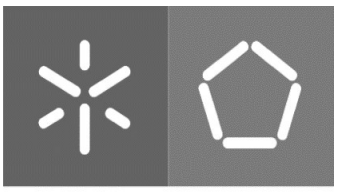
Universidade do Minho
Escola de Engenharia

João Carlos Ferreira Alves

Ontology-Driven Metamodeling Towards Hypervisor Design Automation: Secure Inter-Partition Communication (IPC)

Guimarães, January of 2018



Universidade do Minho
Escola de Engenharia

João Carlos Ferreira Alves

Ontology-Driven Metamodeling Towards Hypervisor Design Automation: Secure Inter-Partition Communication (IPC)

Dissertação de Mestrado em Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Adriano Tavares
Professor Doutor Sandro Pinto

Guimarães, January 2018

Declaração do Autor

Nome: João Carlos Ferreira Alves
Correio Eletrónico: a68515@alunos.uminho.pt
Cartão de Cidadão: 14614957

Titulo da dissertação: Ontology-Driven Metamodeling towards Hypervisor Design Automation: Secure Inter-Partition Communication
Ano de conclusão: 2018
Orientador: Professor Doutor Adriano Tavares

Designação do Mestrado: Ciclo de Estudos Integrados Conducentes ao Grau de Mestre em Engenharia Eletrónica Industrial e Computadores
Área de Especialização: Sistemas Embebidos
Escola de Engenharia
Departamento de Eletrónica Industrial

Universidade do Minho, 23/01/2018

Assinatura: João Carlos Ferreira Alves

Acknowledgements

Firstly, I would to thank my advisor Dr. Adriano Tavares, for proposing me a project for my thesis and for enlightening me throughout its development, with his vast knowledge and expertise. Following, I would to thank Dr. Sandro Pinto that also having made its contribution, by making me feel motivated when things looked more complicated, by providing me the opportunity to make one of my firsts scientific contribution, by being always available and by promoting companionship within the research group.

To the embedded systems class of 2015/16, a huge "Thank You!" for being a great group of work and for the great collaborative environment. Special note to the "German Gang" for being with me on my first experience abroad. Also, a special "Thank you!" To the irreverent Dr. Tiago Gomes, for helping me in achieving my first of all scientific contribution, and by being a great lab companion. To my friend and peer of 4th year projects, David Cerdeira, with whom I've shared moments of difficulty and despair, however always overcoming those things; for the long hours of work, conversation and knowledge sharing, one massive "Thank You!". Last, but certainly not the least, I would to massively thank my companion José Martins, for sharing with me long hours of work throughout the development of this dissertation, helping to overcome adversities that appeared, helping me feel motivated and for always pushing me further.

I kindly thank my friends and family, for the patience and support, for the time of companionship and relaxation, and for never ever giving up on me. I can not name all of them, as would I need a lot more pages. However, there are a few to whom I should leave the most dearest "Obrigado!", for having such a special place in my heart. This goes to: *Tinôcos* (Rita, Isa, Moreira, Teresa, Helena e Bruno), Nuno, Octávio and *Chuck*.

Finally, the biggest "Obrigado!" goes to my parents and little brother, for providing all that I needed, for being the highest reference of hard-work, for dealing with me on the hardest moments, and for never letting me down. My biggest goal is to make you feel proud! So for all this, and for what is yet to come "Obrigado, gosto muito de vocês!!".

Abstract

Embedded systems, which were by definition single-purpose, have evolved rapidly and nowadays are capable of supporting applications that, priorly, would be distributed between different hardware platforms. Virtualization proved its value in other fields, providing a way to safely collocate different applications on the same platform, enforcing security through isolation. Typical virtualization solutions follow a monolithic architecture, which usually contain large Trust Computing Base (TCB). Inherently, these are difficult to maintain, and could likely hide buggy software. Microkernels advocate a minimal TCB, that is restricted to an Inter-Partition Communication (IPC) infrastructure, a scheduler and memory management. Other functionalities are implemented in user-space, isolated from the system's critical functionalities. Service provision is achieved by leveraging Inter-Partition Communication (IPC) infrastructure, with well defined communication channels, and establishing trustworthy communication relations.

The inherent complexity of properly configuring such systems requires the use of dedicated tools, aiming at easing the configuration process. Model-Driven Engineering (MDE) advocates the conception of models towards software development, which would provide a more abstract, simplified view of the final system. Model description is often paired with Domain-Specific Languages (DSLs), that are featured with generative capacities. Thus, it becomes possible to transform a more abstract system into implementation artifacts (e.g. C/C++ code). Semantic technology has also been combined to modeling technologies, providing an alternative system representation, while enhancing modeling tools with: higher consistency, interoperability, automated validation and reasoning support.

Under the light of the above, a collaborative effort was conducted towards the enhancement of the in-house developed RTZVisor with microkernel-like principles, that resulted on the μRTZVisor. This thesis focus on the implementation of a secure IPC infrastructure, featured with a capability-based access-control facility, to improve its overall reliability by imposing Information Control Flow (ICF). Aiming at easing system's configuration, a modeling infrastructure was conceived that enabled the description of systems to be deployed on top of μRTZVisor. The infrastructure also converts the model representation into final source code with μRTZVisor resources configuration.

Resumo

Os Sistemas embebidos, que eram por definição de propósito único, têm evoluído rapidamente, sendo hoje em dia são capazes de suportar aplicações que, anteriormente, estariam distribuídas por diferentes plataformas. A virtualização provou o seu valor em diferentes áreas, possibilitando a consolidação segura de diferentes aplicações numa só plataforma, impondo segurança por isolamento. Tipicamente, a virtualização é implementada numa arquitetura monolítica, que requer muitas linhas de código. Consequentemente, estas são difíceis de manter e de validar, e podem conter erros "escondidos". As arquiteturas *Microkernel* advogam o principio da minimalidade, com o objetivo de minimizar o tamanho da sua implementação. Tipicamente, estas restringem-se a serviços de comunicação entre partições, escalonamento e isolamento espacial. Outras funcionalidades devem ser implementadas como aplicações, isolando-as assim, dos serviços críticos do sistema. Devem-se, por isso, estabelecer canais de comunicação seguros para se providenciarem serviços entre partições.

A configuração de tais sistemas pode ser complexa e morosa. Assim, requer-se a utilização de ferramentas dedicadas à automação deste mesmo processo. *Model-Driven Engineering (MDE)* coloca a conceção de modelos como objetivo primário do desenvolvimento de software. Este paradigma é usalmente combinado com linguagens de domínio específico, que possuem capacidades generativas, tornando-se possível converter a representação de um modelo em artefactos de implementação (nomeadamente código C/C++). A tecnologia semântica tem sido utilizada para conceção de modelos, melhorando as ferramentas de modelação com: verificação de consistência do modelo, interoperabilidade e validação automatizada.

No seguimento do que foi dito anteriormente, foi efetuado o melhoramento da implementação do hipervisor *RTZVisor*, conferindo-lhe príncipios de arquiteturas *Microkernel*, concevendo-se o *μRTZVisor*. Esta tese focou-se na implementação de mecanismos seguros para comunicação entre partições, complementados com funcionalidades para controlo de acessos. De forma a facilitar o processo de configuração, foi desenvolvida uma ferramenta de modelação, que converte a representação de um modelo em código fonte para configuração do *μRTZVisor*.

Contents

List of Figures

List of Tables

Listings

Glossary

μRTZVisor micro Real-Time TrustZone-assisted Hypervisor.

ABI Application Binary Interface.

ADL Architectural Description Language.

API Application Programmable Interface.

CBSE Component-based Software Engineering.

CC Common-Criteria.

CPU Central Processing Unit.

DAC Descritionary Access Control.

DMA Direct Memory Access.

DoS Denial-Of-Service.

DRAM Dynamic Random-Access Memory.

DSL Domain-Specific Language.

EAL Evaluated Assurance Level.

FIQ Fast Interrupt Request.

GIC Generic Interrupt Controller.

GPL General-Purpose Language.

GPOS General-Purpose Operating Sytem.

GUI Graphical User Interface.

ICF Information Control Flow.

IDL Interface Description Language.

IPC Inter-Partition Communication.

IRQ Interrupt Request.

MAC Mandatory Access Control.

MDA Model-Driven Architecture.

MDE Model-Driven Engineering.

MDSD Model-Driven Software Development.

MILS Multiple Independent Levels of Security.

MMU Memory Management Unit.

OCM On-Chip Memory.

ODSD Ontology-Driven Software Development.

OMG Object Management Group.

OS operating system.

OS Operating Sytem.

OWL The Web Ontology Language.

PCB Partition Control Block.

PP Protection Profile.

ROM Read-Only Memory.

RPC Remote Procedure Calls.

RTOS Real-Time Operating Sytem.

RTTI Run-Time Type Information.

RTZVisor Real-Time TrustZone-assisted Hypervisor.

SeML Semantically-enriched Modeling Language.

SoC System on Chip.

SRAM Static Random-Access Memory.

TCB Trust Computing Base.

TLB Translation Lookaside Buffer.

TOE Target Of Evaluation.

TZASC TrustZone Address Space Controller.

TZDL TrustZone Description Language.

TZMA TrustZone Memory Adapter.

TZPC TrustZone Protection Controller.

UML Unified Modeling Language.

VM Virtual Machine.

VMCB Virtual Machine Control Block.

VMM Virtual Machine Monitor.

W3C World Wide Web Consortium.

Chapter 1

Introduction

Embedded systems are widespread in modern society, due to their presence at the technological base for different systems in a broad range of application areas, including automotive, robotics, aerospace, IoT, and so on. Moreover, the whopping evolution of silicon technologies is pushing embedded systems further, making them suitable to integrate complex and powerful modern systems.

Nowadays, embedded devices are becoming increasingly complex, ranging from strict real-time control applications, to rich GUIs on a single embedded computer. Priorly, complex systems could encompass multiple embedded control units, which can now be supported within one single embedded computer [4]. This performance increase is favorable to systems cost reduction, by lowering the amount of hardware required, however, this incurs into an increased software complexity. The evolution of cellphones towards smartphones is another great example of the embedded systems evolution, as they used to be devices with extremely limited resources, supporting only a few set of functionalities, but nowadays, such devices are extremely powerful and can support several tasks on a multitude of scenarios and applications. A device that was provided with a single processor and limited in terms of storage capacity, whose purpose was mainly to make audio calls and send text messages, evolved into a mobile platform that completely surpassed its predecessor. It now includes, on a single device, a set of functionalties that formerly were separately available, such as audio player, high definition cameras, displays, internet, localization capabilities, along with a generous storage capacity [5, 1].

However, consolidating a multitude of systems in one single platform raises several security and safety concerns [1]. Virtualization urged key-enabling technology for addressing this problematic, since it enables the collocation of different Operating Sytem (OS), while ensuring that they run separately, confined to their physical space. This is achieved by providing an abstraction layer to those OSes, that manages hardware resources [6]. This helps to enforce the system's security, as an existent vulnerability in one OS does not affect the others. This approach

has been widely used in servers, as a way to not only provide security but also to achieve energy efficiency [5, 7].

The software that provides the desired abstraction and manages the isolation complexity is called a Virtual Machine Monitor (VMM), or Hypervisor. Monolithic hypervisors are usually software blocks whose implementations encompass a large Trust Computing Base (TCB), which often leads to hidden software bugs that may jeopardize system's functioning, and that are usually hard to maintain [8]. Microkernel architectural approach appeared as a solution to mitigate monolithic implementation problems. Their design aimed at reducing the complexity of kernel infrastructures by providing the bare minimum required functionalities to support process concurrency [9, 4], complying with the principle of minimality [10]. Microkernel implementations solely encompass a scheduler, memory management and Inter-Partition Communication (IPC). Moreover, the provided system calls should not impose any kind of policy. Also, according to the principle of minimality, the remaining functionalities must be implemented in user space, with well established IPC interfaces for service provisioning. This kind of implementation drastically reduces the kernel's size, and reduces the probability to have buggy software in the system's root of trust [8, 11]. Although these principles sound appealing, microkernel performance is poor compared to monolithic approaches due to high dependence on IPC infrastructure. This mindset was broken by Liedtke [12, 13] when he introduced a high-performance IPC in a microkernel implementation. Nowadays, microkernels are being widely used, and a few of them are even aiming virtualiation [14, 15].

In microkernel-based systems, servers and applications are usually of mixed criticality [16], and among them may be executing untrustworthy partitions. As a result, IPC interactions may occur between critical servers and untrusted parties. In these scenarios, the security provided by isolation is not enough. Communication relations are a concern, given that the IPC infrastructure constitute a means for attacking sibling partitions [17, 16]. One major concern whenever implementing IPC mechanisms is how to mitigate the chance of a Denial-Of-Service (DoS) attack [18]. Also, if a component is entrusted with confidential data, leakage of this data to the outside world should not be a possibility. Thus, isolation must be complemented with confinement, usually achieved by providing access-control facilities, in which access to system resources and interfaces must be explicitly granted instead of being available by default [19, 20]. This is often referred to as Information Control Flow (ICF).

One drawback of microkernel-based systems is that they required a high-level of

functionality segregation, which often pairs with a high-level of systems configuration, namely for establishing communication relations and resource multiplexing. In addition, most of such embedded systems are also characterized as real-time systems, which means that the real-time properties, such as response time, time quantum, among others, are also important design concerns. This increased complexity of embedded real-time systems leads to increasing demands with respect to requirements [21]. The use of model representation raises the level of abstraction on systems architecture, enables seeing the system at lower granularity, which is often useful for design validation and requirements verification [22]. Component-based architectures promote model representation by splitting the system into functional units, dubbed components, that interact through well defined interfaces. This kind of system design pairs well with functional segregation and IPC reliance inherent to microkernel-based systems. Also, it provides great control over granularity of representation, by advocating a componentised representations [9]. Pairing these concepts with a Domain-Specific Language (DSL) featuring code generation, not only enables design validation, but also incurs into higher productivity, reliability, portability, and eases software testing [2].

Microkernel implementations provide a greater fit for certification, toward security critical systems, due to its compact implementation [23, 9]. This could be achieved following the Common-Criteria (CC) standard and the Multiple Independent Levels of Security (MILS) architecture. The CC [24, 25, 26] is an international standard that targets computer security. It advocates the specification of a set of requirements that must be fulfilled within a class of products, constituting a Protection Profile (PP). In a PP assurance, requirements are tagged with a representative security level, dubbed Evaluated Assurance Level (EAL). The use of formal methods, mathematical models and proofs at the design stage is usually required for high security levels. Lower levels simply impose functional or structural testing in addition to methodical development processes. The MILS [27, 28, 26] architecture defines a conceptual framework towards system security, by using a divide-and-conquer approach. This is achieved using separation, composition, and layered assurance and it is implemented using four layers, namely: trusted hardware, separation kernel, middleware, and applications. This reduces the amount of code to be verified and, consequently, increases its scrutiny. According to the layered assurance principle, the security level increases when going down the stack [27], e.g. the applications can not be more secure than the separation kernel. The separation kernel is built on fundamental security policy:

- ***Information Flow*** - Regards to the permitted communication channels between partitions;
- ***Data Isolation*** - This ensures that a partition cannot access resources in other partitions;
- ***Temporal Isolation*** - This ensures that applications within partitions are executed for the specified duration in the system schedule;
- ***Fault Isolation*** - This postulates that failure is confined within a partition's domain;
- ***Sanitization*** - To ensure the information flow requirement, the separation kernel should clean any shared resources (microprocessor registers, system buffers, etc.) before being used by another partition.

