	●	○	○	○	○

Table 6.3: Test application steps, each associated with an overview of memory regions and peripheral devices accessed during this step; the symbol '●' indicates that a given master performs at least one read or write access targeted at the given slave.

Afterward, the compiler was instructed to move receive buffers, transmit buffers, and buffer descriptors into the new `.gem` section:

Listing 6.15: Source code modifications for memory partitioning

```
// Compiler attribute to declare GEM-owned memory:
#define GEM __attribute__((section(".gem")))
// Buffer descriptors (in uncached memory):
u8 BdRegion[0x200000] __attribute__((aligned (0x200000))) GEM;
// Transmit buffer:
EthernetFrame TxFrame GEM;
// Receive buffer:
EthernetFrame RxFrame GEM;
```

With this, memory regions used by both masters were in line with the abstraction that the logical isolation pattern expects.

6.3.2 System element interactions

Next, to be able to create a suitable system model, it was necessary to analyze the activities performed by the test application code. This analysis was performed separately for the two main steps of the application: (1) initializing the GEM controller and (2) running the loopback test. For each step, transactions traversing the on-chip interconnect were identified. The results of this analysis are summarized in Table 6.3.

During the GEM initialization phase, the Cortex-A53 will read its code and maintain its variables using μ_{A53}, for example. It will also initialize buffer

descriptors in μ_{GEM3} and access `gem3` to initialize this peripheral device. Two further accesses are not directly related to the GEM controller, however; the initialization code also accesses the following peripheral devices:

1) LPD clock and reset control registers (modeled as `crl`)
2) Configuration registers of the CSU (modeled as `csu`)

The configuration registers of the CSU are read to query the platform version. This property is then considered during the initialization procedure:

Listing 6.16: Adapted code excerpt comprising the CSU access

```
#include "xil_io.h"
#include "xil_types.h"
#define CSU_VERSION 0xFFCA0044

// Get platform version to guide the following initialization:
u32 Platform = Xil_In32(CSU_VERSION);
```

While this access could be eliminated by hard-coding knowledge about the underlying platform, accesses to LPD clock and reset control registers are required to configure clocks of the controller. These accesses are an integral part of the GEM initialization procedure and cannot be eliminated.

During the loopback test itself, the Cortex-A53 processor continues to access the RTE memory region (μ_{A53}). It further interacts with `gem3` and `uart0` to transmit frames, receive frames, and report the result. The second memory region (μ_{GEM3}) is read and written by both masters.

For the purposes of the case study, it was therefore necessary to decide at which point in time the APU configuration code would be applied: before or after the GEM initialization step. Each of these variants has its benefits and drawbacks. They were therefore evaluated both.

6.3.3 Variant I: Protected initialization

In this case, the APU configuration code was executed *before* the GEM initialization step. As the name implies, the beneficial property of this approach is that GEM initialization activities were monitored and controlled by an APU. However, the initialization-specific peripheral devices, `csu` and `crl`, had to be allocated to the SWC representing the test application.

Based on this observation, LIP syntax from Section 6.1 was used to describe the system model and request the generation of an APU configuration:

Listing 6.17: Input model for variant I of the case study

```
platform main : zynqmp {
  alloc (0x800000 - 0xFFFFFF) to gem3;
}

rte a53 : main.a53 {
  alloc 0x0 - 0x7FFFFF;
}

swc loopback : a53 {
  alloc gem3, uart0, csu, crl;
  out ~success;
}

pattern : isolation {
  gen_apu main;
}
```

The environment output (`success`) captured the fact that using its `uart0` controller, the test setup communicated with the host computer.

Based on this input model, the '`-gen`' task of `liptool` was used to generate APU configuration code for the Zynq UltraScale+ MPSoC platform. This code was then added to the very beginning of the test application. Executing the application finally resulted in a successful loopback test, which was the primary goal of the Ethernet controller case study.

With respect to the secondary goal, the completeness of the CF graph in Figure 6.6 was checked on a sample basis. Therefore, XSDB was used to simulate certain CFs. Here, especially the allocation of `crl` and `csu` was identified as a major source for CF potential: by writing to any of these peripheral devices, it was possible to reconfigure the execution platform itself. This observation is correctly represented in Figure 6.6. To conclude the CF graph evaluation, in Figure 6.7, the system model was extended with a safety-critical body controller to be executed on the Cortex-R5. The loopback application was labeled with a low inherent integrity, and other susceptible system elements were labeled with a high inherent integrity. For this case, `liptool` correctly returned a negative safety assessment result.

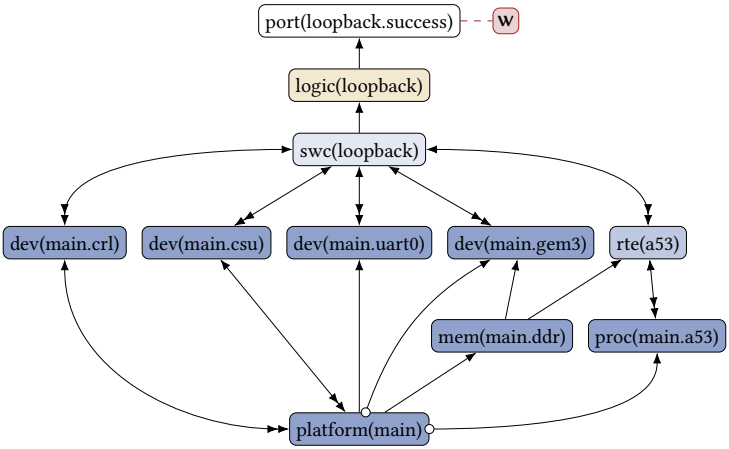


Figure 6.6: CF graph for variant I of the Ethernet controller case study. The two peripheral devices with platform impact, `dev(main.crl)` and `dev(main.csu)`, have outgoing edges to the underlying execution platform.

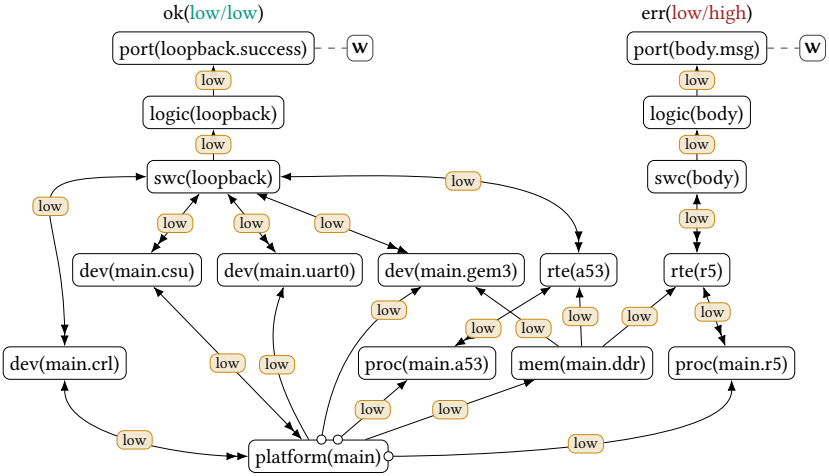


Figure 6.7: Safety assessment based on variant I of the Ethernet controller case study. Given integrity levels $I = \{\texttt{low}, \texttt{high}\}$, a low inherent integrity of `swc(loopback)`, and a high required integrity of `port(body.msg)`, the pattern rejects this attempt to deploy a SWC with safety-critical impact to the Cortex-R5.

6.3.4 Variant II: Unprotected initialization

In this case, APU configuration code was executed *after* the GEM initialization step. This temporarily violated Premise 4.8 and meant that it was not possible to reason about the presence of unreasonable safety risk *during* the initialization procedure. In practice, however, such procedures are typically executed during a well-defined startup phase. During such phases, it might be possible to show that physical harm cannot be caused.

Therefore, in variant II of the case study, initialization-specific peripheral devices (`csu` and `crl`) were no longer allocated to the SWC:

Listing 6.18: Input model for variant II of the case study

```
platform main : zynqmp {
  alloc (0x800000 - 0xFFFFFF) to gem3;
}

rte a53 : main.a53 {
  alloc 0x0 - 0x7FFFFF;
}

swc loopback : a53 {
  alloc gem3, uart0;
  out ~success;
}

pattern : isolation {
  gen_apu main;
}
```

Based on this LIP input, APU configuration code was generated and inserted immediately *after* the end of the GEM initialization procedure. Executing this application resulted in a successful loopback test. Therefore, this design (like the one from variant I) met the primary goal of the case study.

With respect to the secondary goal, the auto-generated CF graph in Figure 6.8 was again analyzed and compared with CF potential in the hardware setup. In comparison to Figure 6.6, this version of the graph does no longer contain a directed edge to the underlying execution platform. This is due to the fact that `csu` and `crl` were no longer in use, i.e., they were not part of $\hat{C}_+$ as defined in Section 4.1.1.1. Therefore, they did no longer represent system elements and,

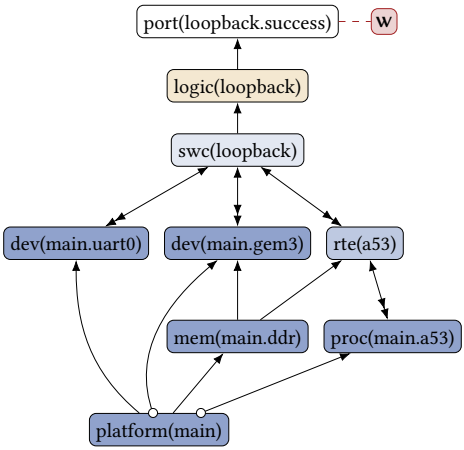


Figure 6.8: CF graph for variant II of the Ethernet controller case study. Compared with variant I, there are no longer edges leading to the execution platform itself.

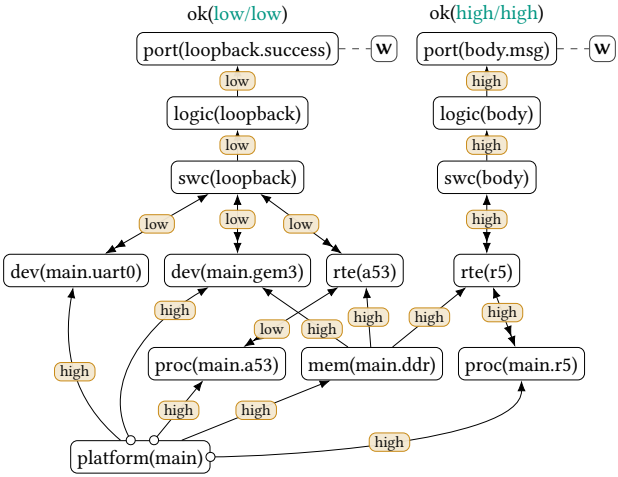


Figure 6.9: Safety assessment based on variant II of the Ethernet controller case study. Despite the low integrity of `swc(loopback)`, the pattern accepts this attempt to deploy a SWC with safety-critical impact to the Cortex-R5.

according to Premise 4.1, were regarded as unable to propagate failures. The fact that the APU was configured to prevent any Cortex-A53 core from accessing these peripheral devices supported the pattern user in verifying that this premise is valid. To conclude the CF graph evaluation, it was again attempted to extend the system model with a safety-critical body control SWC. Like for variant I, in Figure 6.9, this SWC was mapped to an RTE of the Cortex-R5. The inherent integrity of the test application was kept low, while other susceptible system elements were assigned a high inherent integrity. As shown in the CF graph, this analysis led to a positive safety assessment result, which is the expected outcome for this case study variant.

6.4 Case study: Fail-operational architecture

CF potential captures not only how the failure of one system element can propagate to logically unrelated system elements, e.g., via a shared memory module. It also considers CFs due to direct dependencies, for instance:

1) The failure of a processing unit causes its SWCs to fail.
2) An algorithm used to drive an actuator is faulty.

Such factors are not directly related to APU configurations. However, they influence the integrity of individual subsystems and can therefore have an impact on the logical isolation that is required *between* subsystems.

One such property is the sufficiency of applied redundancy measures. Redundancy is often required if failed system portions cannot be transitioned into a safe state—in time or at all. Failures of software controlling an autonomous road vehicle, for example, cannot always be handled by deactivating the implemented function. Another example of systems without a safe state are drive-by-wire architectures without a mechanical fallback. ISO 26262-10 [154] refers to such systems as exhibiting 'safety-related availability' requirements. In this section, the following terminology is used instead:

► **Definition 6.1:** Fail-operational system

A *fail-operational system* must be able to maintain a certain minimum level of functionality, even after the manifestation of specified faults.

The author of this thesis published a comparable definition in [10]. As stated there, this definition is in line with terminology used in [155] and [156].

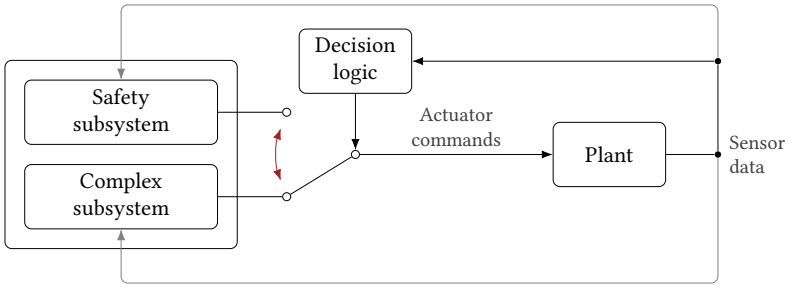


Figure 6.10: Logical view of the system-level simplex architecture according to [159]. Depending on plant behavior, the decision logic activates one of the subsystems to generate actuator commands. The safety subsystem and the decision logic must be executed on dedicated, trusted hardware.

However, it differs from terminology that separates full fail-operational behavior from degraded fail-operational [156] or active fail-safe [157] behavior.

From a control systems perspective, a possible technique to meet the requirements of fail-operational systems is to apply the simplex architecture from [158]. It is based on the idea of partitioning a control system into two parts: a *high-assurance-control* subsystem and a *high-performance-control* subsystem. Decision logic is able to activate either the former or the latter at any time. The high-performance controller provides a superior performance, while the high-assurance controller is able to guarantee a safe operation of the plant. To prevent failures of the high-performance controller from causing physical harm, the decision logic monitors the plant to recognize imminent safety requirement violations. If it detects an imminent violation, it activates the high-assurance controller. This way, the concept provides robustness against systematic faults in the design of the high-performance controller. It is applicable if incorrect actions can be tolerated and recovered from.

The system-level simplex architecture [159] is an extended variant of this approach. As visualized in Figure 6.10, the core structure is kept, but the employed terminology is changed: the high-performance-control subsystem is now referred to as the *complex* subsystem, while the high-assurance-control subsystem becomes the *safety* subsystem. It introduces partitioning constraints to add robustness against additional fault types: both the safety subsystem and the decision logic must be implemented on isolated, trusted hardware. In the particular concept from [159], this is custom hardware, for example in the form of an Application-Specific Integrated Circuit (ASIC). This way, failures

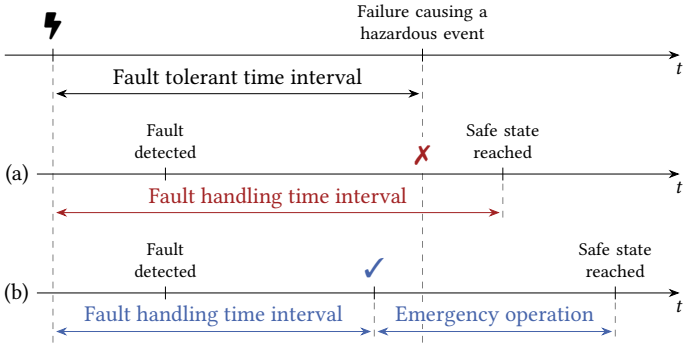


Figure 6.11: Fault-related time intervals defined by ISO 26262. Visualization published in and adapted from [10]. Mechanism (a) is unable to transition the system into a safe state before the failure may cause physical harm. Mechanism (b) transitions the system into an emergency operation mode to prevent this.

of the employed microprocessor, its operating system, and other underlying components can be addressed and handled.

Handling failures of a critical component by switching to a safety-oriented fallback functionality is an interesting concept to tackle fail-operational requirements—even in cases when the failure cannot be detected by monitoring the plant. It can be used to implement an *emergency operation* mode as defined by ISO 26262-1 [16], for example. As shown in Figure 6.11, the emergency mode supports designers in reacting to conditions in which a safe state cannot be entered before the fault tolerant time interval expires.

Therefore, in the first step of this case study, a fail-operational architecture inspired by the system-level simplex architecture was defined as follows:

1) Both a *complex* and a *safety* algorithm are available to drive a plant.
2) At any time, a *decision* algorithm connects either the complex or the safety algorithm to the plant. When the complex algorithm is operating correctly, it is selected. Otherwise, the safety algorithm is selected.
3) Both the safety and the decision algorithm are free of systematic faults. By themselves, neither of them will lead to unsafe plant behavior.
4) The decision logic is able to detect failures of the complex algorithm, e.g., by monitoring the plant.

Two aspects are deliberately *not* covered by this architecture specification. First, it does not describe the fault susceptibility of the complex algorithm. If

the complex algorithm is susceptible to systematic faults, its failure can be due to an inherent fault or a CF affecting its execution. If the algorithm is free of systematic faults, it can only fail if a CF affects its execution.

Furthermore, the actual *implementation* of the architecture is not yet specified. Using the terminology introduced so far, we can say that the context used to execute these algorithms (SWCs, RTEs, ...) is still unknown.

In this case study, three variants of the architecture were considered. For each of them, the base specification from above was completed in a particular manner. Afterward, the logical isolation pattern's applicability for reasoning about certain safety properties was evaluated.

6.4.1 Mapping to cores of a single MPSoC

In this variant, the complex algorithm was considered as potentially affected by systematic faults. Each algorithm was modeled as a dedicated SWC. We refer to them as `complex`, `safety`, and `decision`, respectively. They were mapped to processing units of a single Zynq UltraScale+ MPSoC instance:

1) `complex` → Cortex-A53 processor
2) `safety` → Cortex-R5 processor
3) `decision` → Cortex-R5 processor

To facilitate this mapping, each processing unit was associated with a dedicated RTE, each with its own DDR memory region. Communication between the RTEs was modeled as taking place via another DDR memory region:

Listing 6.19: System model of the MPSoC mapping

```
platform main : zynqmp;
rte a53 : main.a53 { alloc 0x0 - 0xFFFFFF; }
rte r5 : main.r5 { alloc 0x1000000 - 0x1FFFFFF; }
swc complex : a53 { out cmd; }
swc safety : r5 { out cmd; }
swc decision : r5 {
  in complex, in safety;
  out ~res;
}

channel complex.cmd -> decision.complex;
channel safety.cmd -> decision.safety;
path {a53, r5} = 0x10000000 - 0x1000FFFF;
```

▹ Remark 6.3: *This strategy is similar to the approach presented in [8], where the safety controller is executed on a soft-core processor in an FPGA.*

In this code, environment output 'decision.res' represents the outgoing plant interface. To be able to implement this interface, the SWC will—in practice—need to allocate one or more peripheral devices. For clarity, this allocation is not modeled here. Another aspect not reflected in the system model is the fact that the two controllers (complex and safety) will also need to read sensor data from the plant. This aspect can be modeled using the logical isolation pattern, but it is hidden here for the sake of brevity.

To reason about the acceptability of CF potential, two integrity levels were introduced: $I = \{\texttt{low}, \texttt{high}\}$ with $\texttt{low} \leq \texttt{high}$ (and, to achieve reflexivity, $\texttt{low} \leq \texttt{low}$ and $\texttt{high} \leq \texttt{high}$). Based on this, the required integrity of port(decision.res) was set to high. Roughly speaking, this means that a susceptible system element can be labeled with an inherent integrity of high if the following condition is met: its failure is sufficiently rare and, therefore, CF potential leading to port(decision.res) is acceptable:

Listing 6.20: Pattern application for the MPSoC mapping (1/2)

```
pattern : isolation {
  lattice {
    high >= low;
  }

  port(decision.res) >= high;
```

From the specification described so far, it was derived that this property applies to logic(safety) and logic(decision). Furthermore, it was argued that the following system elements are built in such a way that it is also justified to set their inherent integrity to high:

1) platform(main), proc(main.r5), and dev(main.ddr)
2) rte(r5), swc(decision), and swc(safety)

Other susceptible system elements were assigned an inherent integrity of low.

The premise that logic(decision) detects failures of logic(complex) was translated into an input barrier leading from port(decision.complex) to logic(decision). Finally, the ILP approach from Section 5.4.2 was used to decide whether APU configuration code or the implementation of process isolation is required to achieve absence of unreasonable risk.

For these inputs, the ILP approach returned that it is necessary to apply (only) an APU configuration. Therefore, and considering the statements from above, the pattern application was completed as follows:

Listing 6.21: Pattern application for the MPSoC mapping (2/2)

```
  gen_apu main;
  platform(main) = high;
  proc(main.r5) = high;
  mem(main.ddr) = high;
  rte(r5) = high;
  swc(safety) = high;
  logic(safety) = high;
  swc(decision) = high;
  logic(decision) = high {
    no_in! complex;
  }
}
```

For completeness, an explicit application of the pattern was performed to confirm that this design does not pose an unreasonable risk. This verification was successful, as visualized by the annotated CF graph in Figure 6.12.

Note that this result would not be achievable without an APU configuration: `swc(complex)`, `rte(a53)`, and `proc(main.a53)` were all labeled with an inherent integrity of `low`. Unless this assignment is changed, the lack of an APU configuration would mean that they could interfere with a safety-critical region of the DDR memory, for example.

This demonstrates that the pattern is able to both contribute to and reason about the safe on-chip integration of the architecture.

6.4.2 Mapping to distributed MPSoC instances

Here, a variation of the previous implementation strategy was defined: instead of mapping all SWCs to the same MPSoC, two MPSoC instances were used. For the purposes of this section, we refer to these instances as `x` and `y`. Based on this, the following mapping was performed:

1) `complex` → Cortex-A53 processor of the first instance (i.e., of `x`)
2) `safety` → Cortex-R5 processor of the second instance (i.e., of `y`)
3) `decision` → Cortex-R5 processor of the second instance (i.e., of `y`)

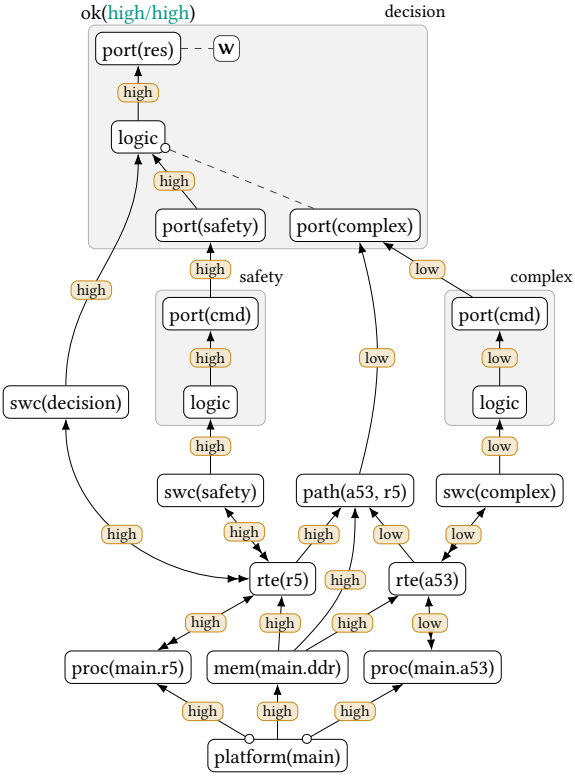


Figure 6.12: CF graph for the single-MPSoC mapping from Section 6.4.1. Annotations show that the effective integrity at `port(decision.res)` is sufficient, i.e., that this design does not pose an unreasonable safety risk.

To facilitate this mapping, one RTE with a dedicated DDR memory region was again specified for each utilized processing unit. Due to the distributed deployment of both RTEs, communication between them made it necessary to define a global path (instead of a local one). For the purposes of the case study, this path was implemented using a CAN network. This also meant that each RTE had to be extended with a CAN controller allocation, and that this CAN controller had to be attached to the CAN network.

Apart from that, the previous system model was kept unchanged. Most importantly, the environment output '`decision.res`' still represented the

outgoing plant interface, and additional plant interactions were not modeled for brevity. This led to the following LIP formulation:

Listing 6.22: System model for the distributed mapping

```
platform x : zynqmp;
platform y : zynqmp;
rte a53 : x.a53 { alloc 0x0 - 0xFFFFFF, can0; }
rte r5 : y.r5 { alloc 0x1000000 - 0x1FFFFFF, can0; }
swc complex : a53 { out cmd; }
swc safety : r5 { out cmd; }
swc decision : r5 {
  in complex, in safety;
  out ~res;
}

channel complex.cmd -> decision.complex;
channel safety.cmd -> decision.safety;
bus can = x.can0, y.can0;
path {a53, r5} = can;
```

The integrity lattice from before (`high >= low`) was kept, and the required integrity of the plant interface was again set to `high` (cf. Listing 6.20). The second part of the pattern application was changed to the following:

Listing 6.23: Inherent integrities and isolation measures

```
// Part 1 from previous variant...
  platform(y) = high;
  proc(y.r5) = high;
  dev(y.can0) = high;
  mem(y.ddr) = high;
  rte(r5) = high;
  swc(safety) = high;
  logic(safety) = high;
  swc(decision) = high;
  bus(can) = high;
  logic(decision) = high {
    no_in! complex;
  }
}
```

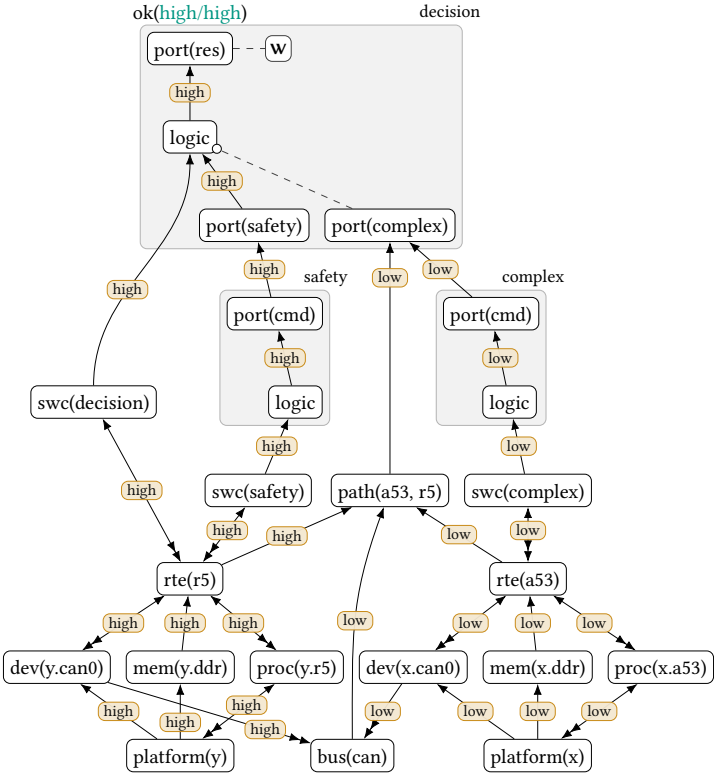


Figure 6.13: CF graph for the mapping from Section 6.4.2. Although no APU configurations are applied, the effective integrity at `port(decision.res)` is sufficient. This is due to the physical separation between the two execution platforms.

As before, the application-level barrier from `port(decision.complex)` to `logic(decision)` was used to capture that `decision` detects and handles failures of the `complex` algorithm. APU configuration code was not requested, because doing so was not necessary for this variant.

As illustrated in Figure 6.13, this design is accepted by the logical isolation pattern. It is important to emphasize that this positive safety assessment result was achieved without requesting any APU configuration code to be generated. For this variant, applying the *analysis* features of the pattern was sufficient to infer the absence of unreasonable risk.

6.4.3 Background: Mirrored architecture concept

The third case study variant is based on a 'mirrored superposition' of two instances of the architecture. This concept is a secondary contribution of this thesis and was published in [9] and [10]. Before its compatibility with the pattern is covered in Section 6.4.4, it is now introduced in more detail.

▹ Remark 6.4: *The content of this background section is a summary of the author's contributions to [10]. For a more detailed coverage of the concept and its prototypical implementation, the reader is referred to this source.*

The mirrored architecture concept is concerned with random hardware faults. Systematic faults introduced during the development of hardware or software are beyond the scope of the approach. Considered hardware faults can be transient, intermittent, or permanent. For simplicity, the remainder of this section treats intermittent faults as a simple repetition of transient ones. With this, it is possible to say that the approach is concerned with random hardware faults of transient or permanent duration.

In practice, transient faults are considerably more likely to occur than permanent ones [160]. Since permanent faults have a significant impact on the system behavior, however, they must still be considered and often handled through the application of space redundancy techniques. One such technique is Triple Modular Redundancy (TMR), which can be applied at various levels of the system hierarchy. Especially if it is applied at the system level, however, triplicating critical components can be prohibitively expensive. This is especially true if these components are heterogeneous MPSoCs instead of lightweight microcontrollers. The mirrored architecture concept is a fault tolerance scheme that attempts to exploit existing redundancy at the system level to achieve fail-operational behavior in a cost-efficient manner.

The mirrored fail-operational architecture handles transient and permanent faults by switching to a degraded version of the full system functionality. In the case of transient faults, the full functionality is restored by resetting the faulty hardware and reversing the transition. In the case of permanent faults, the degraded functionality can be used to achieve a temporary emergency operation (according to ISO 26262) before a safe state is entered.

The approach is applicable whenever there is a pair of execution platforms, each with spare computational resources that cannot be removed. These resources are referred to as *inherent redundancy* and can be due to the fact that a commercially available MPSoC is not fully utilized, for example. The approach is now described using an application from the automotive domain, but it can be transferred to a wide variety of use cases.

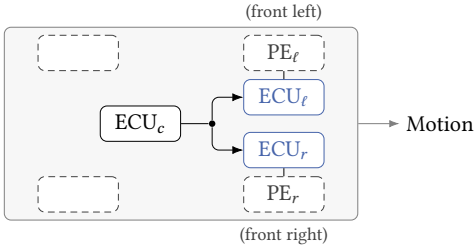


Figure 6.14: Road vehicle with two electric wheel hub motors. Each front wheel integrates the power electronics of a motor (PE$_\ell$ and PE$_r$). Electronic control units attached to each of the front wheels (ECU$_\ell$ and ECU$_r$) receive commands from a central car server (ECU$_c$) to control the motor they are associated with.

6.4.3.1 System-level concept

The sample application is a road vehicle with two electric wheel hub motors. As shown in Figure 6.14, it comprises a power electronics module attached to both the left and the right front wheel; they are referred to as PE$_\ell$ and PE$_r$, respectively. Each module is controlled by a dedicated Electronic Control Unit (ECU); they are referred to as ECU$_\ell$ and ECU$_r$, respectively. To control its respective motor, each ECU reads from sensors integrated into the wheel and drives several digital outputs, most importantly to transmit Pulse-Width Modulation (PWM) signals. Due to the safety-critical nature of this application, each of the ECUs needs to be implemented as a fail-operational system. Ideally, their read and write operations are executed with a frequency of 10 kHz, but a reduction of this frequency is possible, e.g., during emergency operation.

ECU$_c$ is a central controller that provides both ECU$_\ell$ and ECU$_r$ with setpoints they are expected to maintain, such as torque or rotational frequency. For the purposes of this section, ECU$_c$ is assumed to be free of faults. ECU$_\ell$ and ECU$_r$ are subject to random hardware faults and exhibit inherent redundancy, i.e., their nominal control task does not lead to a full utilization. Software executed on this pair of ECUs is trusted, i.e., not subject to systematic faults.

As shown in Figure 6.15, the mirrored version of the fail-operational architecture is applied to ECU$_\ell$, ECU$_r$, and the communication with both power electronics modules (cf. Figure 6.15). The key idea is as follows:

1) Failures of ECU$_\ell$ are handled by controlling PE$_\ell$ from ECU$_r$.
2) Failures of ECU$_r$ are handled by controlling PE$_r$ from ECU$_\ell$.

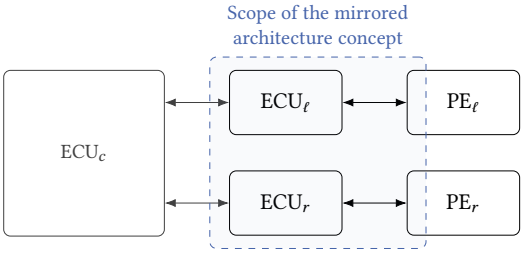


Figure 6.15: Road vehicle use case visualized as an inherently redundant system with two channels. The scope of the mirrored simplex architecture comprises ECU$_\ell$, ECU$_r$, and the communication with both power electronics modules.

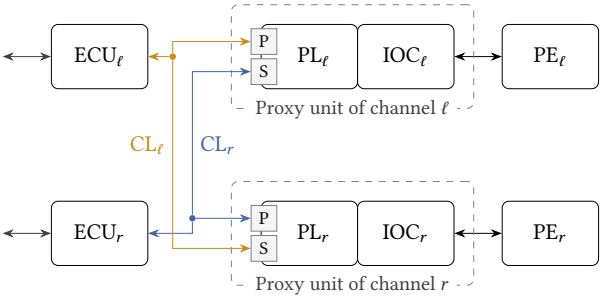


Figure 6.16: Introduction of a proxy unit into both channels. Each proxy unit consists of its proxy logic (PL$_\ell$/PL$_r$) and all I/O components that are necessary to interact with the respective power electronics module (IOC$_\ell$/IOC$_r$). Communication links (CL$_\ell$/CL$_r$) connect each ECU to both the primary port (P) of its own and the secondary port (S) of its opposite proxy logic.

To facilitate this, the inherent redundancy of each ECU must allow for the execution of (at least) a degraded version of the opposite wheel's control algorithm. In addition, the power electronics interface must be extended with decision logic that selects exactly one ECU as the active one. This component is referred to as *proxy logic* and will be referred to as PL$_\ell$ and PL$_r$, respectively. Finally, the issue of an increased physical distance between each ECU and its opposite wheel needs to be addressed. To tackle it, relevant input/output components (such as a PWM controller) are also merged into the power electronics interface. These components are referred to as IOC$_\ell$ and IOC$_r$, respectively. Together with their proxy logic, they are the *proxy unit* of a particular channel.

Figure 6.16 visualizes the introduction of these proxy units. Via system-level communications links, such as a CAN network, each proxy unit is connected to both ECUs; the ECU connected to the primary port (P) controls the proxy unit during normal operation, while the ECU connected to the secondary port (S) controls it after a failure. This symmetrical structure can be described as 'mirrored' with respect to each channel.

The proxy unit is responsible to detect failures of the ECU at its primary port, and it needs to react to such failures by switching to the ECU at its secondary port. Therefore, a challenge-response watchdog is used: the proxy unit transmits periodic watchdog challenges to the ECU at its primary port. Software on this ECU needs to solve this challenge and transmit the response back to the proxy unit. If the proxy unit receives an erroneous or no response, it deducts a failure of the ECU (or the associated communication link).

A final task of the proxy unit is to keep track of the application state: periodically, the ECU that is currently responsible for the control of a particular wheel transmits its state variables to the proxy logic. The proxy logic stores these state variables internally and, whenever a switch to the opposite ECU takes place, provides this ECU with the most recent set of state variables.

This architecture imposes one requirement on the availability of ECUs and CL, and one requirement on the reliability of proxy units:

1) ECUs and communication links are subject to random hardware faults. At any time, however, there is a sufficient[3] probability that at least the left pair (ECU_ℓ and CL_ℓ) or the right pair (ECU_r and CL_r) works correctly.
2) The probability that a proxy unit fails is negligible, i.e., its reliability is in line with the requirements of the power electronics.

Under the assumption that these requirements are fulfilled, the mirrored architecture concept can be described by the following properties:

1) The system exhibits fail-operational behavior with respect to PE_ℓ and PE_r.
2) Components introduced during the transformation affordable. Most importantly, no additional ECU needs to be introduced.
3) ECUs and power electronics can remain at their physical locations.
4) Spare resources used to control the opposite power electronics module remain available during normal (i.e., fault-free) operation.
5) State variables are maintained during mode switches.

[3]The probability is sufficient if it is in line with the requirements of the power electronics.

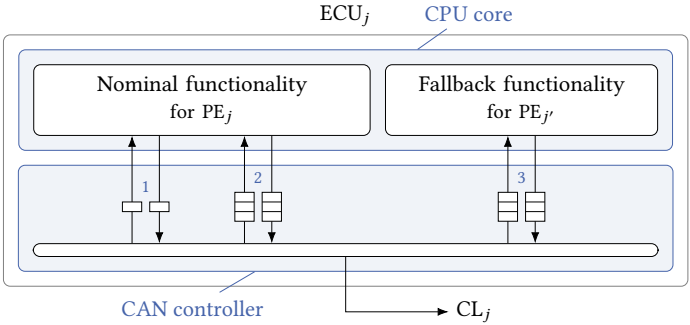


Figure 6.17: AURIX configuration used to implement ECU$_j$ with $j \in \{\ell, r\}$. Both the nominal functionality for PE$_j$ and the fallback functionality for PE$_{j'}$ were implemented as a shared bare-metal application. Three hardware-managed receive/transmit queues were created for CAN messages. One of them (1) was dedicated to the watchdog mechanism and handled with the highest priority. Other queues were used for all remaining messages; the priority of (2) was configured above that of (3).

6.4.3.2 Prototypical implementation

As part of this work, the mirrored architecture variant was prototypically implemented. Therefore, a TriBoard (featuring an AURIX microcontroller by Infineon Technologies) was used to realize each of the ECUs. Each communication link was realized as a CAN bus combined with a direct reset wire.

For rapid prototyping of the proxy units, a custom FPGA implementation was developed. More specifically, the FPGA portion of a ZedBoard (featuring a Zynq-7000 by AMD/Xilinx) was used to implement each proxy unit. The logic mapped to this FPGA portion was a TMR arrangement of the proxy logic, two CAN controllers, a PWM controller, and other input/output functions. Custom voting logic was added to complete the TMR implementation. The interaction with ECU$_c$ was not implemented.

For $j \in \{\ell, r\}$, the schematic in Figure 6.17 shows how the two control algorithms were mapped to ECU$_j$: a shared bare-metal application was developed to simulate both the nominal functionality for PE$_j$ and the fallback functionality for PE$_{j'}$, where j' refers to the opposite channel of j. The built-in CAN controller was configured to hold three bidirectional, hardware-managed message queues: one for watchdog messages, one for management and payload messages of the nominal functionality, and a third one for management and payload messages of the fallback functionality.