The exposed principles can constitute a PP based on MILS architecture, that could be applied in the implementation of an embedded hypervisor. Thus, contributing to the certification of, or at least part of, the Target Of Evaluation (TOE).

1.1 Context

This thesis was conceived within a collaborative effort from students working at Embedded Systems Research Group, the University of Minho. The ultimate goal of the group's work is to provide an embedded secure hypervisor, with great focus on security, integrated with a design automation tool that allows the faster development and configuration on the previous platform, while following a secure software development approach. As such, some of the working members focus on developing a meta-modeling infrastructure, while others will focus on developing and modeling hypervisor artifacts. As the development of a embedded hypervisor from scratch is an extensive task, the developed work will have on its roots the in-house developed Real-Time TrustZone-assisted Hypervisor (RTZVisor), that constitutes a monolithic implementation aimed at real-time applications. As such, concurrently to this thesis, there will be elements focusing on the enhancement of the aforementioned architecture with microkernel-like principles, with great focus on device sharing, spacial and temporal isolation. Other dissertations will be focused on achieving a secure boot process, data and control flow integrity, an ontology-driven DSL, dubbed Semantically-enriched Modeling Language (SeML).

This thesis focus on enhancing the hypervisor kernel with a secure IPC infrastructure, as well as the modeling of developed artifacts. The IPC infrastructure should conceived aiming at security, and should be complemented with access-control to provide a way of achieving ICF.

1.2 Goals

This thesis focus on the enhancement of RTZVisor with microkernel-like principles, in addition to providing a means for easily configure the developed artifacts. RTZVisor is a monolithic hypervisor, implemented using the C language, that leverages ARM TrustZone technology towards virtualization, providing a close to full-virtualized environment. However, as it is going to be discussed in further sections, this comes with a set of limitations, namely the absence of IPC and synchronization mechanisms, as well as the absence of a dedicated modeling infrastructure towards system configuration.

Given this, the following set of goals were specified:

1. Perform a comprehensive survey on IPC and access-control facilities within microkernels context, identifying common vulnerabilities;

2. Implement an IPC infrastructure, with both communication and synchronization mechanisms. This should be achieved by following state-of-the-art principles with respect to microkernels, placing special focus on security;

3. Implement a lightweight access-control facility, that enhances IPC mechanisms with ICF;

4. Adopt a secure development process by recurring to C++ instead of C, performing code certification recurring to the MISRA C++ standard, that advocates a set of principles and rules towards a more secure development;

5. Conceive a meta-model towards hypervisors design automation, placing special focus on IPC resources and communication channels;

6. Evaluate the hypervisor's implemented artifacts, namely in terms of performance and security;

7. Evaluate the conceived work regarding the design automation tool, namely identifying the gains of the used approach, as well as the counter-parts.

1.3 Document's Structure

This document is structured as follows:

- **Chapter 2** provides the knowledge framework for understanding the developed work. It defines virtualization fundamental concepts, followed by making the comparison between two different kernel architectures. Then, it provides a review on IPC comprising elemental concepts, policies, elemental mechanisms and security. Access-control general concepts are introduced, to then support the description of capability-based access-control facilities. Some microkernel implementations are described, emphasizing on implemented Inter-Partition Communication (IPC) policies and access-control approaches. The Arm TrustZone architecture is described to support the architectural description of RTZVisor, which is the root for the developed hypervisor artifacts. Model-Driven engineering is introduced, emphasizing DSL, component-based approaches and ontologies. Lastly, SeML infrastructure is described.

- **Chapter 3** describes the μRTZVisor architecture. It starts by presenting an overview of the implemented work, starting by stating the advocated principles and architectural goals, as well as a general functional description. Then, each subsystem within the kernel is described, following an order that enables to understand how each subsystem role, and which dependencies are established with the remaining. It is then followed by a structural and behavioral description of the developed work; however, confining to communication mechanisms, access-control facilities and other correlated contributions.

- **Chapter 4** describes our approach towards the hypervisor's design automation. It starts by describing the goals for the design automation, specifying SeML's role in all this. Then, a DSL towards microkernel-like systems is presented. It is described the developed ontology for integration with SeML infrastructure, language constructs, and code generation process. An example is then provided, and a discussion is conducted to evaluate the developed work and achieved results.

- **Chapter 5** concludes this thesis. It presents the conclusions obtained from this research, highlighting the contributions, identifying the limitations, and suggesting future work towards addressing pointed limitations.

Chapter 2

Theoretical Foundation and Background

This chapter aims at providing the necessary background to understand and contextualize the developed work. Virtualization is defined, followed by a comparison between microkernels and monolithic architectures. Next, it is presented a concise survey on IPC, approaching security, as well as applied policies and mechanisms. It is followed by the definition of access-control, and then presenting capabilities as the *de facto* standard for access-control in microkernel-like architectures. Then, it is introduced the key-enabling technology towards RTZVisor implementation, followed by the latter's architectural description. Model-driven engineering concepts are also presented, placing emphasis on ontologies and component-based architectures. Lastly, the SeML infrastructure is described.

2.1 Virtualization

Virtualization enables the cooping of multiple OSes, in this context also called Virtual Machine (VM)s, by providing an environment that abstract the underlying hardware platform and enables the safe sharing of available resources. This abstraction is provided by introducing another software layer, often called Virtual Machine Monitor (VMM) or hypervisor [4, 29, 30, 5]. Its main applicability is to

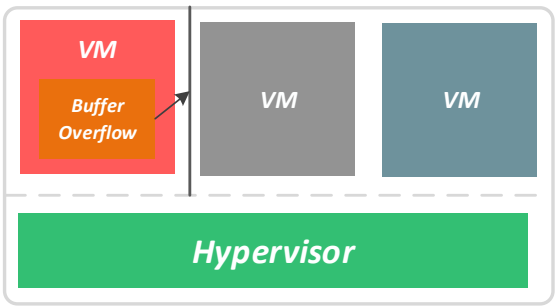
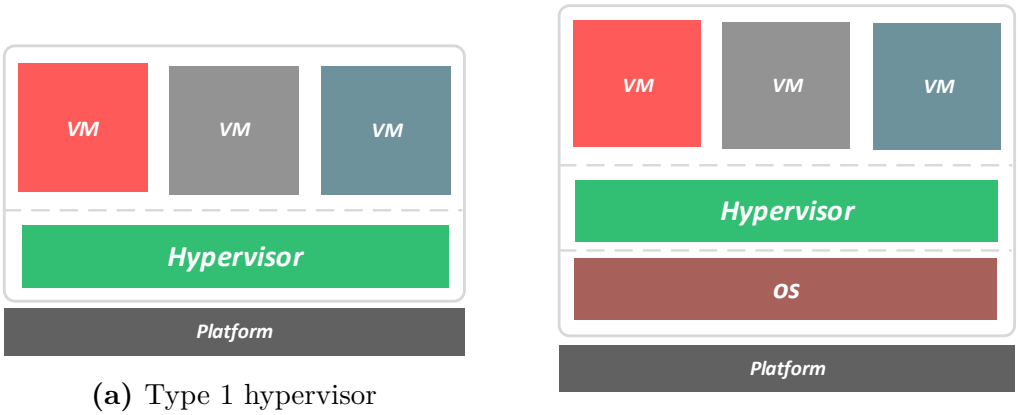


Figure 2.1: Common example of Security achieved by isolation [1].

improve system's safety and security, by providing environment isolation, which is expected to prevent that a given vulnerability or buggy behavior does not jeopardize sibling VMs [26]. Figure 2.1 is a common example of the security provided by isolation, in which a buffer overflow attack happens in a given VM. This is also a necessary requirement for other incurring benefits, like application consolidation, which helps in reducing production costs by lowering the amount of required hardware to support all the desired functionality, and to reduce energy consumption by load-balancing across clusters, creating VMs on slightly utilized hosts[29, 1, 31]. Other uses for applying virtualization techniques are: to enable hardware compatibility with legacy software applications; to increase software portability making it hardware-independent; and to provide an environment that support different OSes [32, 4]. A typical example of a very-well known virtualized system is the cell-phone, that usually requires a Real-Time Operating Sytem (RTOS) to interact with the cell-phone's sensors and to perform important real-time tasks, in addition to a General-Purpose Operating Sytem (GPOS) that supports the Graphical User Interface (GUI), functionality towards network connectivity, and so on. Virtualization in this context prevents, for example, a remote attack on the GPOS to have repercussions on the RTOS.

There are two types of hypervisors: type 1 (Figure 2.2a) that execute directly above the hardware, being the only software executing in kernel mode; type 2 (Figure 2.2b) hypervisors run on top of an OS [29, 5, 30, 31]. In both cases, VMs must behave exactly in the same matter within the virtualized system as they would, by executing directly over the hardware platform. In addition, for tightly coupled systems the IPC infrastructure must behave efficiently, i.e. with good performance and considerable throughput [5, 6, 31]. Typical embedded virtualization solutions belong to type 1 category, as type 2 would not be a good fit for embedded systems. This is due to the overhead inherent to the existence of an OS between the hypervisor and the hardware platform [30].

Regardless of their type, there are two different approaches towards a virtualization solution. Full-virtualization solutions are characterized for providing an environment that allows legacy VMs to execute without modifications, which is achieved by trapping sensitive instructions of the guest OS and emulating them on the actual hardware platform. In contrast, para-virtualization requires specific changes at the OS level in order to make VMs suitable to run over a given hypervisor, replacing specific instructions with system calls to the hypervisor. In one hand, full-virtualization require little engineering effort for VM's deployment in a given virtualized system and does not require code availability. Para-virtualized



(a) Type 1 hypervisor

(b) Type 2 hypervisor

Figure 2.2: Type of hypervisors architectures.

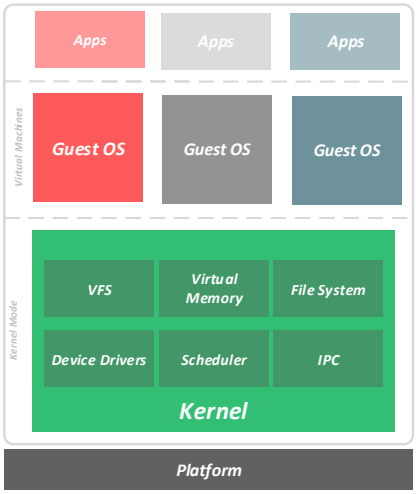
solutions usually incur into better performance, as the performed changes may also aim at removing unnecessary operations. However an obvious obstacle is the OS code availability, necessary to perform the aforementioned changes [33, 5].

Although virtualization is a way to achieve many benefits, these come at expected costs. Typical virtualization solutions can be modified in order to make them usable in embedded platforms, although they lack on fulfilling restrictions and requirements specific of embedded systems, such as memory footprint or real-time scheduling guarantees [6, 4, 29]. For critical systems the amount of code constituting the Trust Computing Base (TCB) is relevant, as the number of buggy behavior is directly proportional with the number of code lines. Thus, in addition to being carefully written, these must be reduced to the strictly necessary, favoring system's safety and also for matching embedded systems' memory requirements [6, 30]. Regarding scheduling capabilities, real-time systems are often required to quickly operate upon events, as well as to provide a meaningful time quantum for hard-real-time systems to complete their work under established deadlines [4, 15].

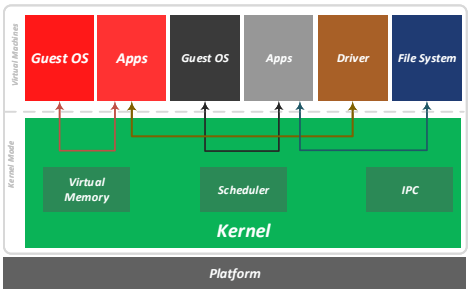
2.2 Microkernels vs Monolitic

The kernel part of any OS is the software part that runs with the highest privilege, executing with no access restrictions over the hardware resources, in addition to being part of its TCB. TCB is defined by a set of assertions that the complete system is built upon, which is intended to be secure and reliable. Penetrating this systems' layer corroborates all asserted security and safety that characterizes the system [18]. In monolitic (Figure 2.3a) architectures, it is where

most OS functionalities are implemented, including interrupt handling, to memory management, device drivers, network stacks, IPC facilities and file systems. [9, 10]. The functionality consolidation, hardware architecture heterogeneity and support for different contexts (i.e. RTOS-based applications and GPOSes) contribute to increased software complexity and, consequently, considerable growth of kernel code size. Considerably large code implementations are usually difficult to maintain, escalate and are often prone to hide bugs which, given the amount of code, are hard to detect [9, 23, 34, 35]. It has been proved that, for the aforementioned reasons, smaller TCBs are preferable for the development of criticality computer systems [8, 36]. Microkernels (Figure 2.3b) constitute a paradigm shift



(a) Monolitic kernel architecture



(b) Microkernel architecture

Figure 2.3: Monolitic vs Microkernel architectures

from monolitic architectures, which basically consist in moving kernel services to userspace. The only acceptable services to provide within the kernel are those of memory management, IPC, and scheduler, thus promoting a minimalistic implementation that aims at considerably reducing TCB's code size [15, 14]. In addition, primitives implemented within the microkernel must comply with the principle of minimality [14, 10]. This advocates that system calls must be characterized with policy absence, i.e., as simple as possible. However, these must comprise a vast and rich number of services that make the implementation of meaningful systems feasible [1]. In order to keep the kernel simple, device drivers and other services (that regard with some type of service) are confined to a userspace server. Device drivers are usually the least trusted code within a kernel, and are usually prone to suffer remote attacks. The isolation in userspace limits the attack surface and prevents them to access system's critical information [8, 23].

The main handicap of microkernel approach is the obligatory reliance on IPC infrastructure, which, for a long time, was the main reason for microkernels to not be accepted as a valid solution [37]. By moving services within the kernel to a userspace server, the way to access them is through Remote Procedure Calls (RPC) interfaces or by client-server communication (more detail about this given in Section 2.3). As such, IPC has to be of great performance, which was achieved by Liedtke et al. [13, 12] by implementing L4 microkernel whose design primarily goal was to provide fast IPC, proving that microkernels are a valid solution for modern computing systems. However, IPC is not only used for client-server communication: Hardware-generated interrupts or traps caused by user level processes can be converted into IPC messages by the microkernel and are sent to a responsible handler (a user level process) [4].

Security is also achieved by providing an environment that fosters functionality segregation, which results in higher level of fault containment, as a given vulnerability is confined in a smaller domain. In virtualization contexts, guest OSes's kernel and all running applications are segregated in its own separated thread, in its own address-space. So, applying microkernels into virtualization domain requires highly paravirtualized VMs , which may be a drawback considering the required engineering effort. In opposition, functionality segregation, combined with clear microkernel interfaces ease to achieve a modular design, which brings benefits for testing, debugging and system scalability [11, 37].

2.3 Inter-Partition Communication

In virtualized environments VMs often need to communicate in order to exchange services. In non-virtualized machines communication happen through TCP/IP network stack, which could also be achieved in VMs. However, when communicating with co-resident machines, this entails unnecessary performance burden, because the infrastructure does not differentiate between data coming from the same machine and from outside the machine, processing it the same way, i.e., through TCP/IP network stack [38]. This could easily be circumvented by making the kernel aware of these scenarios while maintaining the abstraction, i.e., maintaining the interface for IPC mechanisms. This approach would traduce into gains of performance and software portability, as it would these mechanisms transparent to the user API, thus reducing the para-virtualization effort [38]. There already exists some work in virtualization exploiting this kind of approach like in

[39, 40], where shared memory facilities were used to circumvent the stack's software whenever the recipient IP belongs to the same machine as the sender. Given that embedded systems are highly cooperative systems, that often require synchronization facilities and faster data transmission mechanisms, user API transparency is often dropped in favor of performance.