		Synchronized state size (N)				
T_{cycle}	T_{wdg}	32 bit	64 bit	128 bit	256 bit	512 bit
1 ms	1 ms	●	○	○	○	○
2 ms	1 ms	●	●	●	○	○
2 ms	2 ms	●	●	●	○	○
4 ms	1 ms	●	●	●	●	○
4 ms	2 ms	●	●	●	●	○
4 ms	4 ms	●	●	●	●	○

Table 6.4: Design space spanned by T_{cycle}, T_{wdg}, and N. For each point in this space, the feasibility calculated in [10] is shown: parameter combinations labeled with '●' are feasible, while parameter combinations labeled with '○' are infeasible.

Both the nominal and the fallback functionality were implemented as independent periodic procedures, both executed with a cycle time of T_{cycle}. During each cycle, both functionalities were designed to perform three tasks:

1) Read 32 bit of data from the power electronics (sensor data).
2) Write 32 bit of data to the power electronics (PWM voltages).
3) Write a state of size N to the proxy logic's state buffer.

In addition, an interrupt-based handler for watchdog challenges was integrated into the nominal functionality. The proxy logic was designed to issue watchdog challenges with a period of T_{wdg} and to expect a correct response within a time of T_{deadline} after issuing a watchdog challenge.

Then, a feasibility analysis of various $\langle T_{\text{cycle}}, T_{\text{wdg}}, N \rangle$ combinations was performed. During this analysis, a system was considered *feasible* if after the detection of a fault, all CAN messages required to (1) perform the mode transition and (2) transmit the first set of payload messages do not exceed T_{cycle}. For a CAN frequency of $f_{\text{CAN}} = 1\,\text{MHz}$, the obtained results are shown in Table 6.4. For a cycle time of $T_{\text{cycle}} = 1\,\text{ms}$, for example, the theoretical analysis showed that a system with $T_{\text{wdg}} = 1\,\text{ms}$ and $N = 32\,\text{bit}$ is feasible. This feasible cycle time is below the ideal frequency of 10 kHz (see above), but it was still possible to argue an applicability in certain operating modes, e.g., for a degraded operation of the wheel hub motor.

It is possible to achieve higher operating frequencies by switching to a communication link with an increased data rate or reducing the frequency of state synchronizations, for example. During the case study, however, this design space was not explored further.

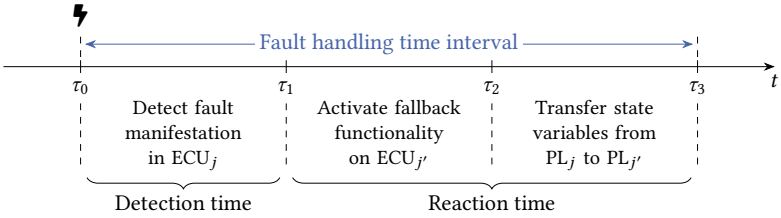


Figure 6.18: Decomposition of the fault handling time interval (according to ISO 26262) into three sequential steps. The former describes the fault detection phase, while the latter two capture fault reaction tasks.

	Duration / ms		
Fault handling interval	Minimum	Average	Maximum
Fault detection ($\tau_0 \rightarrow \tau_1$)	0.40	0.89	1.40
Fallback activation ($\tau_1 \rightarrow \tau_2$)	0.11	0.20	0.52
State transfer ($\tau_2 \rightarrow \tau_3$)	0.14	0.20	0.53

Table 6.5: Fault handling durations for $T_{\text{cycle}} = 1\,\text{ms}$, $T_{\text{wdg}} = 1\,\text{ms}$, and $N = 32\,\text{bit}$. Per interval, 300 measurements were performed, and the table lists the minimum, the maximum, and the average value of these measurements.

6.4.3.3 Evaluation results

The prototypical implementation was used to evaluate different metrics, including the achievable fault handling time and the computational overhead that the architecture causes in both ECUs.

The fault handling time interval is the duration it takes for a proxy unit to (1) detect the fault of its primary ECU, (2) activate the fallback functionality of its secondary ECU, and (3) read the most recent state variables from the proxy logic's state buffer. Figure 6.18 visualizes this interval decomposition.

Due to the symmetrical setup, fault handling times were evaluated for only one of the channels. To do so, a fault injection module was added to the FPGA portion of PL$_\ell$. It was able to activate the external reset pin of ECU$_\ell$, i.e., to simulate the complete failure of this ECU. For each feasible design, this mechanism was triggered and measured 300 times. A watchdog deadline of $T_{\text{deadline}} = 0.5\,\text{ms}$ was used, and ECUs were operated at $f_{\text{CPU}} = 200\,\text{MHz}$.

Table 6.5 shows minimum, maximum, and average durations that were measured for the design with $T_{\text{cycle}} = 1\,\text{ms}$, $T_{\text{wdg}} = 1\,\text{ms}$, and $N = 32\,\text{bit}$. With

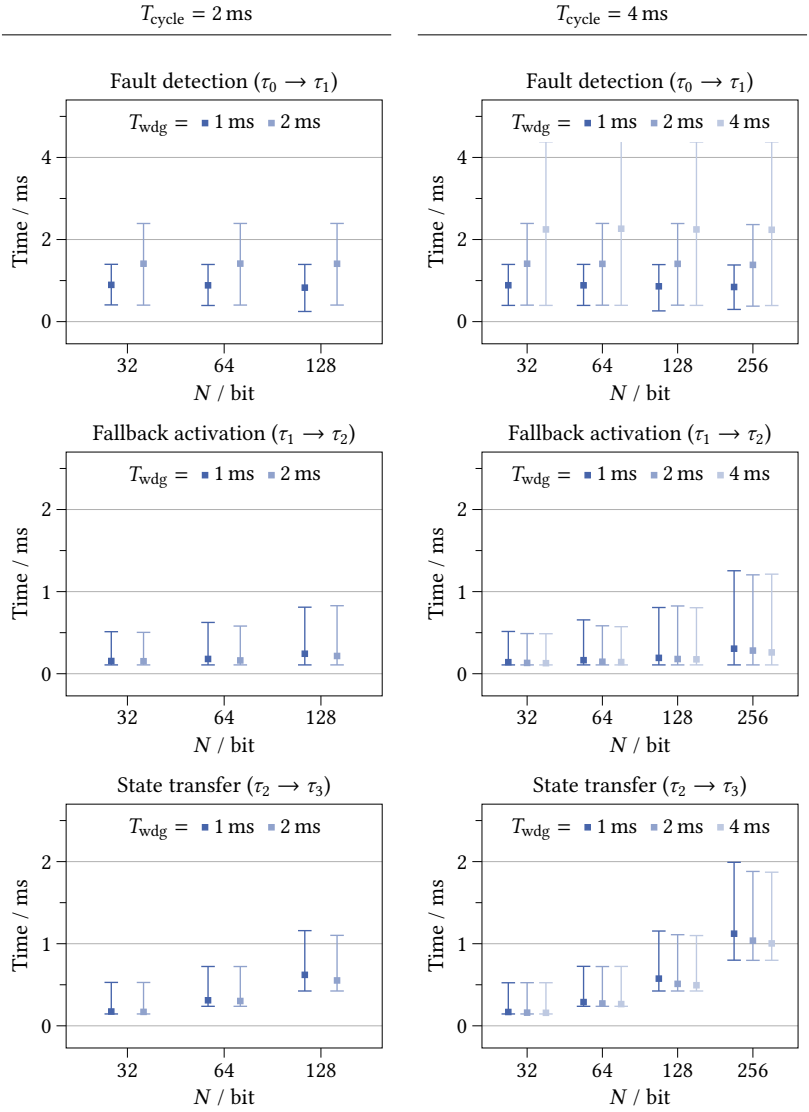


Figure 6.19: Measured fault handling durations for all designs from Table 6.4 with either T_{cycle} = 2 ms or T_{cycle} = 4 ms. For each design, 100 measurements were performed, and the corresponding bar visualizes the range and average of measured durations.

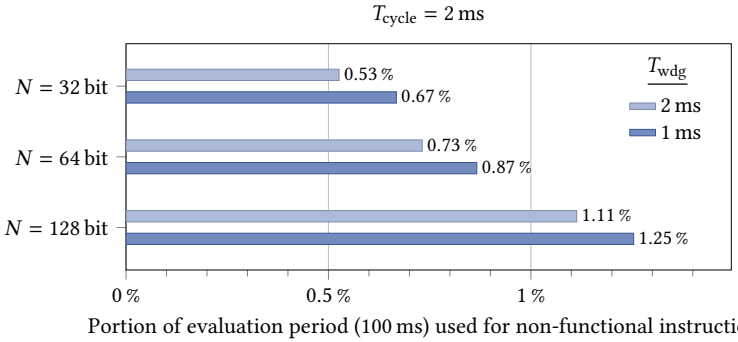


Figure 6.20: Relative number of clock cycles that an AURIX with f_{CPU} = 200 MHz spent executing non-functional tasks during a 100 ms interval. The plot shows maximum values obtained for 300 repeated measurements in a worst-case setup.

this design, the proxy logic was able to detect the failure of its primary ECU in under 1.5 ms. It then required up to 1 ms to activate the fallback functionality on its secondary ECU and provide this fallback functionality with the most recent set of state variables. Therefore, it was able to handle random hardware faults in the same order of magnitude as the cycle time.

For the 18 remaining designs from Table 6.4, the measured fault handling times are visualized in Figure 6.19. The results show, for example, that the watchdog challenge frequency, T_{wdg}, had a significant impact on the duration of the fault detection interval. For all considered designs, the maximum measured fault handling time remained below 10 ms.

Finally, to determine the computational overhead in an ECU, the relative number of clock cycles that were required to execute non-functional tasks during a reference interval was determined. Non-functional tasks comprise solving and responding to watchdog challenges, managing the transfer of internal state variables, and interacting with the CAN controller. To perform this measurement, one AURIX controller was put into a mode in which it had to deliver both the nominal functional for its own and the fallback functionality for the opposite power electronics. This corresponds to the highest computational overhead that the architecture can cause, i.e., it is the worst-case setup. For this specific case, integrated performance counters of the AURIX were used to determine the number of clock cycles spent to execute non-functional tasks. For a cycle time of T_{cycle} = 2 ms, the obtained results are shown in Figure 6.20; they did not exceed a value of 1.25 percent.

6.4.4 Applicability to the mirrored variant

From the perspective of each proxy unit, the terminology from Section 6.4.3 can be mapped to the generic architecture as follows:

1) Its integrated proxy logic implements the decision algorithm.
2) The nominal functionality from port P is its complex algorithm.
3) The fallback functionality from port S is its safety algorithm.

To complete the case study, it was determined if the logical isolation pattern is able to represent the fault tolerance strategy integrated into this concept.

Therefore, it was assumed that each proxy unit, which was previously evaluated in custom hardware, would be implemented on an execution platform. Like in the prototypical implementation, each ECU was assumed to run a shared bare-metal application implementing both the nominal and the fallback functionality. To represent this theoretical scenario using LIP syntax, a special execution platform type (`generic`) was used. It does *not* come with an APU configuration procedure, but it provides the pattern user with an arbitrary number of processing units (`cpu0`, `cpu1`, …), selected I/O controllers (`can0`, `can1`, …), and a large DDR memory.

Based on this, a symmetrical system model was defined. For the left architecture channel (ECU_ℓ, CL_ℓ, and proxy unit ℓ), suitable entities were created and connected (to each other and their respective counterparts):

Listing 6.24: System model for the left architecture channel

```
// Left controller (ECU):
platform cl : generic;
rte cl : cl.cpu0 { alloc can0; }
swc cl : cl { out nominal, out fallback; }
// Left proxy unit:
platform pl : generic;
rte pl : pl.cpu0 { alloc can0, can1; }
swc pl : pl { in prim, in sec, out ~res; }
// Infrastructure of the left proxy unit:
channel cl.nominal -> pl.prim;
channel cl.fallback -> pr.sec;
bus left = cl.can0, pl.can0, pr.can1;
path {cl, pl} = left;
path {cl, pr} = left;
```

The excerpt for the right channel is analogous but omitted for brevity.

In comparison to Section 6.4.1 and Section 6.4.2, the mirrored architecture variant has a considerably different fault model: it assumes that neither algorithm is subject to systematic faults. It also requires that all RTEs, all SWCs, and the hardware used to implement the proxy unit are fault-free. However, it considers the possibility that any ECU will be affected by random hardware faults. Compared with the previous mappings, this means that there is *no* control algorithm that by itself has a reliable hardware foundation.

Therefore, a third integrity level, `mid`, was introduced. It is meant to label hardware entity pairs that will not fail at the same time, i.e., both ECUs along with their CLs (cf. Section 6.4.3). However, this integrity level alone does not meet the integrity requirement of the power electronics interface. Although not strictly necessary in this case, logic decomposition was further applied to capture that each controller SWC (i.e., `cl` and `cr`) hosts two algorithms, one for the nominal and one for the fallback functionality:

Listing 6.25: Excerpt of the safety pattern application

```
pattern : isolation {
  lattice { high >= mid >= low; }

  logic(cl.nominal) = high { no_out! fallback; }
  logic(cl.fallback) = high { no_out! nominal; }

  logic(cr.nominal) = high { no_out! fallback; }
  logic(cr.fallback) = high { no_out! nominal; }

  port(pr.res) >= high;
  port(pl.res) >= high;

  platform(cr) = mid;
  // Other hardware entities...
```

At this point, however, an essential aspect of the architecture's fault tolerance strategy cannot be represented using the logical isolation pattern: the decision algorithms (`pl` and `pr`) are able to *elevate* the effective integrity they receive from `mid` to `high`. In other words: under the condition that the driver of a proxy unit's primary port (`prim`) does not fail at the same time as the driver of its secondary port (`sec`), it is justified to output a `high` effective integrity. In theory, this circumstance could be represented using *conditional* input barriers.

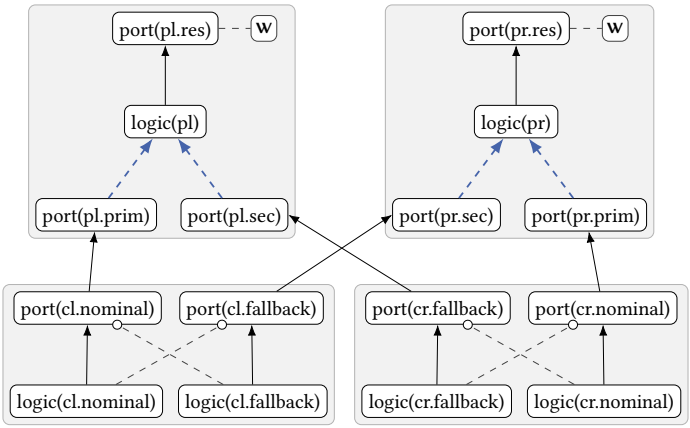


Figure 6.21: Possible application subgraph for the mirrored architecture variant. Dashed arrows highlight CF graph edges for which a conditional input barrier could be argued. If a pair of such edges has an effective integrity of `mid`, it would be justified to raise the effective integrity leaving the logic block to `high`.

Here, they would have to require an effective integrity of at least `mid` and could then be applied to all four edges highlighted in Figure 6.21. Such input barriers are currently not supported by the pattern, however. The safety assessment result the pattern *is* able to deliver will therefore be too pessimistic to show the absence of unreasonable risk.

This is because the pattern was designed to achieve logical isolation, not fault tolerance. If fault tolerance aspects can be reduced to the requirement that less trusted system elements must not interfere with critical ones, the pattern can be used to reason about such aspects. This was the case in Section 6.4.1 and Section 6.4.2. In general, however, its analysis features are unable to capture arbitrary fault tolerance strategies.

6.5 Summary

The utility described in Section 6.2 is a complete implementation of the logical isolation pattern. Its input notation can be used to express all aspects from the formal system model, apply all supported isolation measures, and specify all supported safety requirements. This demonstrates the practical feasibility of key concepts from the previous chapters. It further allows us to consider

the pattern and its implementation as equivalent—what cannot be represented in LIP syntax is not supported by the pattern itself.

The case study in Section 6.3 demonstrated that generated APU configuration code is applicable to commercially available execution platforms, such as the Zynq UltraScale+ MPSoC. Despite the restrictive nature of this code, the pattern was able to generate it in such a manner that full access to the platform's Ethernet controller remained possible. Compared with the built-in configuration feature of the vendor toolchain, the logical isolation pattern comes with CF determination and safety assessment procedures that were shown to capture the potential for possibly unsafe CFs.

The case study in Section 6.4 demonstrated that the pattern's APU configuration and safety assessment capabilities are also applicable to fault-tolerant system designs. For the on-chip variant, for example, the pattern was able to generate APU configuration code that was a necessary requirement to achieve the desired fault tolerance properties. At the same time, the case study identified limitations in the pattern's ability to reflect symmetrical fault tolerance setups. Although support for such setups was not an objective of this thesis, the concepts can be extended to achieve this support in the future. This is one of many possible extensions that can be applied to the current version of the logical isolation pattern; other opportunities for follow-up activities will be discussed in Section 8.2.

Chapter 7

Toolchain integration

At this point, the logical isolation pattern has been fully described—both conceptually and in terms of its reference implementation. Up until now, however, it has been assumed that an embedded software system was already available and would then be formalized using

1) a system model according to Section 3.2,
2) isolation measures according to Section 4.2, and
3) safety requirements according to Chapter 5.

In practice, it is often desirable to perform an *automated* implementation of embedded software systems. The state of the art provides toolchains that can be used to address this requirement. Three academic examples of such toolchains were listed in Section 3.1: RAMSES [142], Lingua Franca [144], and the XANDAR toolchain [1]. In the following, they will be referred to as toolchains for the *synthesis* of a software system.

To be of practical importance, the logical isolation pattern must be compatible with such synthesis toolchains. An automated generation of the above-mentioned pattern inputs is a particularly appealing goal. This chapter gives directions on how such an integration can be realized: in the context of the XANDAR toolchain, it demonstrates the auto-generation of the system model and parts of the isolation measure specification. Safety requirements must still be provided as an external input, but this constraint is not a fundamental limitation. The automatic generation of safety requirements is generally feasible, for example in the context of a HARA procedure.

7.1 Overview of the XANDAR project

As described in [1], the XANDAR project was concerned with the development of embedded software systems for autonomous and distributed applications. During the project, particular emphasis was put on representative use cases from both the automotive and the aviation domain.

In such domains, embedded software systems are currently required to meet an increasingly challenging combination of non-functional requirements. On the one hand, their features are becoming richer, and the underlying hardware/software architecture needs to reflect this trend. In a self-driving car, for example, MPSoCs can be necessary to execute AI algorithms, and the demand for vehicle-to-vehicle communication requires a seamless integration into large-scale networks. On the other hand, such systems are subject to ever-increasing safety, security, and real-time requirements. An autonomous road vehicle aiming for 'high driving automation' according to the SAE definition [161], for example, can no longer expect humans to monitor and handle system malfunctions. Therefore, it is still necessary to ensure that the system does not pose an unreasonable risk to passengers or the environment.

The outcome of XANDAR is a framework that supports developers of such systems in the fulfillment of safety, security, and real-time requirements. The framework consists of two components: (1) the XANDAR toolchain and (2) a tailored, hypervisor-based runtime architecture. Based on model-based input specifications, the toolchain generates implementations for embedded software systems. These implementations are returned as *platform binaries*, each meant to be executed on a particular execution platform. Collectively, they exhibit certain safety, security, and real-time properties. Exhibited properties are captured in the form of *Verification & Validation (V&V) reports*, which allow developers to decide whether relevant requirements are fulfilled.