Microkernels approaches heavily rely on IPC mechanisms for service provision due to the high segregation of VMs into domains, which work in favor of security, however, incurring into performance burden. Given the central role it possesses in microkernels, and also other types of highly cooperative infrastructures, it must be carefully designed as it could be exploited for possible attacks [41, 13, 9]. In [16] some requirements that IPC infrastructure must usually fulfill are presented:

- Reliable delivery, which refers to guaranteeing that messages are successfully delivered;
- Atomicity characterizes IPC operations as uninterruptible;
- Recipients endpoints must not be forgeable;
- The message's integrity is maintained though out the communication process;
- Multiple IPC calls must be independent from each other;
- Snooping traffic must also be impossible.

Also, when specifying system API, it should be taken into account that IPC system calls must be general enough to be applied in a plethora of contexts; however, IPC system calls must be powerful enough to reduce the number of system calls required, because the change between user and kernel modes usually deteriorates performance [13, 42].

2.3.1 Policies and Mechanisms

There are two types of policies that are usually applied for message passing IPC mechanisms: synchronous and asynchronous. In synchronous communication, involved partitions meet at a given point in time to proceed the communication. This means that both are in a known state, i.e., the sender is in a sending state specifying the message to be transfered, while the receiver is prepared for the acceptance of a new message specifying the incoming message's buffer. Thus, at least one of them must be blocked waiting for the other to perform the complementary operation. Inherently there are some accruing benefits in terms of

performance and resource management. The data transfer can happen directly between address spaces, which will reduce the message propagation latency, and no buffering within the kernel is required [17, 16, 14]. Synchronous communication usually encompass a donation scheme, where in a client-server scenario the requester may donate its time execution in order to quicker resolve its dependency towards the server [43]. This is suitable for Remote Procedure Calls (RPC) abstractions, which consists on providing a interface to access services from a given server through the use of function calls. In this scenario, from the client's perspective, a function call is solely happening. However, it abstracts a sequence of IPC operations that sends the request and blocks waiting for the response. It also performs parameter marshallings and unmarshallings upon message sending and upon response arrival[10, 44, 16]. Apart from performance, a sound synchronous IPC design encompass security considerations, as synchronous communication is prone to DoS attacks, or even unintended deadlocks. One widely spoken problem of synchronous communication is the asymmetric-trust problem, which happens when multiple clients rely on the same server. In this relationship that works based on trust, there could be one badly intended partition that jeopardizes the execution of one of its siblings. Figure 2.4 depicts the different scenarios in which the asymmetric trust model becomes a problem. The best case scenario (A), both client and server perform their operations as supposed. Thus, the execution happens without incurring into problems to the system.

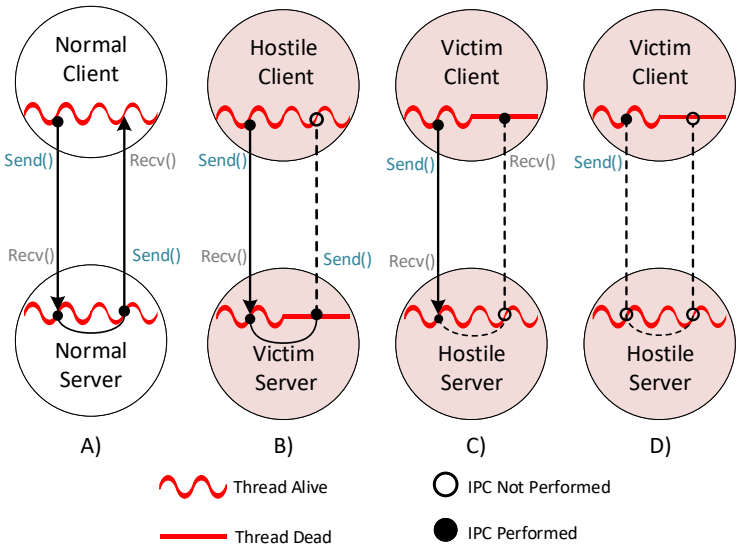


Figure 2.4: Representation of the asymmetric trust model.

On scenario (B) one malicious client may block the server infinitely, causing them to fail answering the requests from other clients. This problem can and

should be considered in the opposite way, where a malicious server may cause a client to block, by failing to perform the desired operations at specific points in time (scenarios C and D) [16, 17]. Timeouts could be used in order to overcome these problems, however from L4 family experience [14] these are not effective, due to inappropriate use from user perspective, and also because it is not a good mensurable way to determine a given timeout value. Synchronous communication pushes a given system towards a purely multi-threaded environment, which forces functionality segregation, isolation and consequently enforces fault-containment. Nonetheless, it does not pose a definite solution against DoS but it helps to reduce the impacts of such an event [14, 16].

Regarding asynchronous communication, in contrast with the previous policy, message sending and reception does not meet at a specific point in time. Consequently, there is no blockage of any involved partition, therefore the asymmetric-trust problem is avoided [17, 16]. However, buffering within the kernel and, consequently, a double data copy are required. This makes partition execution more secure, but with an inherent performance cost [14]. There is no risk of directly causing a partition to fail; however, when not properly mediated, asynchronous communication is prone to suffer attacks of resource exhaustion, so there's the need to provide ICF mechanisms. Also, asynchronous communication can be paired with event mechanisms for interrupt execution upon message arrival [18], which in turn can also provide the opportunity for tampering a partition's execution.

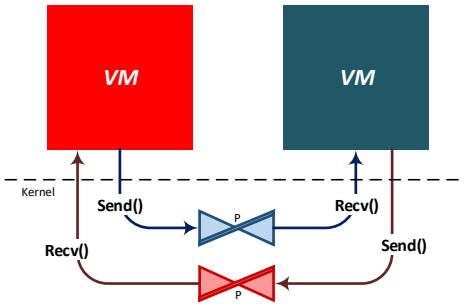


Figure 2.5: Representation of the asymmetric trust model.

In both synchronous and asynchronous communication, one thing to have in mind is the IPC destination. Original L4 microkernel's implementation used threads as the destination of a given IPC operation. This would create the opportunity for a malicious entity to gather information which could be used to create an effective attack [18, 14]. Thus, the concept of endpoint was introduced and it consists on providing an abstraction to communication, from where messages could be read or write to. The most widely used concept is the notion of a *port* that usually is used as being the endpoint for message passing. Figure 2.5 depicts

the simple scenario for using *ports*. These could likely have an owner, who is intended to be the recipient for sent messages [14, 10, 19].

2.3.2 Review Of IPC security

In the literature there are some publications that aim at documenting IPC vulnerabilities in multiserver systems, as well as the attacks [16, 18, 17, 42]. Microkernel's principal of minimality, as explained in Section 2.2, aims at mitigating these effects by moving policies to userspace, preventing malfunctions to spread. Nonetheless, when designing an IPC infrastructure it must be considered where vulnerabilities may be located to prevent or mitigate possible attacks [19].

In [18] attacks have been documented as direct and indirect. Direct DoS attacks that aim at monopolizing a given resource, may cause another partition relying at the resource, or even the whole system, to fail. For example, sending a considerate amount of messages may cause a partition to spend most of its time dealing with messages, forbidding it from performing another concurrent task. Indirect attacks are those that aim at tampering the system's manager, thus not directly tampering a sibling partition, however jeopardizing its execution through means of the manager. For example, sending a lot of spurious messages, with the purpose of exhausting buffering resources. In [16, 17] these notions are extended with categorizing attacks according to the means of exploit. One means of exploit is the IPC subsystem by maliciously tampering the parameters of system calls, which, for example, may include a pointer to an invalid memory region, preventing the communication to happen. Another way to attack the IPC subsystem would be by exhausting memory resources, in the case they are dynamically allocated. The message delivery process is another concern, where addressing for IPC operations may also be secured, since partition A may not be intended to access services from partition B, thus by performing any communication attempt with partition B, could mean A was trying to tamper B. As such, there should be prevented identity spoofing by maliciously party. Another message delivery problem should be the payload size. There should exist prevention against memory overflows, in order to recipient's memory not be overwritten. Finally, another concern should be the established communication relationships and interactions between groups, in which many partitions can be blocked along the process. As previously explained in Section 2.3.1, synchronous communication requires mechanisms to prevent deadlocks and unwanted blocks to happen, as well as to avoid the asymmetric trust problem. In addition, quantum donation combined with

priority inheritance may introduce problems like priority inversion and starvation of low priority partitions [43].

Stated the problems that may occur whilst conceiving an IPC infrastructure, some solutions for them must be pointed out [16, 17]. Regarding attacks through the system call, parameter checking must be thoroughly deployed, namely to ensure that memory accesses are properly specified. To prevent deadlocks and unwanted blockage results, as aforementioned, there is no practical way of defining timeout values, as such, has been dropped in some microkernel implementations [14]. Asynchronous communication prevents these scenarios because partitions are not required to block throughout the communication process. The level of functional segregation inherent to microkernel-based systems also reduces the impact of an unwanted blockage.

2.4 Access-Control

Although Hypervisors and OSes already confine processes to their own sandbox, i.e., enforce separation between partitions preventing system failures and vulnerabilities to spread across the entire system, kernel APIs can still be exploited to cause some unwanted effects on the overall system [45]. Namely, system calls for message sending (among others) should be monitored to prevent malicious communications attempts, that aim at tampering a sibling's partition execution, or even exhaust message buffering resources [16]. Or a malicious partition may ask the system to erase page table entry that contain sensitive critical information. Thus, critical systems must be featured with mechanisms to prevent actions which are performed by unintended agents [19].

On OSes, access-control consists on creating mechanisms to restrict the way partitions operate on system's interfaces and resources, which usually requires the existence of a reference monitor that mediates every activity accessed by a given entity. These monitors must carry an authentication to prevent spoofing attacks, where a given subject masquerades its identity [46]. An important goal of these systems is to ensure that the principle of least-authority prevails. It states that partitions uniquely possess permissions to perform what strictly necessary. [47, 19] advocate that this is a requirement to have confinement within a system. There are also concerns regarding access control mechanisms' simplicity and granularity over resources. A mechanism can not be too simple that is easily fooled or provide more permissions than necessary, and at the same time should provide control over granularity regarding resources within a system [26].

Inside kernels there are two types of applied access policies: Mandatory Access Control (MAC) and Descritionary Access Control (DAC). The first refers to more rigid policies, that determine permissions are configured based on subjects and objects, in addition to being configured uniquely by a privilege entity, while the later refers to more flexible approaches, like the distributed scenario where permissions may be granted by more than one entity within a system [26, 46, 47], often by object owners [48].

2.4.1 Capability-based Access-Control

Capabilities are the *de facto* solution in distributed systems and microkernel-like operating systems for access-control facilities [49]. In their simplest form, these are a reference to an object that can be assigned to one or more partitions [19]. Capability ownership implies some access permission over the referenced object, which in turn encapsulates functionality, accessible through a well-defined interface [19, 44]. In systems that are completely based in capabilities, all operations require the possession of one referencing the object which provides a given functionality, which may include communication objects in operating systems, as well as memory page-tables. The access monitor is the entity that manages the objects, which in this facility may be either the kernel to manage kernel objects, or even applications. An example of an application is a process that supports a file system may mediate files read and write operations recurring to capabilities [44]. Figure 2.6 depicts a general scenario of object access mediate through capabilities. *Subject Y* and *Subject Z* aim at performing one operation each. In order to do so, they should provide their capability to the *Objects Monitor*, that upon permissions verification will perform the operation on the referenced object.

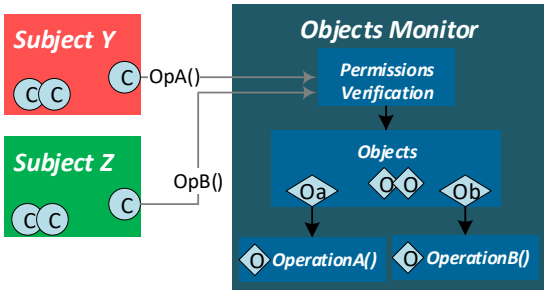


Figure 2.6: Representation of the asymmetric trust model.

In capability-based access-control systems permissions are conveyable, meaning the set of permissions a given subject possesses may change over time [44, 47]. In order to do so, another subject within the system owning a capability, can ask the

objects monitor to perform a grant operation, thus transferring a subset of its-own rights. The grant is, on itself, an operation whose permission must be set in the respective capability. In addition, a revocation operation is also usually available [20]. The existence of grant and revoke operations makes capabilities suitable to provide DAC-like access-control facilities. However, their use must be restricted to limit propagation towards intended subjects [19]. With these operations it is possible to achieve more rigid scenarios, that do not even change over time, like with MAC policies [26].

To make capabilities dependable there are a set of requirements that need to be met: subjects should not be capable of tampering capabilities, to either forge their identity nor possessing rights over an object; and, permission check must be lightweight to avoid performance degradation. This type of access-control facilities can be implemented in a centralized schema, in which protection is achieved by hardware or provided by the OS. In distributed scenarios the capability possesses owner's identity, thus protection is provided by means of signatures, preventing identity tamper towards unintended access [44, 50, 47].

Capabilities are also seen as the perfect fit to manage granularity and to support different types of objects for access-control, as it can be used to obliquely mediate access to a plethora of different types of objects [26]. The most obvious advantage in applying capabilities is for enhancing communication infrastructures with ICF. Thus, enforcing isolation provided, possibly, by a microkernel with containment of subsystems. This means, permissions configuration will influence how capabilities are granted over the time, thus aggregating and compartmentalizing groups of subjects in subsystems [44, 48].

2.5 Microkernels Related Work

Microkernels were not always a viable solution, due to the architecture's reliance on IPC, which constituted a bottleneck on system performance. The L4 microkernel, developed by Jochen Liedtke, appeared to break the stigma surrounding microkernels, proving their utility when providing efficient IPC mechanisms. L4 is the root of a family tree of microkernels that have a proven record of efficient performance and reliability, by following the core idea of kernel minimality and policy-void mechanisms [14]. In this section, we briefly survey some members of this family that served as the main source of inspiration for the ideas implemented in μRTZVisor, emphasizing those which aim to support virtualization.

Fiasco is an open-source descendant of L4 implemented in C++ aimed at security and critical real-time systems. It implements protected address spaces, synchronous IPC and a scheduler with multiple fixed-priority levels, whereby kernel executes a round-robing algorithm on threads characterized with the same priority [51, 52]. The latest version Fiasco.OC also includes capabilities for access-control, which are propagated through IPC [53]. In addition, capabilities are protected within the kernel memory, thus threads do not access directly to capabilities, making the access-control infrastructure reliable. Each partition possess a capability table, that maps the partition's access permissions into kernel resources. Capabilities are propagated through message passing facilities, and mapped into the capability table upon mapping request by the capability's recipient [19].

The NOVA microhypervisor [54] proposes a solution that deallocates virtualization to user space, which will inherently incur performance overhead and augmented engineering effort due to the highly para-virtualized approach, although augmenting security by significantly reducing TCB's size. As such, the kernel solely provides services for spacial and temporal isolation, in addition to message passing and synchronization mechanisms. Also, kernel operations require capabilities to access the kernel objects. Capabilities are immutable, and inaccessible in user-space, thus permissions are statically assigned and prevail through out system's execution. Accessing them is done by an integral number that works as the index for the domain's capability space. This happens every time a system call is performed. Destinations for IPC operations are endpoints designated by portals, which represent an entry to the owner's address space. In client-server communication, a reply capability can be granted to the server referencing a portal for the response to be addressed. Also, semaphores objects are provided for synchronization purposes. [55] presents Mini-NOVA, a simplified version of NOVA ported the Arm cortex-A9 architecture from the original x86 implementation. It aims at achieving lower overhead, smaller TCB size and higher security, thus making it more flexible and portable for embedded-systems.

PikeOS is an early spin-off of the L4 microkernel, whose purpose is to address requirements of safety-critical real-time embedded systems. It features spacial and temporal isolation, favoring minimum code size, in some cases to the detriment of flexibility [4]. It aims at providing a system that enables the coexistence of time-driven and event-driven partitions. The result is not the perfect fit for this kind of system, although by properly configuring each partition, it is possible to achieve a considerably good compromise [15]. It is also featured with access control facilities based on the *abilities* concept. To each partition is assigned a table with

abilities that is not expandable over time, where each entry represents a system call that the partition is allowed to perform. Given that partitions may be created dynamically, a parent partition may further restrict child's permissions, however it can never extend them. In addition, there were efforts on verifying functional correctness of PikeOS [56].

OKL4 adopts a microkernel approach completely directed at virtualization and, thus, is dubbed a microvisor [8]. It features a fast and reliable IPC, which is abstracted by channels and virtual interrupts for synchronization purposes. It implements only asynchronous IPC, which maps better to the VM model, and is less susceptible to DoS attacks. By the heritage of its seL4 predecessor, it provides access-control facilities based on capabilities, since any kernel operation requires one. OKL4 has been augmented to take advantage of the Arm virtualization extensions and support unmodified guest OSes [57].

EROS [47, 17] is a microkernel-based operating systems that was pioneer in providing access-control facilities based on capabilities. All system provided functionality is done by invoking a capability, providing the index from partition's capability list, referencing a kernel object. Regarding the IPC facility EROS provides three different primitives: send, call and return. The first performs a send operation and continues partition's normal flow, while the others block waiting for new messages. The laters distinction resides on the type of used capability to perform the operation. The call primitive will generate a reply capability for the recipient to use with a return primitive. This imposed semantic aims at safe client-server communication.

Finally, MINIX [58] is an open-source microkernel, firstly developed by Tannenbaum trying to prove the worth of microkernel principles towards trustworthy computer systems. Nowadays, the current available version is the MINIX3, that supports IPC operations through the use of ports as the destination for these operations. Most IPC primitives are synchronous, as this has been used for security evaluations regarding IPC policies [16]. In addition to support synchronous, it provides two asynchronous primitives, the *asend* that sends message without blocking the sender, and *notify* that provides an event delivery mechanism.

2.6 ARM Trustzone

TrustZone technology is a set of hardware security extensions, which have been available on Arm Cortex-A series processors for several years [59] and has recently

been extended to cover the new generation Cortex-M processor family. TrustZone for Armv8-M has the same high-level features as TrustZone for applications processors, but it is different in the sense that the design is optimized for microcontrollers and low-power applications. In the remainder of this section, when describing TrustZone, the focus will be on the specificities of this technology for Cortex-A processors (Figure 2.7).

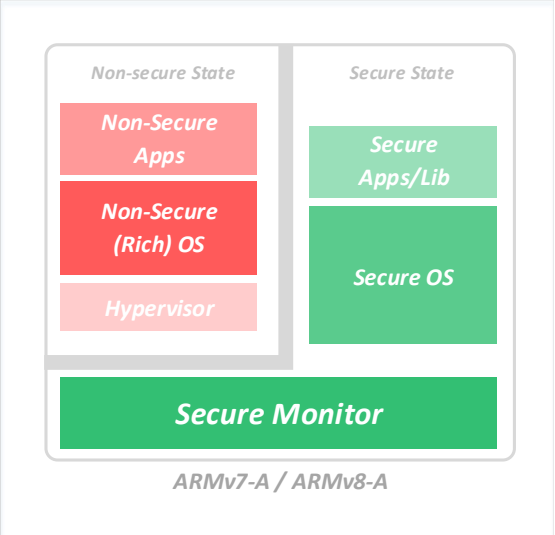


Figure 2.7: RTZVisor system architecture.

The TrustZone hardware architecture virtualizes a physical core as two virtual cores, providing two completely separated execution environments: the *secure* and the *non-secure* worlds. A new 33rd processor bit, the *Non-Secure* (*NS*) bit, indicates in which world the processor is currently executing. To preserve the processor state during the world switch, TrustZone adds an extra processor mode: the monitor mode. The monitor mode is completely different from other modes because, when the processor runs in this privileged mode, the state is always considered secure, independently of the NS bit state. Software stacks in the two worlds can be bridged via a new privileged instruction-*Secure Monitor Call* (*SMC*). The monitor mode can also be entered by configuring it to handle interrupts and exceptions in the secure side. To ensure a strong isolation between secure and non-secure states, some special registers are banked, while others are either totally unavailable for the non-secure side.

The TrustZone Address Space Controller (TZASC) and the TrustZone Memory Adapter (TZMA) extend TrustZone security to the memory infrastructure. TZASC can partition the DRAM into different secure and non-secure memory regions, by using a programming interface which is only accessible from the secure side. By design, secure world applications can access normal world memory, but

the reverse is not possible. TZMA provides similar functionality but for off-chip Read-Only Memory (ROM) or Static Random-Access Memory (SRAM). Both the TZASC and TZMA are optional and implementation-specific components on the TrustZone specification. In addition, the granularity of memory regions depends on the System on Chip (SoC). The TrustZone-aware MMU provides two distinct MMU interfaces, enabling each world to have a local set of virtual-to-physical memory address translation tables. The isolation is still available at the cache-level because processor's caches have been extended with an additional tag that signals in which state the processor accessed the memory.

System devices can be dynamically configured as secure or non-secure through the TrustZone Protection Controller (TZPC). The TZPC is also an optional and implementation-specific component on the TrustZone specification. To support the robust management of secure and non-secure interrupts, the Generic Interrupt Controller (GIC) provides both secure and non-secure prioritized interrupt sources. The interrupt controller supports interrupt prioritization, allowing the configuration of secure interrupts with a higher priority than the non-secure interrupts. Such configurability prevents non-secure software from performing a Denial-Of-Service (DoS) attack against the secure side. The GIC also supports several interrupt models, allowing for the configuration of Interrupt Request (IRQ)s and Fast Interrupt Request (FIQ) as secure or non-secure interrupt sources.

2.7 RTZVisor

Real-Time TrustZone-assisted Hypervisor (RTZVisor) [60] is a bare-metal embedded hypervisor that relies on TrustZone hardware to provide the foundation to implement strong spatial and temporal isolation between multiple guest OSes. RTZVisor is implemented in the C language and follows a monolithic architecture (Figure 2.8).

All hypervisor components, drivers and other critical parts of the virtualization infrastructure run in the most privileged processor mode, i.e., the monitor mode. The hypervisor follows a simple and static implementation approach. All data structures and hardware resources are predefined and configured at design time, avoiding the use of language dynamic features.

Guest OSes are multiplexed on the non-secure world side; this requires careful handling of shared hardware resources, such as processor registers, memory, caches, MMU, devices, and interrupts. Processor registers are preserved in a specific Virtual Machine Control Block (VMCB). This virtual processor state includes

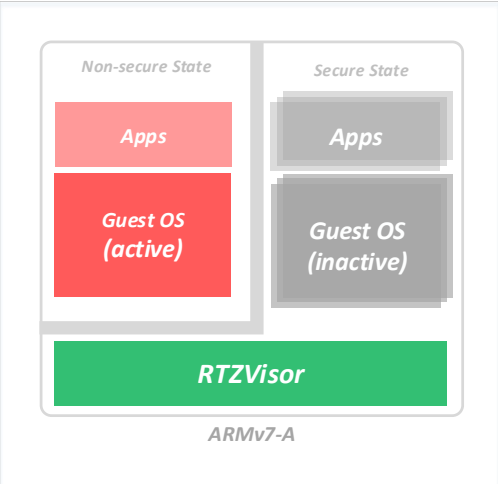


Figure 2.8: RTZVisor system architecture.

the core registers for all processor modes, the CP15 registers and some registers of the GIC. RTZVisor offers as many vCPUs as the hardware provides, but only a one-to-one mapping between vCPU, guest and real CPU is supported. RTZVisor only offers the ability to create non-secure guest partitions, and no means of executing software in secure supervisor or user modes.

The strong spatial isolation is ensured through the TZASC, by dynamically changing the security state of the memory segments. Only the guest partition currently running in the non-secure side has its own memory segment configured as non-secure, while the remaining memory is configured as secure. The granularity of the memory segments, which is platform-dependent, limits the number of supported VMs. Moreover, since TrustZone-enabled processors only provide MMU support for single-level address translation, it means that guests have to know the physical memory segment they can use in the system, requiring relocation and consequent recompilation of the guest OS. Temporal isolation is achieved through a cyclic scheduling policy, ensuring that one guest partition cannot use the processor for longer than its defined CPU quantum. The time of each slot can be different for each guest, depending on its criticality classification, and is configured at design time. Time management is achieved by implementing two levels of timing: there are timing units for managing the hypervisor's time, as well as for managing the partitions' time. Whenever the active guest is executing, the timers belonging to the guest are directly managed and updated by the guest on each interrupt. For inactive guests, the hypervisor implements a virtual tickless timekeeping mechanism, which ensures that when a guest is rescheduled, its internal clocks and related data structures are updated with the time elapsed since its previous execution.

2.7.1 Identified Limitations

RTZVisor's main goal was to prove it was possible to run multiple guest OSes concurrently, completely isolated from each other, on TrustZone-enabled Arm processors without VE support. Despite achieving such a goal, RTZVisor still presented some limitations and open-issues. A list of the main identified limitations follow:

- Hypervisors are not magic bullets and they are also prone to incorrect expectations in terms of security. Guaranteeing a system is secure just by relying on a virtualization layer is not enough. These incorrect expectations probably come from the fact that a hypervisor provides separation and isolation, which positively impacts security. The problem is that security is much more than separation. Security starts from the onset, and hypervisors must be complemented with other security-oriented technologies for guaranteeing a complete chain of trust. The secure boot process is responsible for establishing a chain of trust that authenticates and validates all levels of secure software running on the device. In this sense, the integrity of the hypervisor at boot time is guaranteed.

- RTZVisor does not implement and enforce any existing coding standards. The use of coding standards is becoming imperative in modern security and safety-critical systems to reduce the number of programming errors and achieve certification.

- Although RTZVisor provides real-time support mainly by implementing efficient time services, these are still guest OS dependent and limited to a cyclic scheduling algorithm. The implementation does not allow for event-driven guests to preempt others, resulting in high interrupt latencies.

- The nature of embedded systems requires communication and interaction among the various subsystems. RTZVisor fails in this aspect by not implementing any kind of IPC facilities. All of its guests are completely isolated and encapsulated, having no mechanism to cooperate.

- Finally, and taking into account the previous point, RTZVisor provides no mechanisms for device sharing. Some kind of communication is needed for guests to synchronize, when accessing the same peripheral.

2.8 Model-Driven Engineering

Model-Driven Engineering (MDE) bases the software development on models, which are an abstract representation based on entities, and whose purpose is to provide a simplified view of the final system [61, 62]. This paradigm towards software development is also dubbed Model-Driven Software Development (MDSD). Motivating the use of this development paradigm were a list of problems [63]: the poor quality of developed software; difficulties in making software meeting specifications; projects going beyond schedule and budget; and high maintenance costs. Thus, the main advantage of MDE is that we create models that are loosely coupled with implemented software, starting with an approach that is closer to the problem domain, while making them easier to understand and maintain [64]. In [3, 65] it is advocated that by following this approach it is improved productivity, enhanced software quality and manageability of the development process. Also, that models become an integral part of the system description rather than just simple sketches, with equal importance as the developed software.

Despite that, on its own, models are of great importance as documentation artifacts; however, if they are just mere architectural representations, they do not achieve their full potential. As such, models are being combined with automatic source code generating tools, for example DSLs, towards software reuse. Therefore, MDE meets its full potential by being applied for automation purposes, which includes automatic generation of a complete programs, as well as automatic verification of models on a computer [65, 64].

Regarding the quality of models, in order to make them useful and effective, they must possess all five characteristics [66]:

- *Abstraction* - a model is always a reduced rendering of the system it represents;
- *Understandability* - a model must be described with expressiveness, providing a shortcut for reducing the required intellectual effort towards systems comprehension;
- *Accuracy* - a model most constitute a real representation of the modeled system;
- *Predictiveness* - a model enable us to scrutinize the non-obvious properties of a final system, by either executing the model or some type of formal analysis.

In order to ease software extendability, standards are often specified. As such, the Object Management Group (OMG) specified a standard framework towards

model conception [3, 66]. The OMG proposes the Model-Driven Architecture (MDA), which consists on set of directives following a four-layered approach, in which each layer provides a certain level of abstraction:*M0-layer* regards to real world objects, resulted of instantiation of M1 models; *M1-layer* constitutes a simplified view of real world objects, i.e., featured with abstractions; *M3-layer* defines meta-models, with rules and concepts applied to previous layers; and, lastly, *M2-layer*, i.e. the root for every model-driven development, describing concepts for the M2 layer.

Models can also be characterized as being prescriptive and descriptive. If they control, they prescribe reality; that is, they dictate conditions for forming reality and how reality should be like, once it has been constructed. It can also be said that such models are templates or schemas of reality. For example, a language's meta-model is a prescriptive representation of what a program should be like. On the other hand, the model is a mere representation of reality without any implications on its creation, i.e. the model simply describes it, being characterized as descriptive [67, 66].

2.8.1 Domain-Specific Language

A Domain-Specific Language (DSL) is a programming language that offers, through appropriate notations and abstractions, expressive power for a particular problem domain. Consequently, a DSL's number of constructs is usually small, however these tend to provide a good means for seamlessly describe solutions within the associated problem domain. In addition, these are very descriptive and easy to use, enabling domain experts, that do not even possess technical knowledge, to read and understand the implemented solution [68, 69]. These notions contrast with those regarded to a General-Purpose Language (GPL), that provides elemental, yet general elements to describe and implement solution regardless of their problem domain. However, to achieve the same result these usually require a higher development effort, as DSL usually aim at reducing the amount of required code to be written for a specific problem [69].

DSLs are usually declarative, and as such, these can be viewed as specification languages. However, this constitutes half of their value, so they should be supported by a DSL compiler that generates application code, often implemented into a GPL. This creates the opportunity for software reuse, avoiding the need for going through the implementation of template solutions, which is the main goal of software development paradigms, such as MDE, or Generative Programming [65, 70]. The latter is about modeling software systems belonging to the

same family recurring to software entities, and, given a set of requirements and systems description, automatically manufacturing a customizable instance [70]. Thus, achieving an overall faster development process.

Notwithstanding, adopting DSLs to approach software engineering involves both positive and negative considerations. In [68] it is stated that the use of DSLs provides a level of abstractions, which enables system designers to clearly understand, validate and modify a taken solution. Given the inherit level of expressiveness, these programs are self-documenting to a larger extent. Software reuse helps achieved higher levels of productivity, reliability, portability, and even software testing. On the other hand, it should taken into account the effort inherent to implementing and maintaining a DSL, as well as having to teach target users about the tool. In addition, DSL developers struggle with finding a proper scope, and balancing domain-specificity and GPL constructs. Finally, there is also the risk of efficiency loss compared with hand-coded software. So, before jumping into DSL development and use, it is necessary to have into consideration all aforementioned points.