7.1.1 X-by-Construction (XbC) perspective

From the toolchain user's point of view, properties documented in V&V reports are fulfilled *by construction* and can therefore—to a certain degree—be relied upon. Generally speaking, the goal of achieving non-functional properties by construction is an active area of research. Although the precise interpretation of this goal differs from case to case, approaches to address it share one key characteristic: instead of first building the system and *then* checking for the fulfillment of relevant properties, the impact that a design decision has on these properties is assessed *while* it is made.

This characteristic can also be found in a related, state-of-the-art technique for developing *functionally* correct software programs: the step-wise refinement of a formal program specification into executable code. Based on a formal framework by Hoare [162] and Dijkstra [163], it is often referred to as a Correctness-by-Construction (CbC) approach to programming [164]. Recently, this concept been shown to be applicable to *non-functional* properties: by incorporating language-based security mechanisms into program construction, desired confidentiality properties were enforced [102].

Today, the term CbC is also being used for approaches that enforce or consider *non-functional* properties of *software system* implementations. The methodology from [165], for example, achieves security by detecting and eliminating faults as early as possible after they are introduced. More recently, the authors of [166] proposed a CbC approach to parallelize hard real-time applications for off-the-shelf multicore hardware; they achieve this though a tight orchestration of all implementation phases (such as timing analysis, resource allocation, glue code generation, ...).

To emphasize that an approach does not (primarily) enforce functional correctness but that it is concerned with non-functional properties, it is also referred to as an *X-by-Construction (XbC)* technique. A possible definition of XbC is given by the authors of [167], for example:

> "X-by-Construction (XbC) is concerned with a step-wise refinement process from specification to code that automatically generates software (system) implementations that by construction satisfy specific non-functional properties [...]"

Here, the 'X' is a placeholder for the respective property. Security, reliability, and energy efficiency are three examples of what it can be replaced with.

As described in [1], the XANDAR toolchain is generally in line with this XbC definition: it is a sequential development process that operates on a model-based specification of the desired system architecture, and it generates platform binaries (i.e., compiled code) with certain safety, security, and real-time properties. It is important to emphasize, however, that the XANDAR toolchain does *not* implement a CbC strategy in the sense of [164]. Its approach is instead similar to [165] and [166]. This means that relevant implementation phases are tightly orchestrated and necessary V&V steps are performed during (or as soon as possible after) the respective phase. Enforced properties are documented in V&V reports, and these reports *support* toolchain users in proving the fulfillment of all specified requirements.

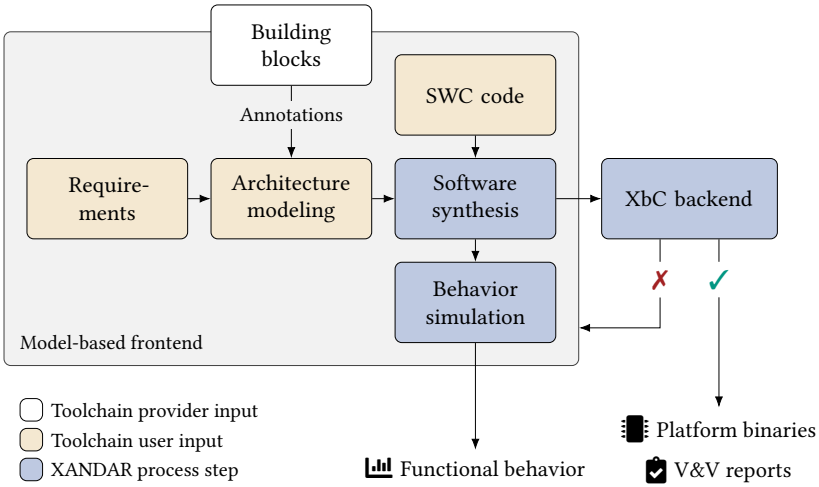


Figure 7.1: XANDAR development process with its two main stages: the model-based frontend and the XbC backend. Figure adapted from [1].

7.1.2 XANDAR development process

The toolchain of XANDAR implements the development process shown in Figure 7.1. It is built as an iterative process and can be decomposed into two main stages: the *model-based frontend* and the *XbC backend*. Toolchain users interact with graphical and textual user interfaces provided by the frontend. They use them to capture requirements, model the envisaged system architecture at different abstraction layers, and provide SWC code (e.g., in C or Rust) that shall be executed by a synthesized software system.

The integration of SWC code is referred to as *software synthesis*; this procedure creates *synthesized SWCs*, which are artifacts ready to be consumed by subsequent process steps. In early development phases, this is primarily the behavior simulation framework. This framework is able to simulate the interaction of all synthesized SWCs and capture simulation results in the form of timed execution traces. These traces allow the toolchain user to analyze (and iteratively refine) the functional behavior of all SWCs.

Afterward, the XbC backend performs a *target-aware implementation* to create the desired platform binaries. If it is unable to do so, for example because of an insufficient input model, it aborts the process and asks the toolchain user to apply suitable modifications. Figure 7.2 summarizes the internal structure of

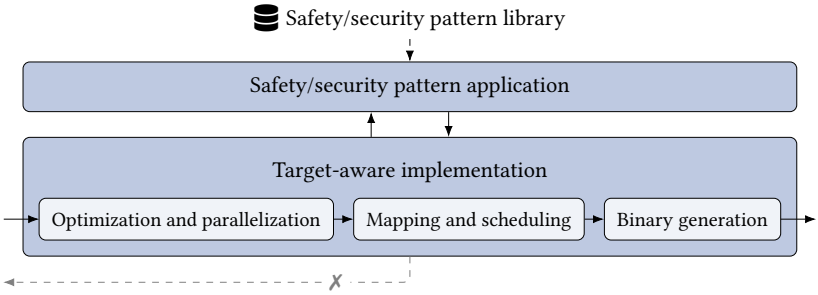


Figure 7.2: Internal structure of the XbC backend, i.e., the second main stage of the XANDAR development process. Figure adapted from [1].

the XbC backend. During all implementation steps, non-functional properties that the toolchain is able to reason about are documented in suitable V&V reports. An example of such a report is a timing trace that shows how SWCs will be executed on the hypervisor-based runtime architecture.

7.1.3 Safety/security pattern library

During the architecture modeling step (cf. Figure 7.1), toolchain users have the opportunity to annotate system entities with building blocks from an extensible library. This library contains, for example, *AI backends* that automate the integration of inference algorithms based on artificial neural networks.

Furthermore, and this is the relevant feature from the perspective of this thesis, it comprises *safety/security patterns* as repeatable procedures to implement certain safety or security measures. After such a pattern is annotated to the system model, the XbC backend performs target-aware steps to apply this repeatable procedure (cf. Figure 7.2).

In this thesis, the logical isolation pattern from Chapter 3 was integrated into the safety/security pattern library of XANDAR. After the following sections outline related contributions the author made to the XANDAR toolchain, this integration will be described in Section 7.4 below.

7.2 Behavior specification and simulation

As part of this thesis, the XANDAR toolchain was extended with an integrated behavior specification and simulation feature. It is one possible implementa-

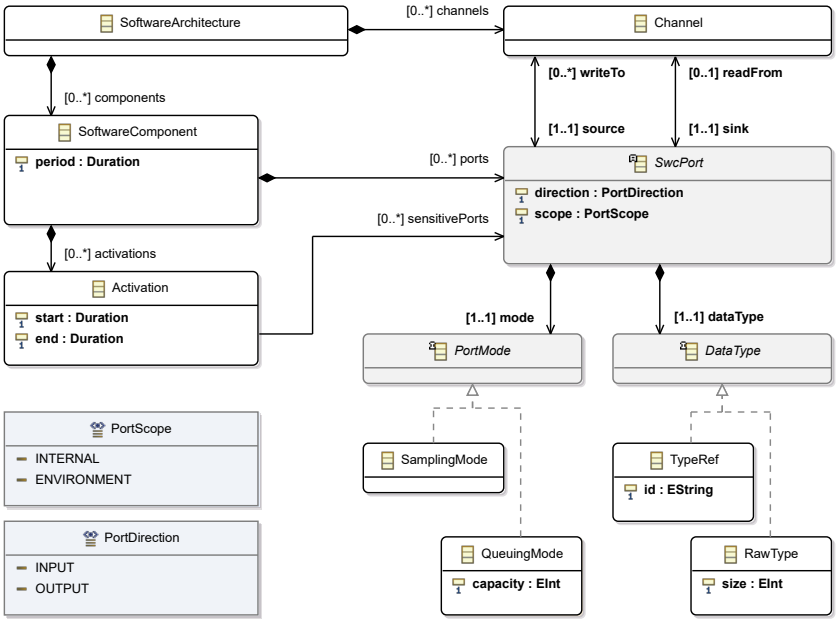


Figure 7.3: Excerpt of the XbCgen software architecture metamodel, here visualized using an Ecore diagram from the Eclipse Modeling Framework [168]. Attributes and associations with bold labels are required, i.e., have a cardinality of at least 1.

tion of the *software synthesis* and *behavior simulation* sequence shown in the model-based frontend (cf. Figure 7.1). The feature is based on the author's work in [7], implemented into a tool referred to as *XbCgen*, and expects the toolchain user to provide it with two inputs: a software architecture model and SWC code in the C programming language.

7.2.1 Software architecture metamodel

Software architecture models provided to XbCgen need to conform to the metamodel in Figure 7.3. This metamodel is intentionally aligned with software architectures according to Definition 3.21 on page 66. Strictly speaking, however, it is neither a subset not a superset of the metamodel from this definition. It was derived from Definition 3.21 by *removing* the opportunity to specify the RTE on which a given SWC is executed and *adding* properties that capture relevant behavioral aspects of SWCs.

In the updated metamodel, SWC ports are additionally characterized by their *mode* and their *data type*. The mode (`PortMode` in Figure 7.3) specifies if the port exhibits sampling or queuing behavior. It determines the manner in which a port handles repeated data events. Sampling ports will always keep the most recent value, while queuing ports buffer values until their capacity (`QueuingMode::capacity`) is exceeded. These modes are comparable to 'last-is-best' and 'queued' semantics of AUTOSAR sender-receiver interfaces [146], respectively. The data type of a port determines (1) the number of bytes that each value occupies and (2) which C data type is used to read from or write to the port. It can either be a reference to a named type (`TypeRef::id`) or a plain byte array of arbitrary size (`RawType::size`).

The second metamodel extension concerns the SWC, which is augmented with time-triggered execution semantics based on the LET paradigm [169]. Therefore, each SWC can be associated with an arbitrary number of *activations*. While a SWC is active, its activations are repeatedly executed; the repetition period is a property of the SWC itself (`SoftwareComponent::period`). The logical start and end time of each activation is specified within this repeated period (`Activation::start` and `Activation::end`, respectively). In line with the LET paradigm, input ports will be updated only at the (logical) start time of an activation, while values written to output ports will become effective only at the (logical) end time of an activation. To give developers more control over this process, updated input ports and committed output ports are explicitly specified: the model captures them as SWC ports that are sensitive to a particular activation (`Activation::sensitivePorts`). This means that the logical procedure to execute an activation is as follows:

1) At the (logical) start time of an activation, input ports sensitive to this activation are updated to reflect the value of their channel. During the (logical) execution of an activation, input ports keep their values.
2) During the (logical) execution of an activation, values written to output ports of its SWC are buffered locally. At the (logical) end time of an activation, locally buffered writes to output ports that the activation is sensitive to become effective, i.e., their channels are updated.

With this semantics in place, logical event times do not necessarily need to correspond to physical time. As long as data consistency is ensured, inter-SWC communication and the execution of activations can be realized at any time. However, there is one important exception: interactions with environment ports (`PortScope::ENVIRONMENT`) need to be synchronized to physical time. In practice, this means that the underlying runtime architecture needs to sched-

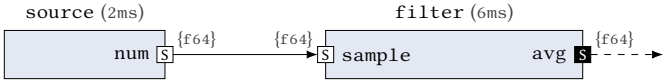


Figure 7.4: Excerpt of a software architecture supplied to XbCgen. The graphical notation is an extended version of that introduced in Section 3.2. Sampling ports are highlighted with an 'S' and labeled with their respective `TypeRef` identifier in curly braces. In addition, the repetition period of each SWC is given in parentheses.

ule the *environment interaction* part of a SWC activation at a particular time. From a hardware perspective, this is exactly the time at which the SWC needs to interact with I/O controllers to read or write data.

To facilitate a textual description of such model instances, the LIP syntax from Section 6.1 was adapted, and the resulting grammar was implemented into XbCgen. For the purposes of this work, the adapted version of the syntax is referred to as *XbCgen syntax* in the following. Files containing this syntax carry an `.xbc` extension.

▹ Example 7.1: *The software architecture from Figure 7.4 defines one activation per SWC. Textually, this software architecture is described as follows:*

Listing 7.1: Software architecture in XbCgen syntax (`system.xbc`)

```
swc source(2ms) {
  out num = f64;
  activation run(0..1ms) = num;
}

swc filter(6ms) {
  in sample = f64;
  out ~avg = f64;
  activation run(2..4ms) = sample, avg;
}

channel source.num -> filter.sample;
```

The data type of all specified ports is `f64`*, which describes double-precision floating point numbers according to IEEE 754. Values of this type occupy 64 bit of memory, and their C equivalent are* `double` *values. All ports (*`source.num`*,* `filter.sample`*, and* `filter.avg`*) are declared as sensitive to the* `run` *activation of their SWC. This leads to the logical timing relations visualized in Figure 7.5.*

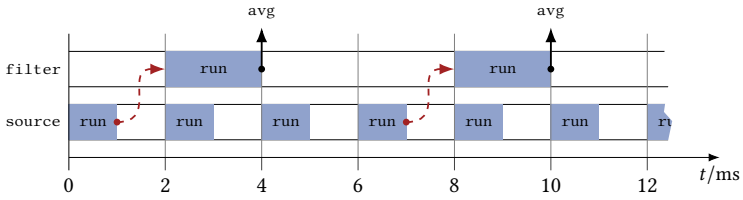


Figure 7.5: Logical activation and interaction times of the two sample SWCs. Dashed arrows highlight messages that are read by `filter`. Solid arrows highlight times at which writes to `avg` become visible to the physical environment.

The activation of `source` *terminates at logical times of 1 ms, 3 ms, and so on. These are the times at which new* `num` *values are written to the outgoing channel. At the* `sample` *port of* `filter`*, these values are received only at logical times of 2 ms, 8 ms, et cetera. This means that* `num` *updates produced at 3 ms and 5 ms, for instance, are not sampled at all. Logically, updates to the* `avg` *output of* `filter` *become effective at 4 ms, 10 ms, and so on. Since this port has environment scope, these events need to be synchronized to physical time.*

Semantically, the SWCs in this example perform the following tasks: `source` *generates a sequence of normally distributed random numbers with mean* $\mu = 3$ *and variance* $\sigma^2 = 1$*;* `filter` *samples every third number from this sequence, calculates the moving average, and outputs this average to the environment. In the real-world system, this means that a human-readable string representation of the* `f64` *value shall be written to a UART interface.*

▹ Remark 7.1: *The metamodel from this section is static in the sense that properties of model instances cannot change during runtime. This applies to SWC periods, activation timing, or the size of port values, for example. This is a deliberate limitation to simplify the mapping to XANDAR's primary runtime architecture, which maps SWCs to partitions of a type-1 hypervisor [1]. A generalization to more flexible programming models is generally feasible. This can for example be achieved by migrating the current metamodel to LF, which is a generalization of LET [170] and therefore able to capture the current model semantics.*

7.2.2 Software synthesis procedure

Based on a software architecture model, such as the one in Listing 7.1, the software synthesis feature of XbCgen generates a model-specific C interface that allows custom SWC code to interact with the underlying runtime archi-

tecture. This interface is event-driven in the sense the runtime architecture will repeatedly distribute *events* to running SWCs. Each event is directed at one particular SWC and falls into one of four categories:

1) *Activation events* communicate that the execution of a particular SWC activation is due, and that input ports sensitive to it have been updated.
2) *Environment read events* mean that a particular environment input shall be populated with values from the respective I/O controller.
3) *Environment write events* mean that the value of a particular environment output shall be written to its respective I/O controller.
4) *Exit events* request the SWC to terminate its operation.

▹ Example 7.1 (continued): *The following C code implements the* `source` *application, i.e., it handles each* `run` *activation by generating a normally distributed random number and writes it to the* `num` *output port:*

Listing 7.2: SWC code for the number generator (`source.c`)

```c
#include "adapter.h"
#include "box_muller.h"

int main() {
  xbc_init();
  for (int e; (e = xbc_next_event()) >= 0; xbc_commit()) {
    if (e == XBC_ACTIVATION_RUN) {
      double num = box_muller(3, 1);
      xbc_write_f64_port(xbc_ports.num, num);
    }
  }

  return 0;
}
```

Identifiers starting with '`xbc_`*' or '*`XBC_`*' are provided by the automatically generated C interface from* `adapter.h`*. The function* `xbc_init`*, for example, establishes a connection to the underlying runtime architecture and returns when the runtime architecture requests the SWC to start handling events. A dedicated C struct,* `xbc_ports`*, contains lightweight descriptors that uniquely identify each port of the respective SWC. In this case,* `xbc_ports.num` *is its only element. Using the* `xbc_write_f64_port` *function, it is possible to write a* `double` *value*

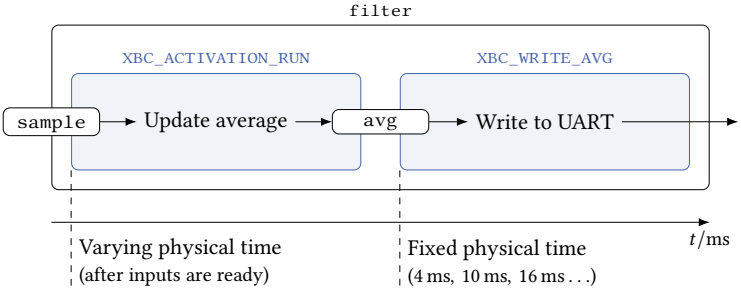


Figure 7.6: Trigger times and port interactions of the two `filter` events. While the activation itself (XBC_ACTIVATION_RUN) is triggered as soon as possible, the write event (XBC_WRITE_AVG) is synchronized to physical time.

to this output port. In the code above, this is done for each `run` *activation event, i.e., whenever* `xbc_next_event()` *returns* XBC_ACTIVATION_RUN.

As described above, the physical trigger time of an activation event does not necessarily need to conform to its logical trigger time. This is why the environment-related events exist: each such event relates to a particular environment port, and the runtime architecture issues this event when the environment interaction is actually expected to take place.

▹ Example 7.1 (continued): *The two main events issued to* `filter` *are visualized in Figure 7.6. In response to* XBC_ACTIVATION_RUN*, the moving average is updated, and its new value is internally written to* `avg`*. Actual writes to standard output are postponed until the next* XBC_WRITE_AVG *is issued:*

Listing 7.3: SWC code excerpt of the environment writer (`filter.c`)

```
if (e == XBC_ACTIVATION_RUN) {
  // Update moving average internally...
  // Write updated value to the 'avg' port...
} else if (e == XBC_WRITE_AVG) {
  printf("avg=%.3f\n", xbc_read_f64_port(xbc_ports.avg));
}
```

SWC code executed in response to activation triggers needs to be independent of the underlying runtime architecture. In fact, the actual runtime architecture used to execute a SWC might not even be defined when this

code is composed. SWC code to handle environment-related events, however, will generally depend on the underlying runtime architecture. While simple operations (such as writes to a UART controller) might be abstracted by the programming language itself, hardware-specific operations (such as CAN communication) will need to make use of suitable drivers.