The development process for a DSL, former to appropriately specifying its requirements and defining the scope, usually encompasses three steps [2, 71]:

- Creation of language model, that will result on the abstract syntax tree of the DSL. This includes specifying the language's elements as a class, defining how these relate to each other, originating a meta-model.
- As concrete syntax, visual symbols must be defined for each one of the languages element, which should be used by DSL's users.
- Implementation of a generator that converts the DSL program into a executable representation. This generator, must map the written concrete syntax to the elements of a abstract syntax, validating the formal specification of the DSL. If everything goes well, the executable code must be generated.

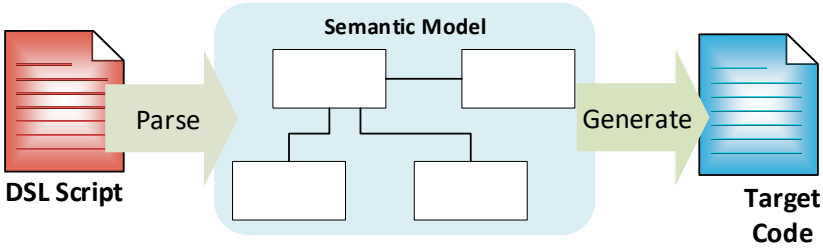


Figure 2.9: Overall DSL workflow [2].

The last bullet remits to the standard DSLs workflow, depicted in Figure 2.9. Regardless of the DSL, the depicted overall architecture constitutes the standard procedure, from a DSL script to the generated source code.

2.8.2 Component-based Software Engineering

Component-based Software Engineering (CBSE) advocates software systems description based on system's composition in well-defined building blocks. Inherently, this approach will incur into higher flexibility by separating stable parts of the system from each other. Components are seen as black-boxes that encapsulate functionality, accessible through well defined interfaces. One main point is that components are functional units, and they should be used according to a software architecture that defines interfaces and composition rules [72, 73].

Also, CBSE facilitates software reusability, since components can be used in varied forms, which inherently incurs into a better management of component variability [74, 68]. In addition, it also provides a good means for extensibility, for either application parts or even whole applications [73]. The provided separation of concerns is a good means for substitutability [75], i.e., if respecting the specified architecture, a given can be easily substituted, at least most of the system will remain functional. Components can also be assembled into subsets creating a new, broaden building block. However, there are compatibility concerns to be addressed, which regards to the capability that two entities possess to interact in a semantically meaningful way. Whenever assembling two components through the same interface, where there is one providing a certain service and the other referencing it, both should have the same understanding of the interface semantics and operation [76]. Figure 2.10 depicts a "dummy" component-based architecture.

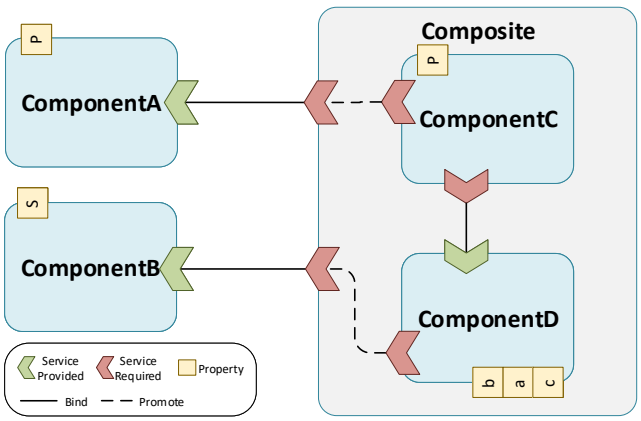


Figure 2.10: Dummy example of a component-based architecture.

An Architectural Description Language (ADL) provides language constructs that aim at system's architectural description, analysis and reuse of software architectures. ADL have been used to describe the structure of a system, which comprises software components, their externally visible properties, and the dependencies they have among them [77, 78]. To better describe the services provided among components, and to help specifying interfaces semantics, ADLs are usually paired with constructs specific of Interface Description Language (IDL). Interfaces are often described as being a collection of methods, events and attributes. Provided interfaces are those that contain operations implemented by a component, accessible to other components or to the component user, while required interfaces are those that contain operations used by the component itself [78].

The potential benefits of component-based development are as attractive in the domain of embedded systems as they are in other areas of the software industry. When building new applications from existing components it is not only necessary to ensure that they behave as expected, but also that properties with impact on non-functional requirements are composed correctly. One important characteristic of embedded systems is their characterization as real-time system, which means real-time properties like response time, worst-case execution time, among others, often need to be specified and validated prior to run-time [78, 21]. By leveraging the aforementioned ADL is possible to validate and verify the correctness of all these parameters, in addition to easily configure the final system. Components are a great means for control over functionality granularity, which is extremely desirable for describing embedded systems [9]. Also, IDL compilers translate a generic interface specifications into stub code that implements the actual communication, both for the provider and requirer components. This would incur into convenience and faster development, because the system developer is spared of the boring, yet bug-prone, development of communication code and custom communication protocols [79].

2.8.2.1 Component-based Modeling Solutions

Foremost, it is worth to mention that we consider that component-based architecture are a perfect fit for describing embedded systems architectures. This due to their simplicity, however, providing great gains in component reuse, composability and association through well define interfaces. Thus, this section approaches the description of component-based solutions that were on the genesis of the taken approach towards providing a means for microkernel-like systems modeling.

Research and engineering efforts that focus on establishing component-based software engineering disciplines specialized at embedded systems can be roughly divided into three categories [9]:

- The first category encompasses component-based architectures aimed at specific application domains such as field devices, consumer electronics, vehicular systems, etc;
- The second category encompasses component-based operating systems, where component modeling is applied to the operating system itself, or to provide component architectures at application level;
- The third category consists of middleware-based component models tailored for embedded and real-time systems with a particular focus on non-functional attributes.

In [72] the Component Architectures for microkernel-based Embedded Systems (CAmkES) project is described, and it falls into the second category, aiming at being used for describing both application and microkernel components, namely the L4 microkernel. The architecture provides a component model, standard interfaces and component definitions, component implementations, standard services, and support for various architectural patterns suited to embedded systems. It supports two types of components: *active* that maps towards a functional unit with execution thread; and *passive* that just encapsulates functionality. Regarding Interfaces, it provides means for describing interface with RPC semantics, *events* to be emitted or consumed by referencing components, in a publish subscribe fashion, and defines a *Dataport* that represent data to be transfered between components, either in a shared memory area or endpoints messaging. As a result, from established dependencies glue code, with communication functionality will be generated. CAmkES model encapsulates communication between components in explicit architectural elements called *connectors* and *connections*. A *connector* has a name and a list of interface types that it connects, allowing for 1-to-1, 1-to-many, many-to-1 and many-to-many relationships between interfaces.

Koala [68] is designed by Philips, and consists on an ADL aiming at software product-line development of consumer electronic devices, thus falling into the first category. It focus on restricted resource constraints and provides a lightweight component model. Components are units of design and reuse, that communicate through *interfaces*. For describing interfaces, Koala is featured with a small IDL supporting one type of interface with RPC semantics, in which procedure calls

are described recurring to C syntax. Interfaces are required and provided among components, through the *requires* and *provides* constructs. One *requires* can only be binded to *provides*, while one *provides* can be binded to zero or more *requires*. Static binding of components is supported and all invocations are hard-coded into components. Configurations within the component are also resolved recurring to the interface mechanism, i.e., the property owner requires a given interface through which the provider will configure the value on a binding. These are called diversity interfaces. On generation, the property value will be hard-coded within the requirer. There is also the concept of *switch* interface, that verifies a given condition on compile time and decides between providers accordingly. The *Component* construct is used to describe elemental components, as well as composites elements. Connections between its subcomponents are defined within it.

2.9 Ontology-Driven Software Development

An ontology defines the basic terms and relations comprising the vocabulary of a topic area, as well as the rules for combining terms and relations to define extensions to the vocabulary [80, 3]. Ontologies major purpose is not the creation of a vocabulary and taxonomies; it is the creation of a *knowledge base* towards knowledge sharing and reuse [66]. Within an ontology, there are three major components that constitute the knowledge base [80, 3]:

- *Classes* - Represent a group of entities with similar characteristics. Usually they are grouped in taxonomies through which inheritance can be applied. Inherently, this may result in stratified layers of classes within the ontology they were defined, allowing for gradations of meaning.

- *Relations/Properties* - These provide a means for creating associations between concepts. Ontologies usually contain binary relations, also dubbed as object properties, in which the first argument regards to the relation's domain, while the second argument is the range. Relations can also be used to represent concept attributes, in which the range will be a datatype (string, integer, etc...). The later type of relations are often dubbed as data properties.

- *Instances* - These are used to represent elements or individuals in an ontology, belonging to a certain group of individuals. Class membership is

either asserted via the type property or inferred based on property values, i.e., relationships with other individuals.

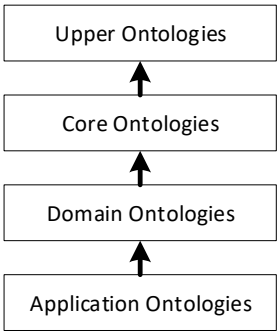


Figure 2.11: A simplified ontologies classification scheme [3].

In [3] is advocated a scheme for classifying ontologies that we also agree, and that despite being simple, still provides a good background for the presented work. Figure 2.11 depicts the referred framework, where each layer denotes a different level of abstraction. The lower the layer the higher the specificity it regards to: *Upper ontologies* define broad concepts, which belong to generic domains; *Core Ontologies* define concepts shared by a considerate number of domains, e.g. one to describe information systems; *Domain Ontologies* are confined to a specific domain, e.g. to describe user interfaces; *Application Ontologies* that specialize a domain-ontology with specific variants, e.g. the instance of buttons and text boxes, with specified properties, like sizes and so on. According to [81], generic ontologies suffice at fulfilling a specific purpose, however provide a good way for knowledge extension towards more concrete ontologies.

Ontologies can be represented in different forms, with different levels of abstraction, recurring to different tools. In a computer, they can be represented recurring to, for example, an XML-based file, or even some specific ontology representation language [66]. To developed work of this dissertation it was stated that it should be used The Web Ontology Language (OWL), from the Semantic Web framework (some of them where listed by [66, 3]). The former is an initiative by World Wide Web Consortium (W3C) that aims at facilitating data sharing, reuse and integration, towards the unification of knowledge bases [82]. OWL allows to specify classes based on property or cardinality restrictions, and supports boolean combination of classes and permits to declare properties as transitive, unique or the inverse of other properties [3]. One major benefit of knowledge representation using Semantic Web languages is their reliance in logic. Reasoning and logical inference facilitate integrity and consistency validation in a knowledge base, as well as the entailment of new statements.

These type of system representations can be applied in a multitude of scenarios in software engineering. Requirements engineering can benefit from ontologies for knowledge representation and process support, in addition to component reuse. Moreover, it can be integrated with software modeling tools, to facilitate software development process. Some authors [83, 82] have been emphasizing the similarities between MDA and semantic technologies. Both aim at providing an abstract system representation, namely a model. However, MDA-based languages lack the logic support existent in semantic technologies, as by semantic technologies lack the code generation features that MDA-based languages are paired with. Some approaches have been taken into combining both approaches, by performing a mapping between constructs in order to achieve a "semantic overlap", thus enhancing software modeling tools with the logic and reasoning benefits, such as: unambiguous domain models, higher consistency, interoperability enhancement, automated validation, and the reasoning support [82].

2.9.1 Example of an Ontology

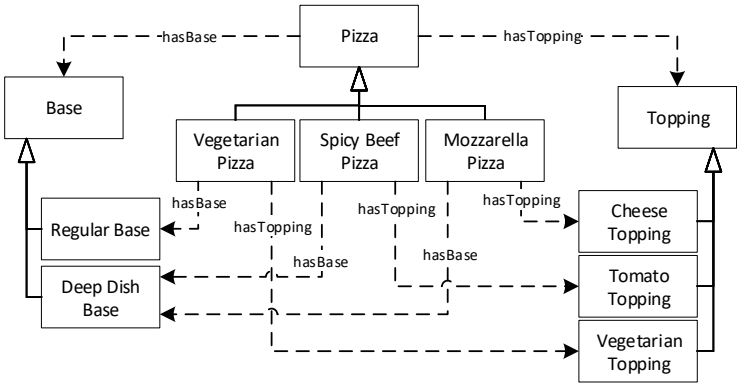


Figure 2.12: Simple example with a Pizza ontology.

Figure 2.12 depicts an ontology describing pizzas. The aim of this example is not to provide a sound ontology for describing this domain, is just to provide a better understanding on how concepts relate to each other within an ontology. The *Pizza* class has a base from class *Base*, and a topping from class *Topping*. By defining that *Pizza* has a topping, it is also being defined that all its derived classes also have a topping. For each different recipe it was limited the type of *Topping* and *Base*, by establishing relations with specific derived classes of *Topping* and *Base*. For example the *Mozzarella Pizza* has a *Cheese Topping* and a *Regular Base*. The mapping between concepts should remain coherent through out the

ontology, otherwise the reasoner will point the inconsistencies as errors, and it is important to have a consistent ontology in order to use it in a specific application.

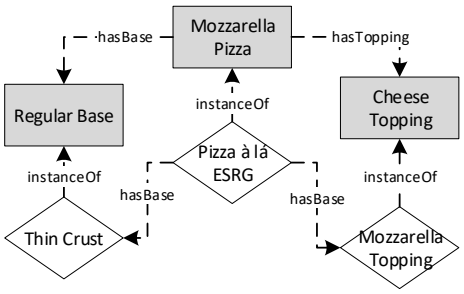


Figure 2.13: Individuals from Pizza ontology.

The established relations between classes should be transposed to the resulting individuals. Figure 2.13 depicts an ontology resultant from the previous one, that describes a specific recipe of a *Mozzarella Pizza*, mapping the resultant individuals according to what defined for classes, thus maintaining the ontology consistent.

2.10 SeML Infrastructure

The Semantically-enriched Modeling Language (SeML) is a modeling infrastructure, that is composed of an upper ontology, and a semantically enriched DSL. The overall architecture leverages the principles from ODSD towards its main goal, that is the transformation of implementation artifacts into a functional system that complies with a model description. The latter consists on requirements specification, and functional units performing some kind of behavior and system's properties.

Figure 2.14 depicts the overall architecture for the SeML infrastructure, and how these external elements relate with its inner structure. Firstly, a *Domain Expert* and a *Knowledge Engineer* are responsible for analyzing the system's domain, and capturing its knowledge, while identifying domain commonalities and variabilities. The captured knowledge should be translated into the domain ontology, written in OWL, basing its description on upper ontology's concepts, i.e., the newly created concepts must be derived from upper ontology's classes and relations.

The aforementioned DSL has almost no fixed keywords and very little constructs. Also, it is not featured with all the necessary capabilities for system's description. The latter's process consists, grossly, in two steps:

1. The instantiation of individuals from the domain ontology classes;

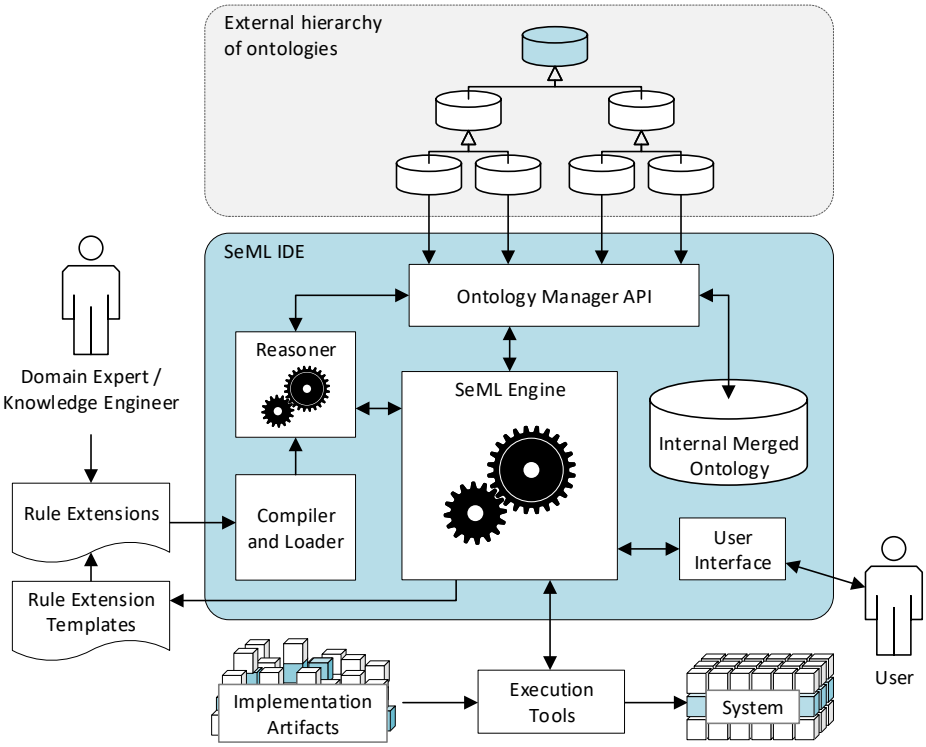


Figure 2.14: SeML infrastructure overall architecture.

2. Connecting the aforementioned individuals through object properties, and asserting properties values.

The first step will originate an input ontology, to be used within the DSL. The latter capture the input ontology's concepts to draw its tokens, allowing for a simplistic syntax in a triple statement (like in example from Section 2.10.2). Despite being reliant on the input ontology's concepts, the DSL does not provide a means for instantiating individuals. Its purpose is restricted to be the means for connecting input ontology's concepts. Despite the fact that used individuals must be instances of the component class deriving from the upper ontology, these might be related via any object property the user might include his domain ontology.

The infrastructure will generate a system ontology, that can be executed at any time as long as it is valid. The interpreter will scan the model and invoke the methods of the execution tools. The implementation artifacts should be also provided, in a manner that complies with the implementation ontologies. Some tools are included within the infrastructure to perform basic operations, such as checking the existence of specific data blocks or replacing textual tags with parameter values, component references or user-defined data types. Additional tools may also be developed to cope with the demands of each system specificity. These can be compiled and loaded by the infrastructure at runtime. The called

tools can check the integrity of existing artifacts, modify them and create new ones as demanded by the SeML interpreter.

The infrastructure supports both OWL and SWRL rules for knowledge description, that whenever verified can prompt some behavior. There are some built-in behaviors, however, others can be implemented in java and provided to the infrastructure. This useful to verify that the model complies with specified requirements, that are difficult to characterize just by recurring to OWL axioms, and inheritance from classes.

2.10.1 Upper Ontology

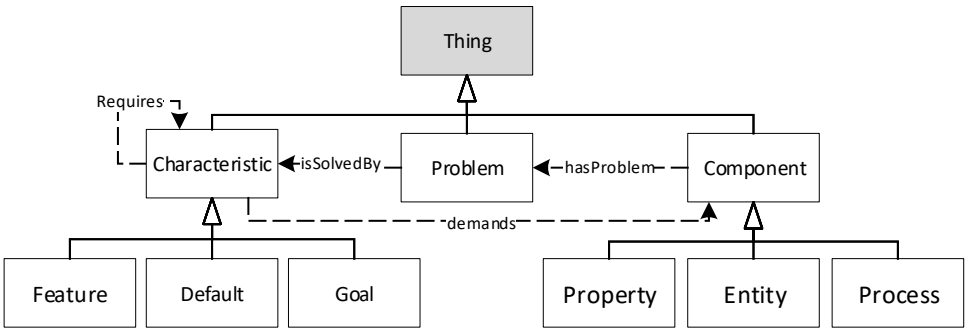


Figure 2.15: SeML upper Ontology.

Figure 2.15 depicts the Semantically-enriched Modeling Language (SeML) upper ontology, that provides the base concepts for domain descriptions. All direct subclasses of *Thing* (owl class) are pairwise disjoint, which means that their sets of individuals do not intersect. The most import concept is the *Component*, which encompasses terms that allow the description of different viewpoints, and works as the bridge that connects the infrastructure to any system description taxonomy, acting as the root class, analogously to the role of *Thing* in OWL. It possesses three derived classes, chosen to normalize the meta-models of different domain experts: *Process* that denotes a functional behavior, e.g., GPL functions, methods or sequential instruction blocks; *Entity* represents structures such as programs, files, modules, etc; and *Property* that stands for constants or variables of a system, including primitive or composite data types. If a *Component* poses an obstacle to the integrity of a domain, it has a *Problem*, for example a vulnerability.

Another important concept is the *Characteristic*, that denotes predefined abstract terms that influence the whole model. It was divided into three terms *Feature* and *Goal*, which are functionally interchangeable but allow the normalization of ontologies by comprising attributes and desired results of a model, respectively.

Finally, the *Default* concept covers the subset of system characteristics which are not optional, but rather required. Thus, these provide a useful means for specifying and validating non-functional, that can be used to trigger rules, prompting some kind of behavior. Another thing to have in mind is that SeML models require the existence of static individuals. These are tagged with a SeML specific OWL annotation, and represent fixed components in the model, working as entry points for the model descriptions.

2.10.2 Simple Example

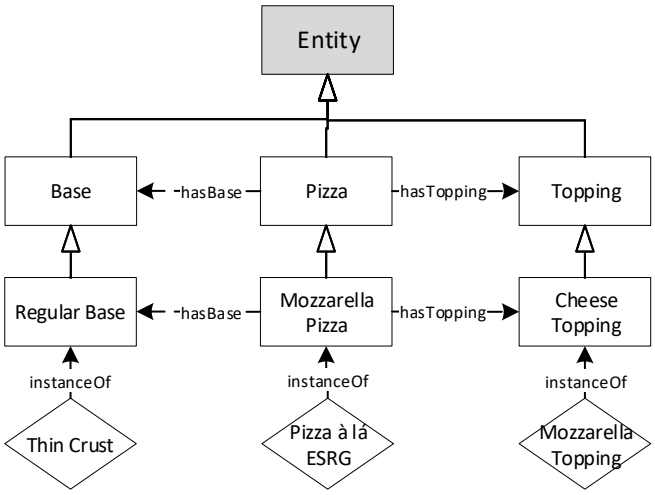


Figure 2.16: Using SeML with pizza ontology.

To provide a better understanding on how modeling with SeML looks like, a little example is presented, leveraging the pizza ontology from Section 2.9.1. Firstly, the concepts from the pizza ontology must be aligned with those from SeML's upper ontology. Figure 2.16 depicts an excerpt of the resulting domain ontology. The concepts depicted from the pizza ontology were all aligned with the *Entity* class. Recurring to an ontology editor, the depicted individuals must be created. Also, there should exist at least one static individual, that would be the *Pizza a la ESRG*.

Listing 2.1: Code for binding pizza properties.

```
1 import "PizzaOntology.owl"
2
3 Pizza-a-la-esrg hasBase ThinCrust
4 Pizza-a-la-esrg hasTopping MozzarellaTopping
```

Listing 2.1 depicts the code for the binding between individuals. Once again, it provides a simplistic scenario; however it serves its purpose, which is show how

a SeML program should look like. This result in the same ontology depicted in Section 2.9.1. Once again, this code is provides a simplistic view of what should be accomplished. There is one static individual, whose relations must be resolved in order to achieve a valid SeML model. Thus, *Pizza-a-la-esrg* is binded with *ThinCrust* and *MozzarellaTopping* to achieve a model representation compliant with what specified in the domain ontology.

Chapter 3

μRTZVisor Architecture

The following section describes the μRTZVisor architecture, which was a collaborative effort by two students. The focus of this dissertation is on the IPC infrastructure, that should be designed securely, advocating a secure-by-design approach that is reflected on the deployment of access-control mechanisms. The other student focus on the remaining core functionality, namely focusing on spacial and temporal isolation. As such, the design goals are presented, followed by development process overview, and architectural description, providing some functional detail of each module within the kernel. There is also an implementation section, that solely provides a structural and behavioral view of the system, however, confined to the artifacts that directly map to the scope of this thesis.

3.1 System Overview

μRTZVisor is based on a refactoring of RTZVisor, designed to achieve a higher degree of safety and security. In this spirit, we start by anchoring our development process in a set of measures that target security from the onset. First, we made a complete refactoring of the original code from C to C++. The use of an object-oriented language promotes a higher degree of structure, modularity and clarity on the implementation itself, while leveraging separation of concerns and minimizing code entanglement. Kernel modules have bounded responsibilities and only interact through well-defined interfaces, each maintaining its internal state while sharing the control structure of each partition. However, we apply only a subset of C++ suitable to embedded systems, removing features such as multiple inheritance, exception handling or Run-Time Type Information (RTTI), which are error prone, difficult to understand and maintain, as well as unpredictable and inefficient from a memory footprint and execution perspective. In addition to the fact that C++ already provides stronger type checking and linkage, we reinforce

its adoption by applying the MISRA (Motor Industry Software Reliability Association) C++ coding guidelines. Due to the various pitfalls of the C++ language, which make it ill-advised for developing critical systems, the main objective of the MISRA C++ guidelines is to define a safer subset of the C++ language suitable for use in safety related embedded systems. These guidelines were enforced by the use of a static code analyzer implementing the standard.

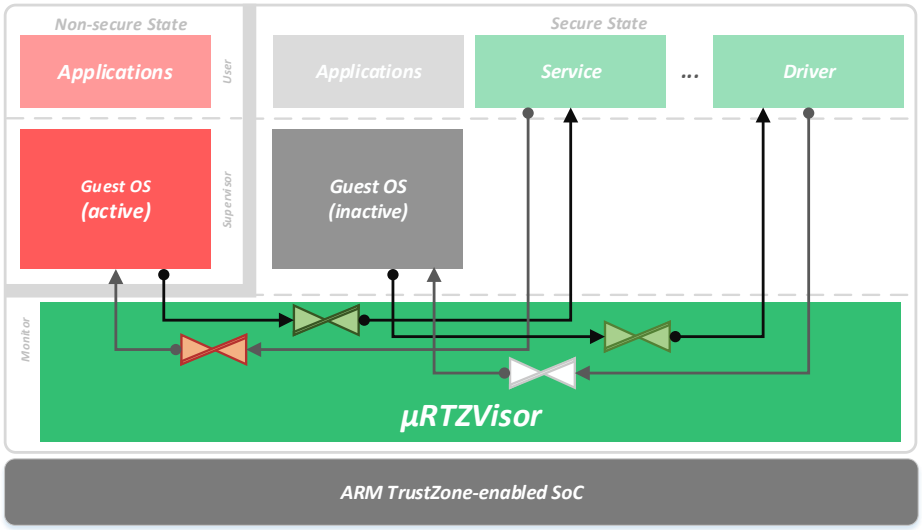


Figure 3.1: μRTZVisor architectural overview.

The main feature of μRTZVisor is its microkernel-like architecture, depicted in Figure 3.1. Nevertheless, we don't strive to implement traditional microkernel virtualization, which, given its para-virtualization nature, imposes heavy guest source-code modification. We aim at gathering those ideas that benefit security and design flexibility, while persevering the capability to run nearly unmodified guest OSes. TrustZone-enabled processors have two virtual Central Processing Units (CPUs), by providing a secure and non-secure view of the processor, and extend this partitioning to other resources such interrupts and devices. Consequently, guests can make full use of all originally intended privileged levels, being allowed to directly configure assigned system resources, manage their own page tables and directly receive their assigned interrupts. Thus, guest OSes only need to be modified if they are required to use auxiliary services or shared resources that rely on the kernel's IPC facilities. Notwithstanding, the lack of a two-level address translation mechanism imposes a segmented memory model for VMs. Hence, guest OSes need to be compiled and cooperate to execute in the confinement of their assigned segments. This limitation is augmented by the fact that segments provided by the TZASC are typically large (in the order of several MB) and must

be consecutively allocated to guests, which leads to high levels of internal fragmentation. In addition, the maximum possible number of concurrent guests is limited by the total number of available memory segments, which varies according to the platform's specific implementation. Nevertheless, despite the small the number of segments, this memory model is likely to suffice for embedded use-cases that usually require a small, fixed number of VMs according to deployed functionalities.

Multi-guest support is achieved by multiplexing them on the non-secure side of the processor, i.e., by dynamically configuring memory segments, devices or interrupts of the active partition as non-secure. Inactive partition resources are set as secure and by saving and restoring the context of CPU, co-processor and system control registers, which are banked between the two worlds. An active guest that attempts to access secure resources triggers an abort exception directly to the hypervisor. As a security preventive measure, given that guests share cache and Translation Lookaside Buffer (TLB) infrastructures, these must be flushed when a new guest becomes active. Otherwise, the entries of the previous guest, which are marked as non-secure, could be accessed without restriction by the incoming one.

μRTZVisor privilege code runs in monitor mode, the most privileged level in TrustZone-enabled processors, having complete access and configuration rights over all system resources. This layer strives to be a minimal Trust Computing Base (TCB), implementing only essential infrastructure to provide the virtual machine abstraction, spatial and temporal partitioning, and basic services such as controlled communication channels. The kernel's design aims for generality and flexibility so that new functionality can be added in a secure manner. For this, it provides a heterogeneous partition environment. As described above, coarse-grained partitions based on the memory segmentation model are used to run guest OSes. In addition, partitions running in secure user mode are implemented by managing page tables used by the MMU's secure interface, which allows for a greater degree of control over their address spaces. Secure user mode partitions are used to implement extra functionality, which would typically execute in kernel mode in a monolithic system. They act as server tasks that can be accessed through RPC operations sitting on the IPC and scheduling infrastructure. For example, shared device drivers or virtual network infrastructures can be encapsulated in these partitions. Herein lies the main inspiration from microkernel ideas. Non-essential services are encapsulated in these partitions, preventing fault-propagation to other components. Hence, they can be untrusted and developed by third-parties, incorporating only the TCB of other partitions that depend on

them. Although these kind of services could be implemented in VMs running in the non-secure world, rendering worthless the extra core complexity added to the kernel, implementing them as secure world tasks provides several benefits. First, running them on a secure virtual address space eliminates the need for the relocation and recompilation and reduces the fragmentation inherent to the segmented memory model. This facilitates service addition, removal or swapping according to guests' needs and overall system requirements. At the same time, it enables finer-grained functionality fault-encapsulation. Finally, both the hypervisor and secure tasks always run with caches enabled, but, since caches are TrustZone-aware, there is no need to flush them when switching from a guest partition to a secure world task due to a service request via RPC, which significantly improves performance.

A crucial design decision relates to the fact that partitions are allocated statically, at compile-time. Given the static nature of typical embedded systems, there is no need for partitions to create other partitions or to possess parent-child relations and some kind of control over one another. This greatly simplifies the implementation of partition management, communication channel and resource distribution, which are defined and fixed according to the system design and configuration. This idea is further advanced in the next few paragraphs.

To achieve robust security, fault-encapsulation is not enough and the principle of the least authority must be thoroughly enforced. This is done at a first level by employing the aforementioned hardware mechanisms provided both by typical hardware infrastructure (virtual address translation or multiple privilege levels) and the TrustZone features that allow control over memory segments, devices and interrupts. Those are complemented by a capability-based access control mechanism. In this way, all the interactions with the kernel, i.e., hypercalls, become an invocation of an object operation through a capability. This makes the referencing of a resource by a non-authorized partition conceptually impossible if they do not own a capability for it. Given that the use of capabilities provides fine-grained control over resource distribution, system configuration almost fully reduces to capability assignment, which shows to be a simple, yet flexible mechanism.