7.2.3 Behavior simulation framework

The distinction between activation-related and environment-related events facilitates the timing-aware simulation of described software behavior. During such simulations, the input-output behavior of all activations can be traced by (1) sending repeated activation triggers to each SWC and (2) keeping track of all port values. Environment-related events are not dispatched, which makes it possible to execute the simulation on any host.

As part of this thesis, XbCgen was extended with a feature that automates the execution of such simulations. Therefore, it compiles each SWC for the host on which the *simulation* shall be executed. In the current implementation, this host is a Linux distribution executed on an x86 architecture. During this process, the runtime architecture interface of each SWC (i.e., `xbc_init`, `xbc_next_event`, ...) is implemented in such a way that it uses Unix domain sockets to communicate with a centralized simulation engine.

This simulation engine orchestrates the temporal execution of the software architecture, implements all specified channels, and emulates the environment (connected to environment inputs and environment outputs). It was implemented using the Ptolemy II framework [171]. XbCgen automatically synthesizes a discrete-event model in Ptolemy II, spawns a user-space Linux process for each SWC, and embeds each SWC into the discrete-event model as a Ptolemy II actor. This integration is further described in [7].

7.3 Target-aware implementation

Another—and particularly important—task of XbCgen is to implement the software architecture (including all SWCs) on the envisaged target hardware. This implementation takes place in the XbC backend (cf. Figure 7.2) and is a semi-automated process. The achieved automation degree depends on the respective *implementation strategy* that is applied to a particular design.

XbCgen supports different implementation strategies, and each strategy defines precisely how the XbC backend handles a provided software archi-

tecture. The strategy determines, for instance, the physical target hardware, how SWCs are mapped and scheduled, and how the application of a particular safety or security pattern influences this process.

Which implementation strategy to apply needs to be explicitly specified by the toolchain user; in XbCgen syntax, this is achieved by introducing an `impl` statement. As part of this thesis, two implementation strategies have been developed: (1) a target-agnostic one for the deployment to Linux user space and (2) a more specialized strategy for the deployment to a Linux distribution running on the Cortex-A53 of an i.MX 8M.

7.3.1 Deployment to Linux user space

The first implementation strategy targets a generic Linux distribution, i.e., one whose target hardware remains unknown to XbCgen itself. To apply it, the following `impl` statement needs to be added to the textual XbCgen input:

Listing 7.4: Strategy selection in XbCgen syntax (`system.xbc`)

```
impl(linux);
```

If this implementation strategy is applied to a software architecture with $|S|$ SWCs, the envisaged deployment consists of $|S| + 1$ user-space processes: one process implementing the behavior of each SWC, and a centralized orchestrator that manages the execution and communication of all SWCs.

The orchestration process (`orchestrator`) is a static component written in Go 1.22. To obtain relevant knowledge about the underlying software architecture, it expects to be provided with an *orchestrator configuration*, which is a binary file generated by XbCgen. To populate this file with required parameters, XbCgen reads them from XbCgen syntax (i.e., an `.xbc` file) and translates them into the binary format. The application binaries representing each SWC are also generated by XbCgen. This process is largely automated, but it requires the toolchain user to provide it with the desired C/C++ toolchain for generating Executable and Linkable Format (ELF) files. This is necessary because XbCgen lacks knowledge about the target hardware.

▹ Remark 7.2: *To simplify the C/C++ toolchain specification, the implementation strategy will automatically create a build system based on the* `make` *utility (according to the POSIX.2 [172] specification). If this build system is invoked from the envisaged Linux distribution, the native C/C++ toolchain can be used. If it*

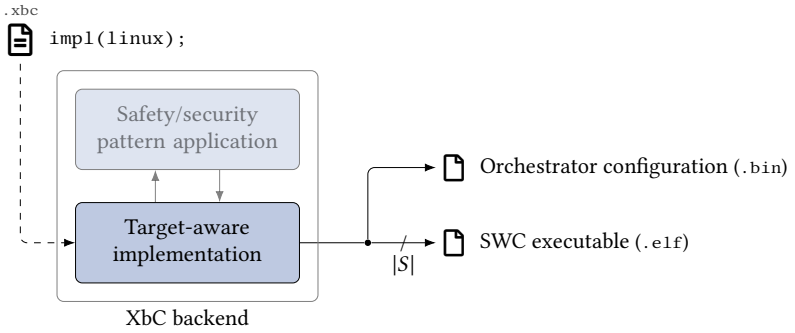


Figure 7.7: Overview of the Linux implementation strategy. With this strategy, the target-aware implementation step creates a global orchestrator configuration and one ELF file for each SWC of the software architecture.

is invoked from a host machine, a suitable cross-compiler toolchain needs to be supplied. In any case, the build process creates the $|S|$ required ELF files.

▹ Remark 7.3: *The orchestrator itself does also need to be compiled for the envisaged target hardware. Compared to SWC binaries, however, the orchestrator is a static component. Thanks to the orchestrator configuration mechanism, it does not depend on an XbCgen input file or provided SWC code. Therefore, this step can be handled separately and is not part of the XbC backend.*

A visual representation of the output artifacts that XbCgen creates for the `linux` strategy is shown in Figure 7.7. To deploy a software architecture, the generated $|S|+1$ output artifacts need to be supplied to the orchestrator. Doing so causes the orchestrator to process the configuration file, spawn each SWC binary, and connect to these binaries via Inter-Process Communication (IPC). Finally, the orchestrator enters the *system execution* phase, whose process composition is visualized in Figure 7.8.

During system execution, the orchestrator keeps track of logical time, derives SWC events that need to be triggered, and communicates with SWC binaries to execute them. This includes the implementation of inter-SWC channels and, if applicable, synchronization with physical time.

For design and debugging purposes, the orchestrator is able to emit different messages via UART. For example, it is able to capture the standard output stream of each SWC and replicate it as its own output. Whenever it needs to wait for physical and logical time to align, it is further able to output the (physical) slack for which it stalls the distribution of SWC events.

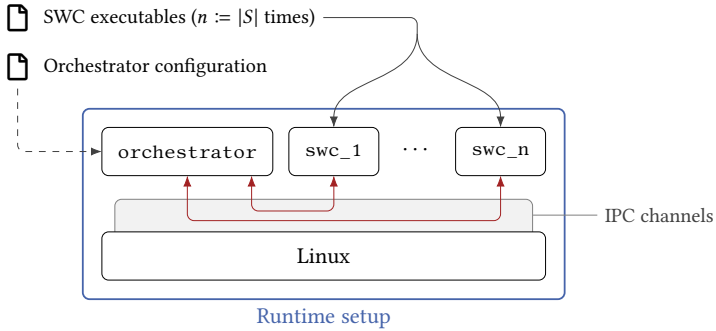


Figure 7.8: Linux user-space deployment generated by XbCgen. SWCs connect to the centralized orchestration process (`orchestrator`) via IPC channels, and the orchestration process manages all inter-SWC communication.

▹ Example 7.2: *For demonstration purposes, this implementation strategy shall now be applied to the software architecture from Example 7.1. To do so, the previous XbCgen syntax is extended with an* `impl(linux)` *statement. The generated build system is invoked using the native C/C++ toolchain of an x86-64 machine running Fedora 39, and a Go toolchain for this system is used to build a compatible version of the orchestrator. By default, the build system writes the* $|S| + 1 = 3$ *output artifacts to a dedicated* `bin` *directory.*

The orchestrator expects exactly this directory to be provided as its working directory. A sample invocation leads to the following console output:

Listing 7.5: Truncated output trace of a system execution

```
$ orchestrator -working-dir bin -print-outputs
filter> avg=2.070
filter> avg=2.704
[...]

filter> avg=3.060
filter> avg=3.066
[...]
```

This trimmed output trace shows four results generated by the `filter` *binary: two from the very beginning of the execution and two from a time when the window of the moving average filter is fully populated.*

Invoking the orchestrator with the following command-line argument triggers a second system execution, but it now causes the program to output the time synchronization slack whenever the `avg` *output is to be written:*

Listing 7.6: Truncated slack time trace for a system execution

```
$ orchestrator -working-dir bin -print-slack
slack(4ms)=3.571ms
slack(10ms)=4.260ms
slack(16ms)=3.746ms
slack(22ms)=4.257ms
[...]
```

A negative slack value would mean that a scheduled environment interaction cannot take place as planned. In this case, however, all reported slack times are positive. This means that the execution of SWC activations was always completed before their outputs were needed for an environment interaction.

7.3.2 Deployment to Linux on the i.MX 8M

The second implementation strategy developed in this work is a more specialized version of that from Section 7.3.1. As before, it targets the user space of a Linux distribution, spawns every SWC as a dedicated process, and employs a centralized orchestrator to execute and connect SWC processes. In this implementation strategy, however, the underlying CPU is known to be the Cortex-A53 of an i.MX 8M platform. To make use of the strategy, the `impl` statement needs to be adapted accordingly:

Listing 7.7: Strategy selection in XbCgen syntax (`system.xbc`)

```
impl(imx8m_linux);
```

With this statement in place, the target hardware is known to XbCgen, but the XbC backend still requires a limited degree of user interaction. This time, the toolchain user is expected to provide a compatible Yocto setup building the Linux distribution that is envisaged for the Cortex-A53 processor. The output artifact generated by this setup is a complete SD card image (generated by the Yocto setup and finally provided as a `.wic` file). As shown in Figure 7.9, this SD card image contains all components that are necessary to boot and

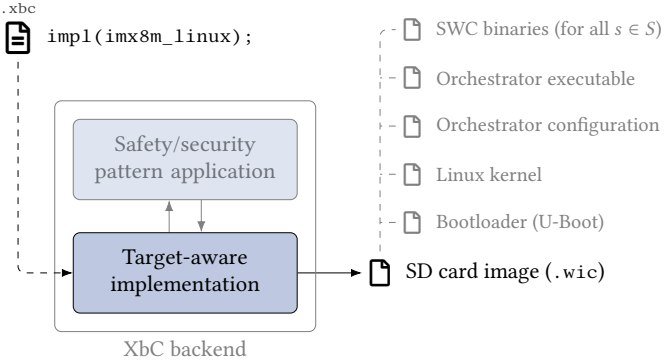


Figure 7.9: Overview of the i.MX 8M implementation strategy. The resulting SD card image is built using the Yocto setup supplied by the toolchain user.

execute the Linux-based system, including a bootloader (U-Boot), a Linux kernel, and a root file system containing both the orchestrator and the $|S| + 1$ outputs of the generic Linux implementation strategy.

As part of this work, the implementation strategy was used and tested with the official Yocto setup by NXP [173]. Leveraging community and release layers for the i.MX 8M, the `core-image-minimal` image of Poky was adapted and built. The final image boots from DDR memory and does *not* mount an SD card partition for persistent storage.

In general, implementation strategies are able to expose certain parameters, which may then be set using XbCgen syntax. This implementation strategy uses this feature to make the UART controller used to implement the Linux console configurable. The following snippet, for instance, configures the Poky distribution built by Yocto to use UART1 of the underlying platform:

Listing 7.8: UART configuration in XbCgen syntax (`.xbc`)

```
impl(imx8m_linux) {
    uart_instance = 1;
}
```

With this parameter in place, XbCgen extends the Yocto setup with a custom recipe to apply the necessary configuration, most importantly by modifying the command-line arguments that U-Boot passes to the Linux kernel.

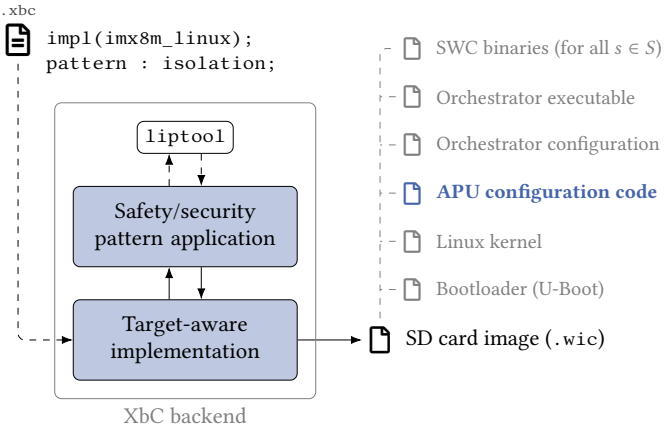


Figure 7.10: Pattern support in the i.MX 8M implementation strategy. The logical isolation pattern is delegated to `liptool`, and APU configuration code to be executed on the Cortex-M4 processor becomes part of the SD card image.

7.4 Logical isolation pattern

To close this chapter, this section describes and demonstrates a strategy to invoke the logical isolation pattern as part of the XbC backend.

7.4.1 Pattern invocation procedure

Since the logical isolation pattern is a hardware-aware concept, it can only be applied if a hardware-aware implementation strategy is used. Therefore, the final contribution of this thesis is the integration of the pattern into the `imx8m_linux` strategy from Section 7.3.2.

This integration is summarized in Figure 7.10 and was successfully implemented as part of XbCgen. As shown in the figure, the XbC backend reacts to the *simultaneous* specification of an `imx8m_linux` strategy and the logical isolation pattern by adding another component to the SD card image: APU configuration code generated according to Section 4.2.2. This code is generated for the Cortex-M4 processor; it waits for the Linux to boot on the Cortex-A53 cores, configures the RDC of the platform, and instructs the Linux distribution to start executing an orchestration process. Communication between the

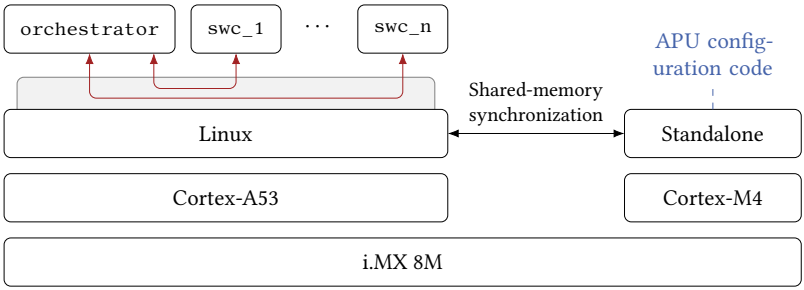


Figure 7.11: APU-protected i.MX 8M deployment generated by XbCgen. Before the SWCs are spawned as individual Linux processes, the APU configuration code is executed by a standalone OS mapped to the Cortex-M4 processor.

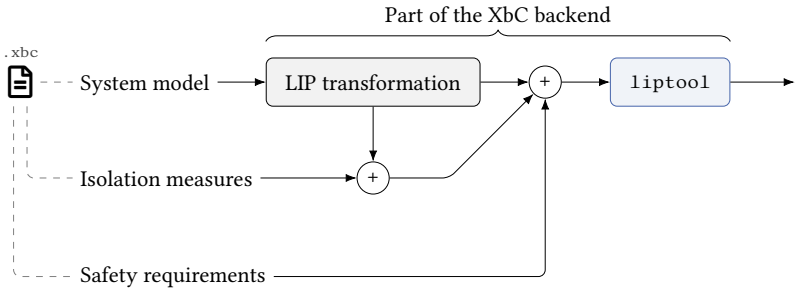


Figure 7.12: Strategy to delegate XbCgen inputs to `liptool` from Section 6.2. Using knowledge about performed deployment steps, the system model is translated into a LIP representation, and automatically applied isolation measures are reflected.

two CPUs is performed via a shared memory region in on-chip RAM. This leads to a deployment as shown in Figure 7.11.

The `liptool` delegation, which generates the core part of the APU configuration code, is realized as shown in Figure 7.12. Here, the specified system model in XbCgen syntax, which consists of the software architecture and a deployment annotation, is supplied to the *LIP transformation* procedure. During this procedure and under consideration of the respective implementation strategy, the system model is translated into an equivalent LIP system model according to Section 3.2. Furthermore, isolation measures from the pattern annotation are automatically extended using knowledge about the implementa-

tion strategy. In case of the i.MX 8M strategy, a `gen_apu` statement is inferred. In general, it is also conceivable to infer `prot_rte!` statements.

▹ Example 7.3: *To illustrate how the LIP transformation works in practice, the software architecture from Example 7.1 is extended as follows:*

Listing 7.9: Extension of previous XbCgen syntax (`system.xbc`)

```
pattern : isolation;
impl(imx8m_linux) {
  uart_instance = 1;
}
```

Provided with this extended input, XbCgen infers a system model according to Section 3.2 and extends the pattern annotation with a `gen_apu` *statement. The result is equivalent to the following LIP formulation:*

Listing 7.10: Transformed input in LIP representation (`system.lip`)

```
swc source : linux { out num; }
swc filter : linux { in sample, out ~avg; }
channel source.num -> filter.sample;

platform target : imx8m;

rte linux : target.a53 {
  alloc 0x40000000 - 0xFFFFFFFF, uart1;
}

rte standalone : target.m4 {
  alloc 0x7E0000 - 0x81FFFF;
}

path {linux, standalone} = 0x900000 - 0x90007F;

pattern : isolation {
  gen_apu target;
}
```

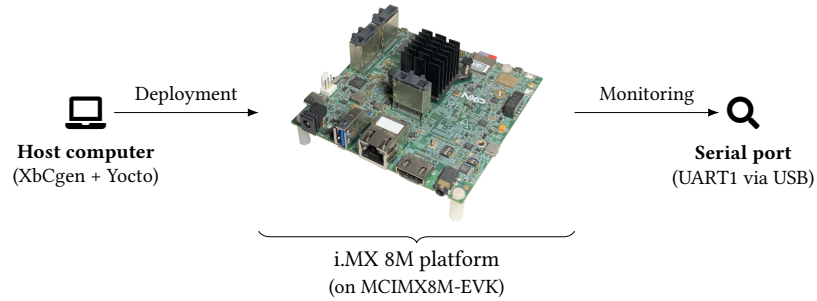


Figure 7.13: Physical hardware setup used to validate the i.MX 8M implementation strategy. Platform binaries generated by XbCgen and Yocto were deployed to the hardware, and the correct system operation was validated.

Note that this representation was manually created to describe the input that is provided to the wrapped `liptool` *instance. In reality, it exists only in the application memory of XbCgen, not as a file in LIP syntax.*

During the LIP transformation process, safety requirements are kept unchanged. This means that while populating a file in XbCgen syntax, the user has full access to the safety assessment framework from Chapter 5. If an integrity lattice is specified and integrity levels are assigned to model entities, for example, the safety assessment procedure is automatically applied to the synthesized model. XbCgen can also be instructed to execute the `liptool` tasks described in Section 6.2.1. Using this feature, it is for instance possible to trigger an automatic generation of CF graphs.