Built upon the capability system, this architecture provides a set of versatile IPC primitives, a crucial aspect of the microkernel philosophy. These are based on the notion of a port, constituting an endpoint to and from which partitions read and write messages. Given that these operations are performed using port capabilities, this enables system designers to accurately specify the existing communication channels. Furthermore, the notion of reply capabilities, i.e., port

capabilities with only the send rights set, which can only be used once, and that are dynamically assigned between partitions through IPC, is leveraged to securely perform client-server type communications, since they remove the need to grant servers full-time access to client ports. Aiming at providing the maximum design flexibility, our architecture provides both synchronous and asynchronous policies. Synchronous primitives are also combined with scheduling functionalities for efficient RPC communication. This service provision by means of RPC overlaps with our approach, but it focuses only on providing a Trusted-Execution Environment (TEE) and not on a flexible virtualized real-time environment.

This architecture categorically differs from classical TrustZone software architectures, which typically feature a small OS running in secure supervisor mode that manages secure tasks providing services to non-secure partitions and that only execute when triggered by non-secure requests or interrupts. This service provision by means of RPC overlaps with our approach, but it focuses only on providing a Trusted-Execution Environment (TEE) and not on a flexible virtualized real-time environment. No such OS exists following this approach, since the hypervisor directly manages these tasks, leaving the secure supervisor mode vacant. This partial flattening of the scheduling infrastructure allows for the direct switch between guest client and server partitions, reducing overhead, and for secure tasks to be scheduled in their own right to perform background computations. At the same time, given that the same API is provided to both client and server partitions, it homogenizes the semantics of communication primitives and enables simple applications that show no need for a complete OS stack or large memory requirements to execute directly as secure tasks. In addition, in some microkernel-based virtualization implementations, the VMs abstraction is provided by user-level components [54], which, in our system, would be equivalent to the secure tasks. This encompasses high-levels of IPC traffic between the VMs and the guest OS and a higher number of context-switches. Given the lightweight nature of the VMs provided by our system, this abstraction directly provided at the kernel level, which, despite slightly increasing TCB complexity, significantly reduces such overhead.

Besides security, the architecture places strong emphasis on the real-time guarantees provided by the hypervisor. Inspired by ideas proposed in [4], the real-time scheduler structure is based on the notion of time domains that execute in a round-robin fashion and to which partitions are statically assigned. This model guarantees an execution budget for each domain which is replenished after a complete cycle of all domains. Independently of their domain, higher priority partitions may

preempt the currently executing one, so that event-driven partitions can handle events such as interrupts as quickly as possible. However, the budget allocated to these partitions must be chosen with care according to the frequency of the events, to not be exhausted, delaying the handling of the event until the next cycle. We enhance this algorithm with a time-slice donation scheme [43] in which a client partition may explicitly donate its domain's bandwidth to the target server until it responds, following an RPC pattern. In doing so, we allow for the co-existence of non-real time and real-time partitions, both time and event-driven, while providing fast and efficient communication interactions between them. All related parameters such as the number of domains, their budgets, partition allocation and their priorities are assigned at design time, providing once again a highly flexible configuration mechanism. For the kernel's internal structure, we opted for a non-preemptable, event-driven execution model. This means that we use a single kernel stack across execution contexts, which completely unwinds when leaving the kernel, and, when inside the kernel, interrupts are always disabled. Although this design may increase interrupt and preemption latencies, which affect determinism by increasing jitter, the additional needed complexity to make the kernel fully preemptable or support preemption points or continuations would significantly increase the system's TCB.

Finally, it is worth mentioning that the design and implementation of the μRTZVisor was tailored for a Zynq-7000 SoC and is heavily dependent on the implementation of TrustZone features on this platform. Although the Zynq provides a dual-core Arm Cortex-A9, the hypervisor only supports a single-core configuration. Support for other TrustZone-enabled platforms and multicore configurations is out of the scope of this work.

3.1.1 Partition Manager

The main role of the partition manager is to guarantee consistency and integrity of the partitions execution context, namely their CPU state. This module also encapsulates the list of Partition Control Block (PCB), which encapsulate the state of each partition and which partition is currently active. Other kernel modules must use the partition manager interfaces to access the currently active partition and entries of the PCB for which they are responsible.

As previously explained, two types of partitions are provided: non-secure guest partitions and secure task partitions. While, for task partitions, state is limited to user mode CPU registers, for guest partitions, the state encompasses

banked registers for all execution modes, non-secure world banked system registers, and co-processor state (currently, only the system configuration co-processor, CP15, is supported). The provided VM abstraction for guest OSes is complemented by the virtualization structures of the GIC's CPU interface and distributor as well as of a virtual timer. These are detailed further ahead in Sections 3.1.5 and 3.1.4.

The partition manager also acts as the dispatcher of the system. When the scheduler decides on a different partition to execute, it informs the partition manager which is responsible for performing the context-switch operation right before leaving the kernel. In addition to saving and restoring context related to the aforementioned processor state, it coordinates the context-switching process among the different core modules, by explicitly invoking their methods. These methods save and restore partition state that they supervise, such as a memory manager method to switch between address spaces.

The partition manager also implements the delivery of asynchronous notifications to task partitions, analogous to Unix-style signals. This is done by saving register state on the task's stack and manipulating the program counter and link registers to jump to a pre-agreed point in the partition's executable memory. The link register is set to a pre-defined, read-only piece of code in the task's address space that restores its register state and jumps to the preempted instruction. The IPC manager uses this mechanism to implement the event gate abstraction (Section 3.1.6).

3.1.2 Capability Manager

The Capability Manager is responsible for mediating partitions access to system resources. It implements a capability-based access control system that enables fine-grained and flexible supervising of resources. Partitions may own capabilities, which represent a single object, that directly map to an abstract kernel concept or a hardware resource. To serve its purpose, a capability is a data structure that aggregates owner identification, object reference and permissions. The permissions field identifies a set of operations that the owning partition is allowed to perform on the referenced object. In this architecture, every hypercall is an operation over a kernel object; thus, whenever invoking the kernel, a partition must always provide the corresponding capability.

Figure 3.2 depicts, from a high-level perspective, the overall capability-based access control system. Each partition has an associated virtual capability space,

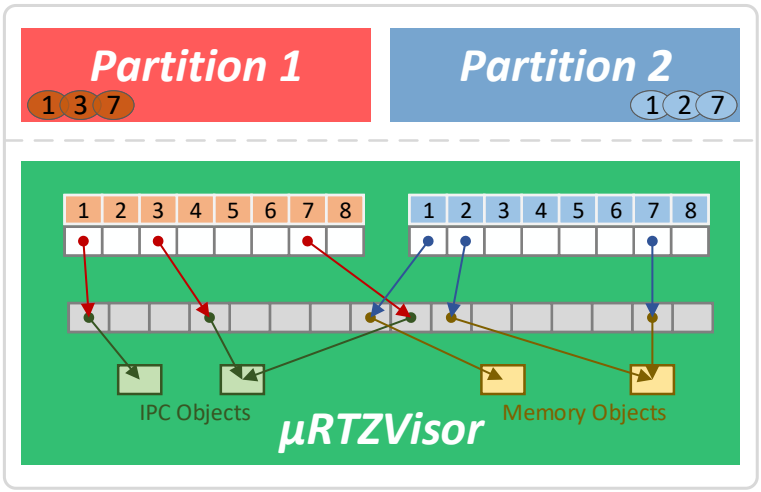


Figure 3.2: Capability-based access control system overview.

i.e., an array of capabilities through which those are accessed. Whenever performing a hypercall, the partition must provide the identifier for the target capability in the capability space, which is then translated to a capability on a global and internal capability pool. This makes it conceptually impossible for a partition to directly modify their capabilities or operate on objects for which it does not possess a capability, as only the capabilities on its capability-space are indirectly accessible. Moreover, for every hypercall operation, the partition must specify the operation it wants to perform along with additional operation-specific parameters. At the kernel's hypercall entry point, the capability is checked to ensure the permission for the intended operation is set. If so, the capability manager will redirect the request to the module that implements the referenced object (e.g., for an IPC port it will redirect to IPC manager), which will then identify the operation and perform it.

At system initialization, capabilities are created and distributed according to a design-time configuration. Some capabilities are fixed, meaning that they are always in the same position of the capability space of all partitions, despite having configurable permissions. These capabilities refer to objects such as the address space, which are always created for each partition at a system's initialization. For capabilities associated with additional objects defined in the system's configuration, a name must be provided such that it is unique in a given partition's capability space. During execution, partitions can use this name to fetch the associated index in their capability space.

μRTZVisor also provides mechanisms to dynamically propagate access rights by invoking *Grant* and *Revoke* operations on capabilities, which, as for any other operation, must have the respective rights set in the permissions field. The *Grant*

operation consists of creating a derived capability, which is a copy of the original one, with only a subset of its permissions, and assigning it to another partition. The operation's recipient is notified about it, through an IPC message in one of its ports, which contains the index for the new capability in its capability space. To perform the *Grant* operation, the granter must specify the granting capability, the subset of permissions to grant, which must be enclosed in the original ones, and the port to which the notification will be delivered. This means that the granter must possess a capability for a port owned by the recipient. A derived capability may further be granted, giving rise to possible complex grant chains. Each derived capability is marked with a grant id, which may later be used to perform the revoke operation. In turn, the revoke operation withdraws a given capability from its owner, and can only be performed by one of the partitions in a preceding grant chain. The revocation process propagates through the donation chain. A revoked capability is maintained in a zombie state in the capability space, until it is used again. When the owning partition tries to use it, it will receive an error and the position will be freed so that it can be used again. Finally, there is a special type of capability, called a one-time capability, that can only be used once. The first time a partition uses this capability it is erased from the partition's capability space. These are also referred to as reply capabilities in the context of the IPC manager, and are leveraged to perform secure RPC communication. This is further detailed in Section 3.1.6.

3.1.3 Memory Manager

At system initialization, the memory manager starts by building the address spaces for all partitions from meta-data detailing the system image layout. For guest partitions, this encompasses figuring out which consecutive memory segments must be set as non-secure when the guest is active, i.e., those for which they were compiled to run on, and loading them to these segments. On Zynq-based devices, memory segments have a granularity of 64 MB, which might lead to high levels of internal fragmentation. For example, for a 1 MB or 65 MB guest OS binary, 63 MB of memory is left unused. During this process, if it is detected that two guests were built to run by sharing the same memory segment, the manager will halt the system, since the spatial isolation requirement cannot be guaranteed. From the remaining free memory, the memory manager will build the virtual address spaces for secure tasks, which currently only use 1 MB pages instead of the available 4 K pages. In addition, in the current implementation, no more memory can be allocated by tasks after initialization, so partition binaries

must contemplate, at compile-time, memory areas to be used as stack and heap, according to the expected system needs.

The hypervisor code and data are placed in the bottom 64 MB memory segment, which is always set as secure. Address spaces always contemplate this area as kernel memory and may extend until the 1 GB limit. Since we manage tasks' page tables, their virtual address space always starts immediately after kernel memory, making the recompilation needed for guest partitions, which always have a confined, but direct view of physical memory unnecessary. Above the 1 GB limit, the address space is fixed for all partitions, contemplating a peripheral, CPU private and system register area, and, at the top, an On-Chip Memory (OCM) region of 4KB segments, which we call slices and use for guest shared memory as detailed below.

This module manages two page tables used by the secure interface of the MMU. The first implements an identity mapping and is used when a guest partition is currently active. The second is used when a task partition is active and is updated each time a new task is scheduled. Since it is expected that secure service partitions do not need to occupy a large number of pages, only the individual page table entries are saved in a list structure. The extra-overhead of updating the table at each task context restore was preferred, instead of keeping a large and variable number of page tables and only switching the page table pointer, reducing the amount of memory used by the hypervisor.

Partition memory mappings are kept active while the kernel executes, in order to avoid manual address translation for the kernel to access partition space when reading information pointed to by hypercall arguments. At the same time, the memory manager offers services to other system modules that enable them to verify the validity of the hypercall arguments (i.e., if data pointed by these arguments is indeed part of the partitions address space), read and write to and from address spaces others than the currently active one, and perform address translations when needed.

Upon address space creation, a capability is inserted in the partitions' capability spaces that enables them to perform operations over it. These operations are exclusively related to the creation and mapping of objects representing some portion of physical memory and that support shared memory mechanisms. We distinguish two different types of memory objects, page and slice objects, always represented and manipulated through capabilities and shared among partitions using grant and revoke mechanisms. Although both types of objects may be created by guest and task partitions, only slice objects may be mapped by guests,

since guest address space control is exclusively performed through TrustZone segmentation mechanisms. For example, a guest that needs a task to process its data may create a page object representing the pages containing that data using its address space capability. It then grants the page object capability to the task. The task uses this capability to map the memory region to its own address space and processes the data. When the task is done, it signals the client partition, which revokes the capability for the page object, automatically unmapping it from the task's address space. The same can be done among tasks, and among guests, although, in the latter case, only using the slice memory region.

TrustZone-aware platforms extend secure and non-secure memory isolation to both the cache and memory translation infrastructure. Nevertheless, a cache or TLB marked as non-secure may be accessed by non-secure software, despite the current state of memory segment configuration. Hence there is a need to flush all non-secure cache lines when a new guest partition becomes active, so that they cannot access each other cached data. This is also performed for non-secure TLB entries since a translation performed by a different guest might be wrongly assumed by the MMU. This operation is not needed when switching secure tasks since TLB entries are tagged. Although it is impossible for non-secure software to access secure cache entries, the contrary is possible by marking secure page table entries as non-secure, which enables the kernel and secure tasks to access non-secure cache lines when reading hypercall arguments or using shared memory. This, however, puts forth coherency and isolation issues, which demands maintenance that negatively impacts performance. First, it becomes imperative for guest partition space to be accessed only through a memory manager's services so that it can keep track of possible guest lines pulled to the cache. When giving control back to the guest, these lines must be flushed to keep coherency in case the guest is running with caches disabled. This mechanism may be bypassed if, at configuration time, the designer can guarantee that all guest arguments are cached. Moreover, if a guest shares memory with a task, further maintenance is required, in order to guarantee that a guest with no access permission for the shared region cannot find it in the cache. This depends on the interleaved schedule of task and guests as well as the current distribution of shared memory objects.

3.1.4 Device Manager

The job of the device manager is similar to that of the memory manager since all peripherals are memory mapped. It encompasses managing a set of page tables and configuring TZPC registers to enable peripheral access to task

and guest partitions, respectively. Each peripheral compromises a 4 KB aligned memory segment, which enables mapping and unmapping of peripherals for tasks, since this is the finest-grained page size allowed by the MMU. The peripheral page tables have a transparent and fixed mapping, but, by default, bar access to user mode. When a peripheral is assigned to a task, the entry for that device is altered to allow user mode access at each context switch. For all non-attributed devices, the reverse operation is performed. An analogous operation is carried out for guests, but by setting and clearing the peripheral's secure configuration bit in a specific TrustZone register. If a peripheral is assigned to a partition, it can be accessed directly, in a pass-through manner, without any intervention of the hypervisor.

At initialization, the device manager also distributes device capabilities for each assigned device according to system configuration. Here, when inserting the capability in a partition's capability space, the manager automatically maps the peripheral for that partition, allowing the partition to directly interact with the peripheral without ever using the capability. However, if partitions need to share devices by granting their capabilities, the recipient must invoke a map operation on the capability before accessing the device. The original owner of the capability may later revoke it. This mechanism is analogous to the shared memory mechanism implemented by the memory manager.