▹ Remark 7.4: *It should be emphasized that in order to use the features described above, the toolchain user is not required to specify either the runtime or the hardware architecture. Both of them are inferred from knowledge about the employed implementation strategy.*

7.4.2 Practical validation

To complete the prototypical toolchain integration of the logical isolation pattern, a practical validation procedure was performed. Therefore, the extended input model from Example 7.3 was used to generate an SD card image for the i.MX 8M platform, and this image was deployed to an MCIMX8M-EVK evaluation board by NXP (cf. Figure 7.13). By interacting with the Linux con-

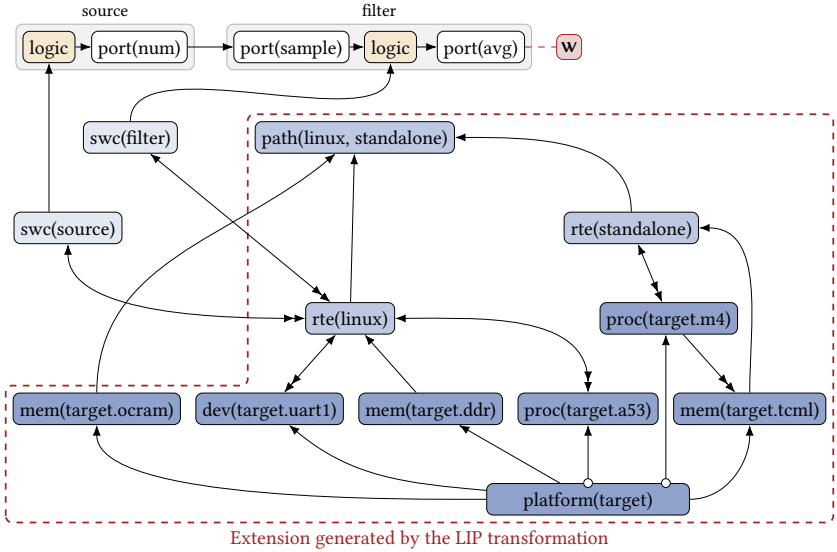


Figure 7.14: CF graph generated for the APU-protected i.MX 8M deployment. System elements that result from the automatic LIP transformation are wrapped in a dashed polygon, which was manually added to the auto-generated visualization.

sole exposed via the UART1 controller, it was ensured that (1) the functional behavior of the system was as expected and (2) the RDC protection (enforced by auto-generated APU configuration code) was operating correctly.

Finally, using the `liptool` integration of XbCgen, a CF graph for the deployed system was generated. An annotated version of this CF graph is shown in Figure 7.14. It captures the fact that the correct functionality of either SWC depends on the correct functionality of the DDR module, for instance. At the same time, it shows that the platform itself remains unaffected by possible failures of the Linux distribution.

7.4.3 Closing remarks

This concludes the description of the performed toolchain integration. Using the XANDAR toolchain as an illustrative example, it was shown that a seamless interaction of the logical isolation pattern and an automated toolchain for software system synthesis is practically feasible.

Compared with a standalone application of the pattern, the integrated variant reduces the number of inputs that need to be provided by the toolchain user. This can reduce both the development effort and the likelihood of inconsistencies between a system model and its implementation.

At the same time, a standalone application of the logical isolation pattern provides the user with more flexibility. In the toolchain-driven variant, the exploitable design space is determined by the respective implementation strategy. The strategy-based parameterization concept, which was used to allocate a particular UART controller in Example 7.2, is one possible approach to add some degree of flexibility to implementation strategies and still provide a usable abstraction of the logical isolation pattern.

Chapter 8

Conclusion

APUs, which are part of the shared on-chip interconnect of modern MPSoCs, play a pivotal role in the ongoing consolidation of embedded computing architectures. Semiconductor vendors are progressively integrating them into commercially available devices, positioning them as key mechanisms to achieve spatial isolation on MPSoCs. Despite this steadily increasing availability, however, a structured methodology for the proper configuration of APUs has not been the subject of active research.

Based on the cascading failure concept from ISO 26262 [16], this work introduced the *logical isolation* abstraction. Logical isolation captures the impact that APUs and other isolation measures have on CF potential in embedded software systems. This abstraction was then used to define a *pattern* for

1) the configuration of APUs in a network of MPSoCs,
2) reasoning about the remaining CF potential, and
3) evaluating functional safety properties.

This pattern is one of the first approaches that perceive and solve APU configuration as a complex and cross-layered decision problem.

The results of this thesis comprise the pattern definition, formal models for the inputs required by the pattern, a prototypical implementation (including platform models for two MPSoCs), and practical case studies that demonstrate the pattern's applicability in real-world scenarios. They further comprise approaches that complement the decision capability of the pattern. The presented ILP formulation, for example, determines an optimum composition of APU configurations that in combination lead to the fulfillment of given

safety requirements. These outcomes demonstrate that automatically configuring APUs and reasoning about the functional safety impact of generated configurations is feasible, which is a remarkable result.

In addition to this conceptual finding, the obtained results offer practical potential—both in their current version and as a starting point for further work. The following paragraphs give an overview of these perspectives.

8.1 Application potential

To make full use of its capabilities, the logical isolation pattern can be applied as documented in Chapter 3, i.e., by creating a full (three-layered) system model, generating APU configurations, and using the introduced analysis features to reason about CFs and their safety impact. The reference implementation from Chapter 6 is readily available to do so. The central car server example, which appeared repeatedly in this thesis, demonstrated how such an application is able to support the ongoing process of E/E architecture centralization [21]. While the integrity lattice that was finally applied to this example defined only two levels (`low` and `high`), it can easily be extended to a finer granularity, e.g., according to the Automotive Safety Integrity Level (ASIL) concept from [16]. Non-comparable integrity levels (e.g., `left` and `right`) are also conceivable and facilitate a strict *partitioning* of system elements.

The pattern is also integrated into the XANDAR toolchain. This means that it is automatically applied as part of the XANDAR development process (cf. [1] and Chapter 7). Since the supported implementation strategy maps all SWCs to only one (multicore) processor, APU configurations are currently not required for partitioning purposes. Nevertheless, prohibitive APU configurations contribute to both the detection and the confinement of failures. In addition, analysis features of the logical isolation pattern provide active support in verifying the absence of unreasonable safety risk.

The proposed ILP formulation (cf. Section 5.4) is available to optimize certain figures of merit, such as the number of APU configurations that are required to meet given safety requirements. Currently, this part of the methodology is not fully automated; it requires a certain degree of user interaction. Nevertheless, the ILP framework was used successfully as part of the second case study (cf. Section 6.4.1), and numerous other application scenarios come to mind. It can, for instance, be adapted in such a way that it performs a safety-aware allocation of SWCs to RTEs.

In addition to these primary use cases, the presented results can also be applied in a partial or more specialized manner:

No generation of APU configurations As mentioned in Chapter 3, it is feasible to apply *only* the static analysis capabilities of the pattern, i.e., without issuing any APU configuration request. In a network of execution platforms without APUs, for example, it can still be useful to have the safety assessment procedure determine the absence of unreasonable risk.

No (or custom) safety assessment The safety assessment procedure from Chapter 5 is based on the premise that only environment output ports of SWCs are able to cause physical harm. In scenarios where this premise does not hold, the safety assessment procedure is not directly applicable. In this case, the pattern can still be used to auto-generate APU configurations and CF graphs. A generated CF graph can then be processed manually, e.g., using fault trees or other traditional safety assessment techniques.

Partial system model Rather than providing the pattern with a full system model, it is conceivable to specify only one layer (a hardware architecture) or two layers (a runtime architecture mapped to a hardware architecture). In both cases, CF graphs will be generated, and certain guarantees about the CF potential in the system can be extracted. Since there will be no SWCs and therefore no environment output ports in such system models, the safety assessment procedure from Chapter 5 is no longer applicable. It needs to be replaced, e.g., as described in the previous paragraph.

Execution platform library A populated execution platform library (in the sense of Definition 3.9) has value, even if it is not used to create a system model. By maintaining it, designers are able to capture and communicate knowledge about execution platforms using a formal syntax and precise semantics. Manually consulting this library can help to identify factors that limit a platform's applicability to mixed-criticality systems. The relationships formally captured as Γ_1, Γ_2, and Γ_3 are examples of such factors. Considering them can also support the *design* of novel MPSoC architectures. Section 6.3 has demonstrated how a shared clock control register for multiple peripheral devices can lead to undesired CF potential in the system. Individual clock control registers would help to eliminate this CF potential.

Application-specific refinements The goal of this thesis was to develop a highly automated approach for the proper usage of APUs, similar to how hypervisors use MMUs. This goal necessitated a trade-off between the degree of automation and the achievable flexibility. To navigate this trade-off, the proposed formalism was made *configurable* via selected interfaces for user intervention: the system model, isolation measures, and safety requirements. In practice, more flexibility might be necessary to cover particular applications. In such situations, the concepts introduced by this work can serve as a formal starting point and be refined on a case-by-case basis. For example, it is conceivable to use the system model of this work, generate a CF graph, manually remove edges from the CF graph, and then apply the safety assessment from Chapter 5. Doing so is no longer an application of the pattern, because the pattern does not allow for the manual modification of CF graphs. However, this process is still *guided* by the proposed methodology.

Security perspective In certain cases, the pattern can be used to support security processes. In the automotive domain, for example, the cybersecurity standard ISO/SAE 21434 [174] recommends assessing the impact of damage scenarios with respect to four categories: safety, financial, operational, and privacy. An infamous example of a safety-relevant damage scenario was identified by Miller and Valasek, who showed that vulnerabilities in a production vehicle could be remotely exploited and used to transmit arbitrary CAN messages [175]. This allowed the security researchers to control steering and braking functions using a wireless interface of the vehicle. The logical isolation pattern can be used to identify and eliminate such attack paths. To do so, the environment input introducing untrusted data needs to be considered faulty (in the sense of Section 3.3). CFs that propagate from this port to more critical system elements are then identified by the safety assessment procedure.

8.2 Future work

This work creates various opportunities for further research. The following paragraphs highlight particularly interesting ones.

Evaluation First and foremost, it will be beneficial to evaluate the pattern's applicability in additional contexts. From a theoretical point of view, the pattern is fully agnostic to the application domain. Illustrative examples and practical case studies, however, were so far based on requirements from the automotive

domain. An application to safety-critical systems from other domains, such as avionics or industrial automation, is a topic for future work.

Further and future MPSoC architectures It will also be insightful to extend the current execution platform library with models and code generators for additional MPSoCs. Tracking the ongoing evolution of MPSoC architectures will especially help to identify hardware capabilities that are currently not used or cannot be represented by the pattern. At the time of writing, one such feature is TrustZone protection, which is supported by APUs of many commercially available MPSoCs. However, the pattern does not make use of this feature. With future hardware generations, it is likely that APUs will become increasingly capable, and an extension of the pattern is warranted.

Pattern-specific hardware It is conceivable to design MPSoCs for particularly high compatibility with the logical isolation pattern. Specific hardware choices, such as designated clock control registers for each peripheral, can minimize the cardinality of Γ_1, Γ_2, and Γ_3. This will lead to reduced CF potential and simplify the fulfillment of given safety requirements. Today's open hardware initiatives are a fertile starting point for such work.

Automatic enforcement of isolation measures With respect to isolation measures, it would be beneficial to reduce the support for barrier declarations and put more emphasis on generation requests (cf. Section 3.1). At the time being, the pattern is able to generate APU configurations, but knowledge about process isolation or application-level barriers can only be declared. This was a deliberate decision, because this knowledge is usually managed by other entities (such as a hypervisor) or provided by external processes (such as a formal analysis of an algorithm). Since asking the user to transfer this knowledge leaves a residual risk of human error, approaches to automate this process would strengthen the pattern. Chapter 7 mentioned how process isolation can generally be inferred from a particular implementation strategy, but such a feature has not been implemented. Knowledge about application-level barriers might be inferable using language-based IFT concepts, but this topic was beyond the scope of this work. Therefore, the automatic enforcement of isolation measures remains a direction for future research.

System model extensions With respect to the system model from Section 3.2, several possible extensions come to mind. The extension process

involves a crucial trade-off, however. With each additional capability introduced into the system model, the CF determination framework needs to be adapted. An example is the support for overlapping memory regions: the current system model allows every memory region to have up to one owner (e.g., a peripheral device, an RTE, or a local path); in CF graphs, it is therefore not necessary to represent memory regions as dedicated vertices. Relaxing such restrictions could be beneficial, but may not be necessary for practical use cases and would increase the complexity of the CF determination framework. The proposed system model was shown to be applicable in practical case studies, and it still resulted in CF graphs that could easily be visualized to the reader. Nevertheless, the controlled relaxation of current restrictions is an interesting research question for future work.

Dynamic usage of APUs We close this thesis by discussing the idea to configure APUs dynamically, i.e., under consideration of time-dependent system states. On commercially available MPSoCs, doing so is technically possible: bus masters with sufficient access permissions are capable of rewriting APU configuration registers an arbitrary number of times. However, this practice is often discouraged by semiconductor vendors: for safe and secure systems using the Zynq UltraScale+ MPSoC, the vendor recommends to lock at least the XMPU [24]. In addition, a reconfiguration of APUs would introduce novel perspectives that are currently not reflected by this work: time-dependent system models, time-dependent CF potential, and therefore time-dependent safety properties. Arguably, these factors turn the dynamic configuration idea into the most ambitious research direction listed here. Nevertheless, it can be seen as the logical continuation of this work, and its pattern-based solution would provide users of modern MPSoCs with additional options to design mixed-criticality systems of the future.

Appendix

A.1 Total unimodularity of ILP constraints

This section is concerned with a matrix $A \in \{-1, 0, 1\}^{m \times n}$ representing the $\boldsymbol{\sigma}$ coefficients from the following system of inequalities (cf. Section 5.4.1):

$$\forall \langle v_j, v_k \rangle \in E\colon \quad \sigma_k - \sigma_j \leq 0, \tag{5.1}$$

$$\forall v_j \in V_s\colon \quad \sigma_j \leq \mathrm{int}'(v_j), \tag{5.2}$$

$$\forall v_j \in V_w\colon \quad -\sigma_j \leq -\mathrm{ireq}'(v_j). \tag{5.3}$$

Section 5.4.1 claims that A is a totally unimodular matrix. This means that the determinant of every square submatrix of A must be 0, 1, or -1. To show that every $\ell \times \ell$ submatrix Q of A has determinant 0 or ± 1, we argue based on the structure that is used to prove Lemma 8.2.5 from [52, p. 146].

▹ Proof: The proof is made by induction. For $\ell = 1$, the submatrix Q contains a single coefficient of either 0 or ± 1. Therefore, $\det(Q)$ is either 0 or ± 1.

For $\ell > 1$, the submatrix Q consists of rows that (by construction) have either no, one, or two non-zero entries. If there is a row with only zeros, then $\det(Q) = 0$. If there is a row i with exactly one non-zero entry, and if this entry is located in column j, the Laplace expansion along row i can be applied. This expansion is based on Q_{ij}, which is the $(\ell-1) \times (\ell-1)$ submatrix obtained by removing row i and column j from Q:

$$\det(Q) = (-1)^{i+j} \cdot q_{ij} \cdot \det(Q_{ij}).$$

Each factor of this product is either 0 or ± 1, which leads to $\det(Q) \in \{0, \pm 1\}$. Finally, if every row of Q contains exactly two non-zero entries, then these rows must originate from Equation 5.1, and the two non-zero entries of each row are +1 and -1. In this case, the sum of all columns in Q is $\mathbf{0}$, i.e., a column

vector composed of ℓ zeros. This means that the columns of Q are linearly dependent and, therefore, that $\det(Q) = 0$. □

A.2 XMPU/XPPU configuration library

This section describes a C library that was developed as part of the pattern's reference implementation. As reported in Section 6.2.4, this library encapsulates the static part of APU configuration code generated for `zynqmp` platforms. It is here described as a representative example of how the actual APU configuration process *can* be performed. Although the description is platform-specific, underlying ideas are transferrable to most MPSoCs.

In this specific case, the library is capable of configuring all XMPUs and the XPPU of a Zynq UltraScale+ MPSoC instance. A high-level description of how these APUs need to be configured is given in Section 4.2.2.2. For a detailed explanation of the expected configuration process, the reader is referred to the platform's reference manual [33] and its register reference [39].

The following code excerpts assume that the '`xil_types.h`' header provided by AMD/Xilinx is included. This header defines the data type `u32` to hold unsigned 32-bit integers. They further expect that a function

```
void write_reg(u32 addr, u32 value);
```

writing a value (`value`) to a memory-mapped register (`addr`) is available.

A.2.1 Public interface

This section documents the public interface of the library, i.e., the macros and functions that the dynamic part of APU configuration code uses or calls. First, it contains macros that identify specific bus masters:

Listing A.1: Master description macros

```
// Register value for a particular master profile:
#define MREG(id, msk) (((msk & 0x3FFUL) << 16) | (id & 0x3FFUL))
// Precomputed values (excerpt):
#define MREG_A53 (MREG(0x080UL, 0x3C0UL))
#define MREG_GEM3 (MREG(0x077UL, 0x3FFUL))
#define MREG_DAP_APB (MREG(0x062UL, 0x3FFUL))
#define MREG_CSU (MREG(0x050UL, 0x3FFUL))
```

`MREG_A53` identifies any core of the Cortex-A53, for example.

For the specific task of configuring the platform's eight XMPU instances, the library exposes the following interface:

Listing A.2: XMPU configuration interface

```
// XMPU constants:
#define XMPU_REGION_COUNT (16)
// XMPU configuration functions:
void xmpu_clear_configs(void);
void xmpu_set_ddr_region(u32 id, u32 start, u32 end, u32 mreg);
void xmpu_set_ocm_region(u32 id, u32 start, u32 end, u32 mreg);
void xmpu_set_fpd_region(u32 id, u32 start, u32 end, u32 mreg);
void xmpu_finalize_configs(void);
```

Clients are first expected to call `xmpu_clear_configs` to reset all XMPUs. They may then issue an arbitrary number of `xmpu_set...` calls to grant specific permissions. A call to `xmpu_finalize_configs` finalizes the XMPU configuration. When granting permissions, the `id` parameter specifies the XMPU region to populate; it may range from 0 to `XMPU_REGION_COUNT`−1. Using `start` and `end`, the address region to be made accessible is described; the corresponding bus master is specified via the `mreg` parameter.

Analogously, the following public interface is provided to configure the single XPPU of an execution platform instance:

Listing A.3: XPPU configuration interface

```
// XPPU constants (excerpt):
#define XPPU_MASTER_COUNT (20)
#define XPPU_APERTURE_COUNT (401)
#define XPPU_APER_UART0 (0)
#define XPPU_APER_IOU_SCNTR (37)
#define XPPU_APER_IOU_SCNTRS (38)
#define XPPU_APER_PMU_GLOBAL (216)
// XPPU configuration functions:
void xppu_clear_config(void);
void xppu_set_master_profile(u32 id, u32 mreg);
void xppu_set_permissions(u32 aper, u32 perm);
void xppu_finalize_config(void);
```

Analogous to the XMPU case, the configuration needs to be started by a call to `xppu_clear_config` and finalized by a call to `xppu_finalize_config`. In between, clients may issue an arbitrary sequence of `xppu_set...` calls to configure master profiles and grant access permissions to these master

profiles. When a master profile is configured, the `id` may range from 0 to `XPPU_MASTER_COUNT` − 1. When permissions are set, the `aper` parameter identifies the specific aperture to configure; predefined macros support clients in specifying it. The `perm` parameter is a bit field that encodes whether each of the 20 master profiles may access the aperture.

A.2.2 XMPU configuration functions

The implementation of XMPU configuration functions is based on knowledge about how XMPU register sets are structured:

Listing A.4: XMPU structure macros

```
#define XMPU_REGION_COUNT (16)
#define XMPU_REGIONS_OFFSET (0x100UL)
#define XMPU_REGION_OFFSET (0x10UL)
#define XMPU_REGION_START_OFFSET (0x0UL)
#define XMPU_REGION_END_OFFSET (0x4UL)
#define XMPU_REGION_MASTER_OFFSET (0x8UL)
#define XMPU_REGION_CONFIG_OFFSET (0xCUL)
```

Based on this, the library implements the following (internal) helper function to configure a specific region of a given XMPU:

Listing A.5: XMPU region configuration helper

```
void xmpu_set_region(u32 xmpu, u32 id, u32 start, u32 end, u32 mreg) {
  // Base address of the XMPU region:
  u32 base = xmpu + XMPU_REGIONS_OFFSET + id*XMPU_REGION_OFFSET;
  // Rxx_START[ADDR] = precomputed 'start' value:
  write_reg(base + XMPU_REGION_START_OFFSET, start);
  // Rxx_END[ADDR] = precomputed 'end' value:
  write_reg(base + XMPU_REGION_END_OFFSET, end);
  // Rxx_MASTER[MASK] = precomputed mask from 'mreg' value and
  // Rxx_MASTER[ID] = precomputed ID from 'mreg' value:
  write_reg(base + XMPU_REGION_MASTER_OFFSET, mreg);
  // RxxCONFIG[Enable] = 1 (enable region),
  // RxxCONFIG[RdAllowed] = 1 (permit read access),
  // RxxCONFIG[WrAllowed] = 1 (permit write access),
  // RxxCONFIG[RegionNS] = 1 (declare non-secure), and
  // RxxCONFIG[NSCheckType] = 0 (relaxed TrustZone checking):
  write_reg(base + XMPU_REGION_CONFIG_OFFSET, 0xFUL);
}
```

This helper is used from the functions implementing the public interface. These functions are provided with more specialized knowledge about the particular XMPU to configure. For the XMPU/FPD case, for instance:

Listing A.6: XMPU/FPD configuration macros

```
#define XMPU_FPD_BASE (0xFD5D0000UL)
#define XMPU_FPD_ADDR(x) (x >> 12)
```

With these macros, the region configuration helper is invoked as follows:

Listing A.7: XMPU/FPD region configuration

```
void xmpu_set_fpd_region(u32 id, u32 start, u32 end, u32 mreg) {
  xmpu_set_region(XMPU_FPD_BASE, id,
    XMPU_FPD_ADDR(start),
    XMPU_FPD_ADDR(end),
    mreg);
}
```

XMPU-specific macros are also used in helpers such as the following:

Listing A.8: XMPU/FPD finalization helper

```
void xmpu_finalize_fpd_config(void) {
  // CTRL[PoisonCfg] = 1 (enable address poisoning),
  // CTRL[DefWrAllowed] = 0 (disable default write access), and
  // CTRL[DefRdAllowed] = 0 (disable default read access):
  write_reg(XMPU_FPD_BASE + XMPU_CTRL_OFFSET, 0x4UL);
}
```

The public finalization function invokes all finalization helpers:

Listing A.9: XMPU finalization

```
void xmpu_finalize_configs(void) {
  xmpu_finalize_mem_config(XMPU_OCM_BASE);
  for (int i = 0; i < XMPU_DDR_INSTANCES; i++)
    xmpu_finalize_mem_config(XMPU_DDR_BASE + i*XMPU_DDR_OFFSET);
  xmpu_finalize_fpd_config();
}
```

Note how the public finalization function calls the finalization helper of all XMPUs, including those of all XMPU/DDR instances. This strategy is also applied when XMPU/DDR regions are configured: any activity related to an XMPU/DDR is automatically applied to all instances.

A.2.3 XPPU configuration functions

Compared with the XMPU configuration process, XPPU-related activities involve less logic. They mainly forward parameters to XPPU configuration registers. This forwarding is based on the following platform knowledge:

Listing A.10: XPPU structure macros

```
#define XPPU_BASE (0xFF980000UL)
#define XPPU_MASTERS_OFFSET (0x100UL)
#define XPPU_MASTER_OFFSET (0x4UL)
#define XPPU_APERTURES_OFFSET (0x1000UL)
#define XPPU_APERTURE_OFFSET (0x4UL)
```

The function to configure a master profile is implemented as follows:

Listing A.11: XPPU master configuration

```
void xppu_set_master_profile(u32 id, u32 mreg) {
  // MASTER_IDxx[MASK] = precomputed mask from 'mreg' value and
  // MASTER_IDxx[ID] = precomputed ID from 'mreg' value:
  write_reg(XPPU_BASE + XPPU_MASTERS_OFFSET +
    id*XPPU_MASTER_OFFSET, mreg);
}
```

Aperture permissions are forwarded as follows:

Listing A.12: XPPU permission configuration

```
void xppu_set_permissions(u32 aper, u32 perm) {
  // APERPERM_xxx[PERMISSION] = precomputed 'perm' value,
  // APERPERM_xxx[TRUSTZONE] = 1 (accept non-secure access), and
  // APERPERM_xxx[PARITY] = 0 (clear parity bits):
  write_reg(XPPU_BASE + XPPU_APERTURES_OFFSET +
    aper*XPPU_APERTURE_OFFSET, (0x1UL << 27) | perm);
}
```

Finalizing an XPPU configuration means to enable the XPPU protection:

Listing A.13: XPPU finalization

```
void xppu_finalize_config(void) {
  // CTRL[ENABLE] = 1 (enable XPPU protection),
  // CTRL[MID_PARITY_EN] = 0 (disable master ID parity), and
  // CTRL[APER_PARITY_EN] = 0 (disable aperture parity):
  write_reg(XPPU_BASE + XPPU_CTRL_OFFSET, 0x1UL);
}
```

Note that this specific code does not make use of a particular safety feature provided by the XPPU: the possibility to execute parity checks before configuration entries are considered and enforced. This was done to keep this appendix concise. Extending the library in such a way that parity checks are performed is easily possible.

Bibliography

First-author publications

[1] T. Dörr, F. Schade, J. Becker, G. Keramidas, N. Petrellis, V. Kelefouras, M. Mavropoulos, K. Antonopoulos, et al. XANDAR: An X-by-Construction Framework for Safety, Security, and Real-Time Behavior of Embedded Software Systems. In: *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Valencia, Spain, Mar. 2024.

[2] T. Dörr, T. Sandmann, and J. Becker. A Formal Model for the Automatic Configuration of Access Protection Units in MPSoC-Based Embedded Systems. In: *2020 23rd Euromicro Conference on Digital System Design (DSD)*, Kranj, Slovenia, Aug. 2020.

[3] T. Dörr, T. Sandmann, and J. Becker. Model-based configuration of access protection units for multicore processors in embedded systems. *Microprocessors and Microsystems*, 87, Nov. 2021.

[4] T. Dörr, T. Sandmann, H. Mohr, and J. Becker. Employing the Concept of Multilevel Security to Generate Access Protection Configurations for Automotive On-Board Networks. In: *2021 24th Euromicro Conference on Digital System Design (DSD)*, Palermo, Italy, Sept. 2021.

[5] T. Dörr, F. Schade, L. Masing, J. Becker, G. Keramidas, C. P. Antonopoulos, M. Mavropoulos, V. Kelefouras, et al. Safety by Construction: Pattern-Based Application of Safety Mechanisms in XANDAR. In: *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, Nicosia, Cyprus, July 2022.

[6] T. Dörr, F. Schade, and J. Becker. Pattern-Based Information Flow Control for Safety-Critical On-Chip Systems. In: J. Guiochet, S. Tonetta, and F. Bitsch, editors, *Computer Safety, Reliability, and Security (SAFECOMP 2023)*. Springer, Cham, 2023.

[7] T. Dörr, F. Schade, A. Ahlbrecht, W. Zaeske, L. Masing, U. Durak, and J. Becker. A Behavior Specification and Simulation Methodology for Embedded Real-Time Software. In: *2022 IEEE/ACM 26th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, Alès, France, Sept. 2022.

[8] T. Dörr, T. Sandmann, F. Schade, F. K. Bapp, and J. Becker. Leveraging the Partial Reconfiguration Capability of FPGAs for Processor-Based Fail-Operational Systems. In: C. Hochberger, B. Nelson, A. Koch, R. Woods, and P. Diniz, editors, *Applied Reconfigurable Computing (ARC 2019)*. Springer, Cham, 2019.

[9] T. Dörr, T. Sandmann, P. Friederich, A. Leitner, and J. Becker. An Approach to Cost-Efficient Fault Tolerance in Inherently Redundant Fail-Operational Systems. In: *2020 23rd Euromicro Conference on Digital System Design (DSD)*, Kranj, Slovenia, Aug. 2020.

[10] T. Dörr, T. Sandmann, P. Friederich, A. Leitner, and J. Becker. Achieving cost-efficient fail-operational behavior based on inherent redundancy at the system level. *Microprocessors and Microsystems*, 87, Nov. 2021.

Co-author publications (selection)

[11] A. Leitner, J. Becker, T. Dörr, and F. Bapp. Method for controlling an electrical drive of a motor vehicle and computer program product. Patent number WO 2019/242804 A1. Dec. 2019.

[12] L. Masing, T. Dörr, F. Schade, J. Becker, G. Keramidas, C. P. Antonopoulos, M. Mavropoulos, E. Tiganourias, et al. XANDAR: Exploiting the X-by-Construction Paradigm in Model-based Development of Safety-critical Systems. In: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Antwerp, Belgium, Mar. 2022.

[13] F. Schade, T. Dörr, and J. Becker. Hypervisor-Based Target Deployment Strategies for Time Predictability in Model-Based Development. In: *2022 IEEE 35th International System-on-Chip Conference (SOCC)*, Belfast, UK, Sept. 2022.

[14] F. Schade, T. Dörr, A. Ahlbrecht, V. Janson, U. Durak, and J. Becker. Automatic Deployment of Embedded Real-Time Software Systems to Hypervisor-Managed Platforms. In: *2023 26th Euromicro Conference on Digital System Design (DSD)*, Golem, Albania, Sept. 2023.

Further references

[15] J. C. Knight. Safety critical systems: challenges and directions. In: *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*, Orlando, Florida, USA, May 2002.

[16] ISO 26262-1:2018: Road vehicles — Functional safety — Part 1: Vocabulary, International Organization for Standardization (ISO), Dec. 2018.

[17] J. Ziegler, P. Bender, M. Schreiber, H. Lategahn, T. Strauss, C. Stiller, T. Dang, U. Franke, et al. Making Bertha Drive—An Autonomous Journey on a Historic Route. *IEEE Intelligent Transportation Systems Magazine*, 6(2), Sum. 2014.

[18] G. Cooper. The Evolution of Neural Processing for Embedded Applications. 2022. URL: https://www.synopsys.com/designware-ip/technical-bulletin/neural-processor-npx-ip.html (visited on 13 Aug. 2023).

[19] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5), Oct. 1974.

[20] S. Borkar and A. A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5), May 2011.

[21] O. Burkacky, J. Deichmann, and J. P. Stein. Automotive Software and Electronics 2030, McKinsey & Company, July 2019.

[22] V. Bandur, G. Selim, V. Pantelic, and M. Lawford. Making the Case for Centralized Automotive E/E Architectures. *IEEE Transactions on Vehicular Technology*, 70(2), Feb. 2021.

[23] ISO 26262-6:2018: Road vehicles — Functional safety — Part 6: Product development at the software level, International Organization for Standardization (ISO), Dec. 2018.

[24] S. McNeil, P. Schillinger, A. Kolarkar, E. Puillet, and U. Gertheinrich. Isolation Mechanisms in Zynq UltraScale+ MPSoCs (XAPP1320), Xilinx, Inc., July 2021.

[25] ISO 21448:2022: Road vehicles — Safety of the intended functionality, International Organization for Standardization (ISO), June 2022.

[26] I. Allende, N. Mc Guire, J. Perez, L. G. Monsalve, N. Uriarte, and R. Obermaisser. Towards Linux for the Development of Mixed-Criticality Embedded Systems Based on Multi-Core Devices. In: *2019 15th European Dependable Computing Conference (EDCC)*, Naples, Italy, Sept. 2019.

[27] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 5th edition, Sept. 2011.

[28] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, and T. Strudel. A Distributed Run-Time Environment for the Kalray MPPA-256 Integrated Manycore Processor. *Procedia Computer Science*, 18, 2013.

[29] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, et al. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro*, 36(2), Mar.–Apr. 2016.

[30] A. Kamaleldin and D. Göhringer. AGILER: An Adaptive Heterogeneous Tile-Based Many-Core Architecture for RISC-V Processors. *IEEE Access*, 10, 2022.

[31] C. Kühbacher, T. Ungerer, and S. Altmeyer. Redundant dataflow applications on clustered manycore architectures. In: *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing (SAC '22)*, 25 Apr. 2022.

[32] Y. Gao and P. Zhang. A Survey of Homogeneous and Heterogeneous System Architectures in High Performance Computing. In: *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, New York, USA, Nov. 2016.

[33] Advanced Micro Devices, Inc. Zynq UltraScale+ Device: Technical Reference Manual (UG1085), Jan. 2023.

[34] W. Wolf, A. Jerraya, and G. Martin. Multiprocessor System-on-Chip (MPSoC) Technology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10), Oct. 2008.

[35] M. Hassan. Heterogeneous MPSoCs for Mixed-Criticality Systems: Challenges and Opportunities. *IEEE Design & Test*, 35(4), Aug. 2018.

[36] Arm Ltd. AMBA AXI and ACE: Protocol Specification (IHI 0022 H), Mar. 2020.

[37] Advanced Micro Devices, Inc. AXI Interconnect: LogiCORE IP Product Guide (PG059), May 2022.

[38] F. Restuccia, M. Pagani, A. Biondi, M. Marinoni, and G. Buttazzo. Is Your Bus Arbiter Really Fair? Restoring Fairness in AXI Interconnects for FPGA SoCs. *ACM Transactions on Embedded Computing Systems*, 18, 5s, Oct. 2019.

[39] Advanced Micro Devices, Inc. Zynq UltraScale+ Devices Register Reference (UG1087), Mar. 2024.

[40] A. S. Tanenbaum. *Modern Operating Systems*. Pearson, 4th edition, 2015.

[41] N. Asokan, J.-E. Ekberg, K. Kostiainen, A. Rajan, C. Rozas, A.-R. Sadeghi, S. Schulz, and C. Wachsmann. Mobile Trusted Computing. *Proceedings of the IEEE*, 102(8), Aug. 2014.

[42] GlobalPlatform, Inc. Introduction to Trusted Execution Environments. May 2018. URL: https://globalplatform.org/resource-publication/introduction-to-trusted-execution-environments (visited on 27 Aug. 2023).

[43] S. Pinto and N. Santos. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Computing Surveys*, 51(6), Nov. 2019.

[44] Arm Ltd. ARM Security Technology: Building a Secure System using TrustZone Technology (PRD29-GENC-009492), Apr. 2009.

[45] NXP B.V. i.MX 8M Dual/8M QuadLite/8M Quad Applications Processors Reference Manual (IMX8MDQLQRM), June 2021.

[46] M. Masmano, I. Ripoll, and A. Crespo. XtratuM: a Hypervisor for Safety Critical Embedded Systems. In: *11th Real-Time Linux Workshop*, Dresden, Germany, 2009.

[47] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[48] J. A. Bondy and U. S. R. Murty. *Graph Theory*. S. Axler and K. A. Ribet, editors, volume 244 of *Graduate Texts in Mathematics*. Springer London, 2008.

[49] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 4th edition, 2022.

[50] N. Mahadev and U. Peled. *Threshold Graphs and Related Topics*, volume 56 of *Annals of Discrete Mathematics*. Elsevier, 1995.

[51] G. Grätzer. *Lattice Theory: Foundation*. Birkhäuser Basel, 2011.

[52] J. Matoušek and B. Gärtner. *Understanding and Using Linear Programming*. Universitext. Springer Berlin, Heidelberg, 2007.

[53] lp_solve v5.5. URL: https://lpsolve.sourceforge.net/5.5/ (visited on 3 May 2024).

[54] J. P. Cerrolaza, R. Obermaisser, J. Abella, F. J. Cazorla, K. Grüttner, I. Agirre, H. Ahmadian, and I. Allende. Multi-core Devices for Safety-critical Systems: A Survey. *ACM Computing Surveys*, 53(4), Aug. 2020.

[55] D. Kliem and S.-O. Voigt. Scalability evaluation of an FPGA-based multi-core architecture with hardware-enforced domain partitioning. *Microprocessors and Microsystems*, 38, 8B, Nov. 2014.

[56] G. Gracioli, R. Tabish, R. Mancuso, R. Mirosanlou, R. Pellizzoni, and M. Caccamo. Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms. In: *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Volume 133, Leibniz International Proceedings in Informatics (LIPIcs). 2019.

[57] P. Modica, A. Biondi, G. Buttazzo, and A. Patel. Supporting temporal and spatial isolation in a hypervisor for ARM multicore platforms. In: *2018 IEEE International Conference on Industrial Technology (ICIT)*, Lyon, France, Feb. 2018.

[58] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. Kvm: the Linux Virtual Machine Monitor. In: *Proceedings of the Linux Symposium, Volume One*, Ottawa, Ontario, Canada, June 2007.

[59] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer. Look Mum, no VM Exits! (Almost). In: *Proceedings of the 13th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2017)*, Duprovnik, Croatia, June 2017.

[60] J. De Bonfils Lavernelle, P.-F. Bonnefoi, B. Gonzalvo, and D. Sauveron. Assessment of spatial isolation in Jailhouse: Towards a generic approach. *Computer Networks*, 245, May 2024.

[61] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto. Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems. In: *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*. Volume 77, Open Access Series in Informatics (OASIcs). 2020.

[62] B. Sá, J. Martins, and S. E. S. Pinto. A First Look at RISC-V Virtualization from an Embedded Systems Perspective. *IEEE Transactions on Computers*, 71(9), Sept. 2022.

[63] AUTOSAR. Specification of Operating System, Nov. 2020. No. 34, CP R20-11.

[64] IEC 61508-3:2010: Functional safety of electrical/electronic/programmable electronic safety-related systems — Software requirements, International Electrotechnical Commission (IEC), Apr. 2010.

[65] J. Perez, D. Gonzalez, S. Trujillo, and T. Trapman. A Safety Concept for an IEC-61508 Compliant Fail-Safe Wind Power Mixed-Criticality System Based on Multicore and Partitioning. In: J. A. De La Puente and T. Vardanega, editors, *Ada-Europe 2015*. Springer, Cham, 2015.

[66] T. Nojiri, Y. Kondo, N. Irie, M. Ito, H. Sasaki, and H. Maejima. Domain Partitioning Technology for Embedded Multicore Processors. *IEEE Micro*, 29(6), Nov.–Dec. 2009.

[67] A. Hattendorf, A. Raabe, and A. Knoll. Shared memory protection for spatial separation in multicore architectures. In: *7th IEEE International Symposium on Industrial Embedded Systems (SIES '12)*, Karlsruhe, Germany, June 2012.

[68] J. Porquet, C. Schwarz, and A. Greiner. Multi-compartment: A new architecture for secure co-hosting on SoC. In: *2009 International Symposium on System-on-Chip*, Tampere, Finland, Oct. 2009.

[69] J. Porquet, A. Greiner, and C. Schwarz. NoC-MPU: A secure architecture for flexible co-hosting on shared memory MPSoCs. In: *2011 Design, Automation & Test in Europe*, Grenoble, France, Mar. 2011.

[70] D. M. Ancajas, K. Chakraborty, and S. Roy. Fort-NoCs: Mitigating the threat of a compromised NoC. In: *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, San Francisco, California, USA, June 2014.

[71] A. Shalaby, Y. Tavva, T. E. Carlson, and L.-S. Peh. Sentry-NoC: A Statically-Scheduled NoC for Secure SoCs. In: *2021 15th IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, Oct. 2021.

[72] B. Tan, M. Biglari-Abhari, and Z. Salcic. A system-level security approach for heterogeneous MPSoCs. In: *2016 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, Rennes, France, Oct. 2016.

[73] B. Tan, M. Biglari-Abhari, and Z. Salcic. Towards decentralized system-level security for MPSoC-based embedded applications. *Journal of Systems Architecture*, 80, Oct. 2017.

[74] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee. FlexPRET: A processor platform for mixed-criticality systems. In: *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Berlin, Germany, Apr. 2014.

[75] M. Schoeberl, W. Puffitsch, S. Hepp, B. Huber, and D. Prokesch. Patmos: a time-predictable microprocessor. *Real-Time Systems*, 54(2), Apr. 2018.

[76] E. R. Jellum, S. Lin, P. Donovan, C. Jerad, E. Wang, M. Lohstroh, E. A. Lee, and M. Schoeberl. InterPRET: a Time-predictable Multicore Processor. In: *Proceedings of Cyber-Physical Systems and Internet of Things Week 2023 (CPS-IoT Week '23)*, San Antonio, Texas, USA, May 2023.

[77] L. Ecco, S. Tobuschat, S. Saidi, and R. Ernst. A mixed critical memory controller using bank privatization and fixed priority scheduling. In: *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Chongqing, China, Aug. 2014.

[78] J. Jalle, E. Quiñones, J. Abella, L. Fossati, M. Zulianello, and F. J. Cazorla. A Dual-Criticality Memory Controller (DCmc): Proposal and Evaluation of a Space Case Study. In: *2014 IEEE Real-Time Systems Symposium (RTSS)*, Rome, Italy, Dec. 2014.

[79] F. Kluge, M. Schoeberl, and T. Ungerer. Support for the logical execution time model on a time-predictable multicore processor. *ACM SIGBED Review*, 13(4), 3 Nov. 2016.

[80] S. Reder and J. Becker. Interference-Aware Memory Allocation for Real-Time Multi-Core Systems. In: *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Sydney, Australia, Apr. 2020.

[81] C. Bradatsch, F. Kluge, and T. Ungerer. A Cross-Domain System Architecture for Embedded Hard Real-Time Many-Core Systems. In: *2013 IEEE 10th International Conference on High Performance Computing and Communications (HPCC) & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, Zhangjiajie, China, Nov. 2013.

[82] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, et al. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9), Oct. 2015.

[83] A. Larrucea, I. Martinez, V. Brocal, S. Peirò, H. Ahmadian, J. Perez, and R. Obermaisser. DREAMS: Cross-Domain Mixed-Criticality Patterns. In: *2016 4th International Workshop on Mixed Criticality Systems (WMC)*, Porto, Portugal, Nov. 2016.

[84] H. Kim and R. Rajkumar. Predictable Shared Cache Management for Multi-Core Real-Time Virtualization. *ACM Transactions on Embedded Computing Systems*, 17(1), Jan. 2018.

[85] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory Bandwidth Management for Efficient Performance Isolation in Multi-Core Platforms. *IEEE Transactions on Computers*, 65(2), Feb. 2016.

[86] A. Crespo, P. Balbastre, J. Simo, J. Coronel, D. Gracia-Perez, and P. Bonnot. Hypervisor-Based Multicore Feedback Control of Mixed-Criticality Systems. *IEEE Access*, 6, 2018.

[87] A. Saeed, D. Dasari, D. Ziegenbein, V. Rajasekaran, F. Rehm, M. Pressler, A. Hamann, D. Mueller-Gritschneder, et al. Memory Utilization-Based Dynamic Bandwidth Regulation for Temporal Isolation in Multi-Cores. In: *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Milano, Italy, May 2022.

[88] D. Costa, L. Cuomo, D. Oliveira, I. M. Savino, B. Morelli, J. Martins, F. Tronci, A. Biasci, et al. IRQ Coloring: Mitigating Interrupt-Generated Interference on ARM Multicore Platforms. In: F. Terraneo and D. Cattaneo, editors, *Fourth Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2023)*. Volume 108, Open Access Series in Informatics (OASIcs). 2023.

[89] R. Achermann, L. Humbel, D. Cock, and T. Roscoe. Formalizing Memory Accesses and Interrupts. In: H. Hermanns and P. Höfner, editors, *Models for Formal Analysis of Real Systems (MARS 2017)*. Volume 244, Electronic Proceedings in Theoretical Computer Science. Mar. 2017.

[90] devicetree.org. Devicetree Specification (v0.4), June 2023.

[91] R. Achermann, L. Humbel, D. Cock, and T. Roscoe. Physical Addressing on Real Hardware in Isabelle/HOL. In: J. Avigad and A. Mahboubi, editors, *Interactive Theorem Proving (ITP 2018)*. Springer, Cham, 2018.

[92] R. Achermann, D. Cock, R. Haecki, N. Hossle, L. Humbel, T. Roscoe, and D. Schwyn. Generating correct initial page tables from formal hardware descriptions. In: *Proceedings of the 11th Workshop on Programming Languages and Operating Systems (PLOS '21)*, Virtual Event, Germany, Oct. 2021.

[93] B. Fiedler, D. Schwyn, C. Gierczak-Galle, D. Cock, and T. Roscoe. Putting out the hardware dumpster fire. In: *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HOTOS '23)*, Providence, Rhode Island, USA, June 2023.

[94] B. Fiedler, R. Meier, J. Schult, D. Schwyn, and T. Roscoe. Specifying the de-facto OS of a production SoC. In: *Proceedings of the 1st Workshop on Kernel Isolation, Safety and Verification (KISV '23)*, Koblenz, Germany, Oct. 2023.

[95] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner. Information flow isolation in I2C and USB. In: *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, San Diego, California, USA, June 2011.

[96] W. Hu, A. Ardeshiricham, and R. Kastner. Hardware Information Flow Tracking. *ACM Computing Surveys*, 54(4), May 2022.

[97] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5), May 1976.

[98] K. Biba. Integrity Considerations For Secure Computer Systems. MTR-3153, Mitre Corporation, June 1975.

[99] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7), July 1977.

[100] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3), Jan. 1996.

[101] A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1), Jan. 2003.

[102] I. Schaefer, T. Runge, A. Knüppel, L. Cleophas, D. Kourie, and B. W. Watson. Towards Confidentiality-by-Construction. In: T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Modeling (ISoLA 2018)*. Springer, Cham, 2018.

[103] T. Runge, A. Knüppel, T. Thüm, and I. Schaefer. Lattice-Based Information Flow Control-by-Construction for Security-by-Design. In: *Proceedings of the 8th International Conference on Formal Methods in Software Engineering (FormaliSE '20)*, Seoul, Republic of Korea, Oct. 2020.

[104] T. Runge, A. Kittelmann, M. Servetto, A. Potanin, and I. Schaefer. Information Flow Control-by-Construction for an Object-Oriented Language. In: B.-H. Schlingloff and M. Chai, editors, *Software Engineering and Formal Methods (SEFM 2022)*. Springer, Cham, 2022.

[105] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, et al. seL4: From General Purpose to a Proof of Information Flow Enforcement. In: *2013 IEEE Symposium on Security and Privacy*, Berkeley, California, USA, May 2013.

[106] C. Brant, P. Shrestha, B. Mixon-Baca, K. Chen, S. Varlioglu, N. Elsayed, Y. Jin, J. Crandall, et al. Challenges and Opportunities for Practical and Effective Dynamic Information Flow Tracking. *ACM Computing Surveys*, 55(1), Jan. 2023.

[107] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In: *11th IEEE Symposium on Computers and Communications (ISCC '06)*, Cagliari, Italy, 2006.

[108] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In: *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA '07)*, London, UK, July 2007.

[109] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In: *Proceedings of the Network and Distributed System Security Symposium 2011 (NDSS)*, San Diego, California, USA, Feb. 2011.

[110] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, et al. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Transactions on Computer Systems*, 32(2), June 2014.

[111] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making Information Flow Explicit in HiStar. In: *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, USA, Nov. 2006.

[112] X. Guo, H. Zhu, Y. Jin, and X. Zhang. When Capacitors Attack: Formal Method Driven Design and Detection of Charge-Domain Trojans. In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Florence, Italy, Mar. 2019.

[113] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner. Theoretical Fundamentals of Gate Level Information Flow Tracking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(8), Aug. 2011.

[114] J. Oberg, T. Sherwood, and R. Kastner. Eliminating Timing Information Flows in a Mix-Trusted System-on-Chip. *IEEE Design & Test*, 30(2), Apr. 2013.

[115] W. Hu, J. Oberg, J. Barrientos, Dejun Mu, and R. Kastner. Expanding Gate Level Information Flow Tracking for Multilevel Security. *IEEE Embedded Systems Letters*, 5(2), June 2013.

[116] W. Hu, D. Mu, J. Oberg, B. Mao, M. Tiwari, T. Sherwood, and R. Kastner. Gate-Level Information Flow Tracking for Security Lattices. *ACM Transactions on Design Automation of Electronic Systems*, 20(1), Nov. 2014.

[117] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In: *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*, Istanbul, Turkey, Mar. 2015.

[118] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. *ACM SIGARCH Computer Architecture News*, 32(5), Dec. 2004.

[119] H. Kannan, M. Dalton, and C. Kozyrakis. Decoupling Dynamic Information Flow Tracking with a Dedicated Coprocessor. In: *2009 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, Lisbon, Portugal, June–July 2009.

[120] V. Nagarajan, H.-S. Kim, Y. Wu, and R. Gupta. Dynamic Information Flow Tracking on Multicores. In: *12th Workshop on Interaction between Compilers and Computer Architectures (INTERACT-12)*, Salt Lake City, Utah, USA, Feb. 2008.

[121] Z. Jiang, S. Dai, G. E. Suh, and Z. Zhang. High-Level Synthesis with Timing-Sensitive Information Flow Enforcement. In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, San Diego, California, USA, Nov. 2018.

[122] C. Pilato, K. Wu, S. Garg, R. Karri, and F. Regazzoni. TaintHLS: High-Level Synthesis for Dynamic Information Flow Tracking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5), May 2019.

[123] J. Porquet and S. Sethumadhavan. WHISK: An uncore architecture for Dynamic Information Flow Tracking in heterogeneous embedded SoCs. In: *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Montreal, Quebec, Canada, Sept.–Oct. 2013.

[124] L. Piccolboni, G. Di Guglielmo, and L. P. Carloni. PAGURUS: Low-Overhead Dynamic Information Flow Tracking on Loosely Coupled Accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11), Nov. 2018.

[125] L. P. Carloni. The Case for Embedded Scalable Platforms. In: *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, Austin, Texas, USA, June 2016.

[126] P. Mantovani, D. Giri, G. Di Guglielmo, L. Piccolboni, J. Zuckerman, E. G. Cota, M. Petracca, C. Pilato, et al. Agile SoC Development with Open ESP. In: *Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD '20)*, Virtual Event, USA, Nov. 2020.

[127] M. Hassan, V. Herdt, H. M. Le, D. Große, and R. Drechsler. Early SoC security validation by VP-based static information flow analysis. In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Irvine, California, USA, Nov. 2017.

[128] P. Pieper, V. Herdt, D. Grose, and R. Drechsler. Dynamic Information Flow Tracking for Embedded Binaries using SystemC-based Virtual Prototypes. In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*, San Francisco, California, USA, July 2020.

[129] T. Prosvirnova, M. Batteux, P.-A. Brameret, A. Cherfi, T. Friedlhuber, J.-M. Roussel, and A. Rauzy. The AltaRica 3.0 Project for Model-Based Safety Assessment. In: *4th IFAC Workshop on Dependable Control of Discrete Systems (DCDS '13)*, York, UK, Sept. 2013.

[130] G. Point and A. Rauzy. AltaRica: Constraint automata as a description language. *European Journal on Automation*, 33(8-9), 1999.

[131] A. Rauzy. Mode automata and their compilation into fault trees. *Reliability Engineering & System Safety*, 78(1), Oct. 2002.

[132] M. Güdemann and F. Ortmeier. A Framework for Qualitative and Quantitative Formal Model-Based Safety Analysis. In: *2010 IEEE 12th International Symposium on High-Assurance Systems Engineering (HASE)*, San Jose, California, USA, Nov. 2010.

[133] S. Getir, A. van Hoorn, L. Grunske, and M. Tichy. Co-Evolution of Software Architecture and Fault Tree models: An Explorative Case Study on a Pick and Place Factory Automation System. In: *5th International Workshop Non-functional Properties in Modeling: Analysis, Languages and Processes (NiM-ALP)*, Miami, Florida, USA, Sept. 2013.

[134] F. Mhenni, N. Nguyen, and J.-Y. Choley. SafeSysE: A Safety Analysis Integration in Systems Engineering Approach. *IEEE Systems Journal*, 12(1), Mar. 2018.

[135] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, M. Roveri, and R. Wimmer. A Model Checker for AADL. In: T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification (CAV 2010)*. Springer, Berlin, Heidelberg, 2010.

[136] M. Bozzano, H. Bruintjes, A. Cimatti, J.-P. Katoen, T. Noll, and S. Tonetta. COMPASS 3.0. In: T. Vojnar and L. Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2019)*. Springer, Cham, 2019.

[137] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. CMU/SEI-2006-TN-011, Carnegie Mellon University, Feb. 2006.

[138] SAE AS5506D: Architecture Analysis & Design Language (AADL), SAE International, Apr. 2022.

[139] J. Delange and P. Feiler. Architecture Fault Modeling with the AADL Error-Model Annex. In: *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, Verona, Italy, Aug. 2014.

[140] J. Brunel, P. Feiler, J. Hugues, B. Lewis, T. Prosvirnova, C. Seguin, and L. Wrage. Performing Safety Analyses with AADL and AltaRica. In: M. Bozzano and Y. Papadopoulos, editors, *Model-Based Safety and Assessment (IMBSA 2017)*. Springer, Cham, 2017.

[141] J. Hugues, B. Zalila, L. Pautet, and F. Kordon. From the prototype to the final embedded system using the Ocarina AADL tool suite. *ACM Transactions on Embedded Computing Systems*, 7(4), July 2008.

[142] F. Cadoret, E. Borde, S. Gardoll, and L. Pautet. Design Patterns for Rule-Based Refinement of Safety Critical Embedded Systems Models. In: *2012 IEEE 17th International Conference on Engineering of Complex Computer Systems (ICECCS)*, Paris, July 2012.

[143] F. Singhoff, J. Hugues, H. Nam Tran, G. Bardaro, D. Blouin, M. Bozzano, P. Denzler, P. Dissaux, et al. ADEPT 2022 workshop: a summary of strengths and weaknesses of the AADL ecosystem. *ACM SIGAda Ada Letters*, 43(1), June 2023.

[144] M. Lohstroh, C. Menard, S. Bateni, and E. A. Lee. Toward a Lingua Franca for Deterministic Concurrent Systems. *ACM Transactions on Embedded Computing Systems*, 20(4), July 2021.

[145] ISO 11898-1:2015: Road vehicles — Controller area network (CAN) — Part 1: Data link layer and physical signalling, International Organization for Standardization (ISO), Dec. 2015.

[146] AUTOSAR. Virtual Functional Bus, Nov. 2022. No. 56, CP R22-11.

[147] M. Lohstroh. Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems. UCB/EECS-2020-235, PhD thesis, University of California, Berkeley, Dec. 2020.

[148] T. Coe, T. Mathisen, C. Moler, and V. Pratt. Computational aspects of the Pentium affair. *IEEE Computational Science and Engineering*, 2(1), Spr. 1995.

[149] N. Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 20(11), Nov. 1977.

[150] The Go Programming Language Specification, Feb. 2024. URL: https://go.dev/ref/spec/ (visited on 13 Apr. 2024).

[151] R. Bernstein. Testing for semilattices. *ACM SIGACT News*, 18(1), Sum. 1986.

[152] J. Pohlmann. Configurable Graph Drawing Algorithms for the TikZ Graphics Description Language. Diploma thesis, Universität zu Lübeck, Oct. 2011.

[153] Advanced Micro Devices, Inc. ZCU102 Evaluation Board (UG1182), Feb. 2023.

[154] ISO 26262-10:2018: Road vehicles — Functional safety — Part 10: Guidelines on ISO 26262, International Organization for Standardization (ISO), Dec. 2018.

[155] R. Ernst. Automated Driving: The Cyber-Physical Perspective. *Computer*, 51(9), Sept. 2018.

[156] K. Becker, S. Voss, and B. Schätz. Formal analysis of feature degradation in fault-tolerant automotive systems. *Science of Computer Programming*, 154, Mar. 2018.

[157] R. Isermann, R. Schwarz, and S. Stölzl. Fault-tolerant drive-by-wire systems. *IEEE Control Systems*, 22(5), Oct. 2002.

[158] L. Sha. Using Simplicity to Control Complexity. *IEEE Software*, 18(4), July–Aug. 2001.

[159] S. Bak, D. K. Chivukula, O. Adekunle, M. Sun, M. Caccamo, and L. Sha. The System-Level Simplex Architecture for Improved Real-Time Embedded System Safety. In: *15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, San Francisco, California, USA, Apr. 2009.

[160] P. Pop, V. Izosimov, P. Eles, and Z. Peng. Design Optimization of Time- and Cost-Constrained Fault-Tolerant Embedded Systems With Checkpointing and Replication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(3), Mar. 2009.

[161] SAE J3016: Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles, SAE International, Apr. 2021.

[162] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), Oct. 1969.

[163] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8), Aug. 1975.

[164] D. G. Kourie and B. W. Watson. *The Correctness-by-Construction Approach to Programming*. Springer Berlin, Heidelberg, 2012.

[165] A. Hall and R. Chapman. Correctness by construction: developing a commercial secure system. *IEEE Software*, 19(1), Jan.–Feb. 2002.

[166] K. Didier, D. Potop-Butucaru, G. Iooss, A. Cohen, J. Souyris, P. Baufreton, and A. Graillat. Correct-by-Construction Parallelization of Hard Real-Time Avionics Applications on Off-the-Shelf Predictable Hardware. *ACM Transactions on Architecture and Code Optimization*, 16(3), Sept. 2019.

[167] M. H. ter Beek, L. Cleophas, I. Schaefer, and B. W. Watson. X-by-Construction. In: T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Modeling (ISoLA 2018)*. Springer, Cham, 2018.

[168] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, editors. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2nd edition, 2009.

[169] C. M. Kirsch and A. Sokolova. The Logical Execution Time Paradigm. In: S. Chakraborty and J. Eberspächer, editors, *Advances in Real-Time Systems*. Springer Berlin, Heidelberg, 2012.

[170] E. A. Lee and M. Lohstroh. Generalizing Logical Execution Time. In: J.-F. Raskin, K. Chatterjee, L. Doyen, and R. Majumdar, editors, *Principles of Systems Design*. Springer, Cham, 2022.

[171] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, et al. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1), Jan. 2003.

[172] ANSI/IEEE 1003.2-1993 — Information technology–Portable Operating System Interface (POSIX(R)) — Part 2: Shell and Utilities, Dec. 1993.

[173] NXP B.V. i.MX Yocto Project User's Guide (LF6.6.23_2.0.0), June 2024.

[174] ISO/SAE 21434:2021: Road vehicles — Cybersecurity engineering, International Organization for Standardization (ISO), Aug. 2021.

[175] C. Miller. Lessons Learned from Hacking a Car. *IEEE Design & Test*, 36(6), Dec. 2019.