3.1.5 Interrupt Manager

The interrupt manager works on the assumption that only one handler per interrupt exists in the system. These may reside in the kernel itself or in one of the partitions. In the first and simplest case, kernel modules register the handler with the manager at initialization time, which will be called when the interrupt occurs. The only interrupt used by the kernel is the private timer interrupt that is used by the scheduler to time-slice time domains. If the interrupt is assigned to one partition in the configuration, the capability for the interrupt will be added to its capability space with the grant permission cleared. Partition interrupts are always initially configured as disabled and with the lowest priority possible. The details on how a interrupt is configured and handled depends on the type of partition it is assigned to.

Task partitions cannot be granted access to the GIC, since, if so, by running on the secure world, they would have complete control over all interrupts. All interactions with the GIC are thus mediated by the hypervisor by invoking capability operations, such as enable or disable. These partitions receive interrupts as

an IPC message. Hence, before enabling them, they must inform the interrupt manager on which port they want to receive it. The kernel will always keep task interrupts as secure, and when the interrupt takes place, it will place a message in the task's port and disable it until the task signals its completion through a hypercall. Since this process relies on IPC mechanisms, the moment in which the task takes notice of the interrupt will completely depend on what and when it uses the IPC primitives and on its scheduling priority and state. It is allowed for a task to set a priority for an interrupt, but this is truncated by an upper-bound established during system configuration and the partitions' scheduling priority.

Guest partitions can directly interact with the GIC, but a virtual GIC is maintained in the virtual machine. While a guest is inactive, its interrupts are configured as secure but disabled. Before dispatching the guest, the hypervisor will restore the guest's last interrupt configurations as well as a number of other GIC registers that are banked between worlds and may be fully controlled by the guest. Active guests receive their interrupts transparently when they become pending. Otherwise, as soon as the guest becomes active, the interrupts that became pending during its inactive state are automatically triggered and are received normally through the hardware exception facilities in the non-secure world. As such, at first sight, a guest has no need to interact with the GIC or the interrupt manager through capabilities as task partitions do. However, if the capability has the right permission set, the guest can use it to signal the kernel that this interrupt is critical. If so, the interrupt is kept active while the guest is inactive, albeit with a modified priority, according to the partition's scheduling priority and a predefined configuration. Regardless of which partition is running, the kernel will receive this interrupt and manipulate the virtual GIC to set the interrupt pending as would normally happen. In addition, it will request for the scheduler to temporarily migrate the guest partition to time domain-0 (if not there already), so that it can be immediately considered for scheduling before its time domain becomes active and handle the interrupt as fast as possible. Although the worst case interrupt latency persists as the execution cycle length minus the length of the partition's time domain, setting it as critical increases the chance of it being handled earlier, depending on the partition's priority.

3.1.6 Port Manager

Communication is central to a microkernel architecture, and as μRTZVisor inherits microkernel-like principles this should be reliable. Our approach implements the notion of *ports*, which are kernel objects that act as endpoints through

which information is read from and sent to in the form of messages. Communication mechanisms are built around the capability-system, in order to have ICF and enforce the principle of least authority. In order to perform an operation over a *port*, a partition must own a capability referencing that same *port*, with the permissions for the needed operations set.

Port operations may work in a synchronous or asynchronous style, and are further classified as blocking or non-blocking. Despite our focus on security, we also want to offer the flexibility provided by synchronous communication, which enables fast RPC semantics for efficient service provision. As such, the μRTZVisor provides *port* operations for both scenarios, combining synchronous operations with the scheduling infrastructure, explained in Section 3.1.9. The most elemental *port* operations are *Send* and *Receive*, where the former is never blocking, but the latter may be. Other operations are composed as a sequence of these elemental operations, in addition to other services provided by the scheduler, for time-slice donation, or by the capability manager for one-time capability propagation. Table 3.1 summarizes all IPC primitives over available *port*s. As shown in the table, there are two types of receive operations, one blocking and the other non-blocking. In the first case, the partition will stall and wait for a complementary send operation to happen on the respective *port* to resume its execution. The second one will check for messages in the *port*'s message queue, which stores them in first-in-first-out (FIFO) arrival order. If the queue is empty, an error value is returned.

Table 3.1: Port operations characterization, i.e., if it is synchronous or asynchronous and either blocking or non-blocking.

Port Operations	Synchronous	Asynchronous	Blocking	Non-Blocking
Send	x	x	-	x
RecvNonBlock	-	x	-	x
RecvBlock	x	-	x	-
SendReply	x/-	x/x	-	x
SendReplyDonate	x/x	x/-	x	-
ReceiveDonate	x	-	x	-

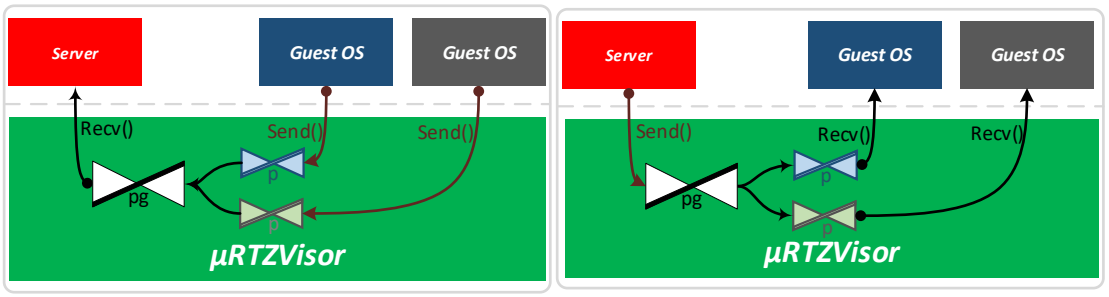
If the *Send* operation follows a receive that is blocking on the recipient partition, it will happen in a synchronous style, as both meet in a specific point in time, with the latter being set to an active state. However, the copy procedure does not happen directly between address spaces, because of the way memory is managed. Otherwise, the communication will be asynchronous, which means that a message will be pushed into the *port*'s message buffer. Both *SendReply*

operations will perform an elemental send followed by an elemental receive. In addition, they rely on services from the capability manager to grant a capability to the recipient partition. These capabilities are erased after being used once, and are dubbed reply-capabilities. When performing an operation with the -Donate suffix, the partition is donating its execution time-slice to the recipient partition, and it blocks its execution until receiving a response message from that same partition. More details about the donation process will be given in Section 3.1.9. In addition, when a given partition asynchronously sends a message, the recipient partition may receive an event, if the *port* has been properly configured, to notify it about the message arrival. In addition, we extended the *port* operations to support a notify operation. This triggers an event at the recipient guest. As event structures are more simple than messages, this enables a simpler way to communicate in scenarios where messages are overkill. More details about events in μRTZVisor provided in section 3.1.8.

All communication objects must be specified at design time, which means that partitions are not allowed to dynamically create *port*s. In addition, the respective capabilities should be carefully distributed and configured, since, depending on its permissions, it may be possible for a partition to dynamically create new communication relations through the capability *Grant* operation, or by *SendReply* operations. Therefore, all possible existing communication channels are, at least implicitly, defined at configuration time. Although partitions may grant *port* capabilities, if no relation for communication exists between partitions, they will never be able to spread permissions. Thus, designers must take care to not unknowingly create implicit channels, since isolation is reinforced by the impossibility of partitions to communicate with each other, when they are not intended to.

We later extended the concept of *port* to create a *port group*, which is another kernel object that also works as a destination for message passing operations. It basically enables the aggregation of multiple *ports* under the same abstraction, enabling messaging operations for all of them with only one system call. Figure 3.3 illustrates the idea behind *portgroup*s. However, *port group* operations solely encompass the *send*, *notify* and *receive* blocking and unblocking. All abstracted *port*s must be owned by the *portgroup* owner, in order to both receive operations to be valid, otherwise, an error will be prompted. This also means, that for these ports there should exist a capability referencing it within the partition's capability space. Whenever performing a *receive* over a *portgroup*, the received message will include the index of capability space, whose capability references the recipient port for the received message. This incurs into flexibility whenever implementing server

VMs, as connection oriented information can be maintained within the kernel.



(a) *Receive* operation. **(b)** *Send* operation.

Figure 3.3: Operations using a *port group* as endpoint.

The *port* abstraction hides the identity of partitions in the communication process. Partitions only read and write messages to and from the endpoints but do not know about the source or the recipient of those messages. This approach is safer since it hides information about the overall system structure that might be explored by malicious partitions. However, *port*s can be configured as privileged and messages read from these *port*s will contain the ID of the source partition. This enables servers to implement connection-oriented services, which might encompass several interactions, in order to distinguish amongst their clients and also to associate their internal objects with each one.

3.1.7 Lock Manager

For synchronization purposes, there are also specific kernel objects called locks, which functionally can work as a *mutex* or *spin-lock*. A partition must try to acquire it whenever accessing some shared resource, for example shared memory. The other partition must verify the state of the *lock*, before proceeding with an access to the resource that is being protected. If the *lock* has already been acquired, the partition should wait. Otherwise, the lock becomes acquired to the partition and it can proceed with the operation. The *lock* will be used as a *mutex*, it partition's state changes to blocked until the *lock* gets unlocked again.

A partition may try to acquire a *lock* by performing one of two versions of a *Acquire*: *AcquireBlocking* and *AcquireNonBlocking*. The first changes partition state to blocked in case the object has already been acquired by other partition, originating a scheduling point. The latter will return the execution to the invoker, thus working like a *spin-lock*. To release the object, there is an *Unlock* operation. Additionally, there is the *AcquireDonate*, which will perform a donation to the partition that acquired the *lock*. Thus, accelerating the chances to resolve this

dependency over the resource. As well as *ports* and *portgroups*, *lock*s must be specified at design time, with respective capabilities.

3.1.8 Event Manager

Events are asynchronous notifications that alter the partition's execution flow. For the secure tasks' partitions, they are analogous to Unix signals and are implemented by the partition manager described in Section 3.1.1. For guest partitions, they resemble a normal interrupt, to not break the illusion provided by the virtual machine. In addition, it would be extremely difficult to implement them as signals, given that the hypervisor is OS-agnostic and has no detailed knowledge about the internals of the guest. Hence, services from the interrupt manager (Section 3.1.5) are used to inject a virtual interrupt in the virtual machine. To receive events, guests must configure and enable the specified interrupt in their virtual IPC. In this way, events are delivered in a model closer to the VM abstraction, and OS agnosticism is maintained. Partitions interact with the event infrastructure through a kernel object called event gate. To receive events, partitions must configure the event gate and associate it with ports that will trigger an event upon message arrival. To lower implementation complexity, each partition is assigned a single event gate, which will handle events for all ports in a queued fashion. In addition, a capability with static permissions is assigned to each partition for its event gate at system's initialization. The aforementioned permissions encompass only the *Configure* and *Finish* operations. The *configure* operation allows partitions to enable events, and also to specify the memory address of a data structure where event-related data will be written to upon event delivery, and that should be read by the owning partition to contextualize the event.

3.1.9 Scheduler

The presented approach for the μRTZVisor scheduler merges the ideas of [15, 43]. It provides a scheduling mechanism that enables fast interaction between partitions, while enabling the coexistence of real-time and non-real-time applications without jeopardizing temporal isolation, by providing strong bandwidth guarantees.

The scheduler architecture is based on the notion of a time domain, which is an execution window with a constant and guaranteed bandwidth. Time domains are scheduled in a round-robin fashion. At design time, each time domain is assigned an execution budget and a single partition. The sum of all execution

budgets constitutes an execution cycle. A partition executes in a time domain, consuming its budget until it is completely depleted, and the next time domain is then scheduled. Whenever a complete execution cycle ends, all time budgets are restored to their initial value. This guarantees that all partitions run for a specified amount of time in every execution cycle, providing a safe execution environment for time-driven real-time partitions.

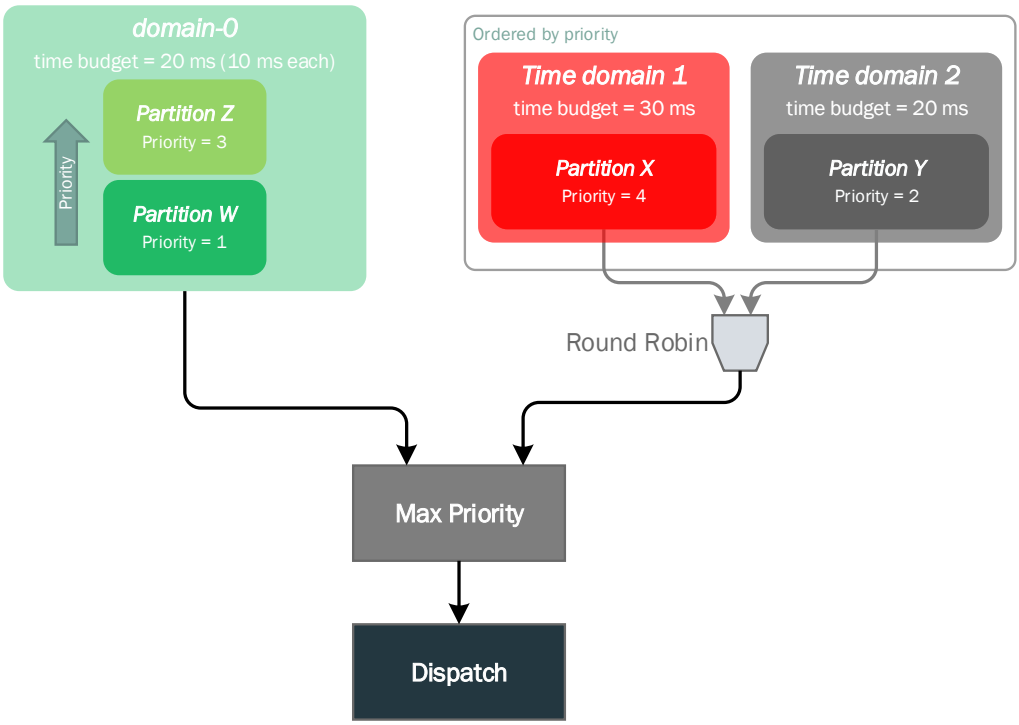


Figure 3.4: Overview of the scheduling algorithm.

The scheduler allows that multiple partitions may be assigned to a special-purpose time domain, called domain-0. Inside domain-0, partitions are scheduled in a priority-based, time-sliced manner. Furthermore, domain-0's partitions may preempt those running in different domains. It is necessary to mention that any partition is assigned a priority, which only has significance within the context of domain-0. At every scheduling point, the priorities of the currently active time domain's partition and the domain-0's highest priority ready partition are compared. If the latter possesses a higher priority, it preempts the former, but consuming its own domain's (i.e., domain-0's) time budget while executing. The preemption does not happen, of course, if domain-0 itself is the currently active domain. Figure 3.5 presents an example scenario containing two domains: time domain 1, assigned with partition X; and time domain 2, assigned with Partition Y, in addition to domain-0, assigned with partitions Z and W. Time domain 1 is first in line, and since partition X has higher priority than the ones in domain-0, it

will start to execute. After its time budget expires, a scheduling point is triggered. Time-domain 2 is next, but since domain-0's partition Z possesses higher priority than partition Y from time domain 2, Z is scheduled consuming domain-0's budget. At a certain point, partition Z blocks, and since no active partition in domain-0 has more priority than domain 2's Y, the latter is finally scheduled and executes for its time domain's budget. The next scheduling point makes domain-0 the active domain, and the only active partition, W, executes, depleting domain-0's budget. When this expires, a new execution cycle begins and domain 1's partition X is rescheduled.

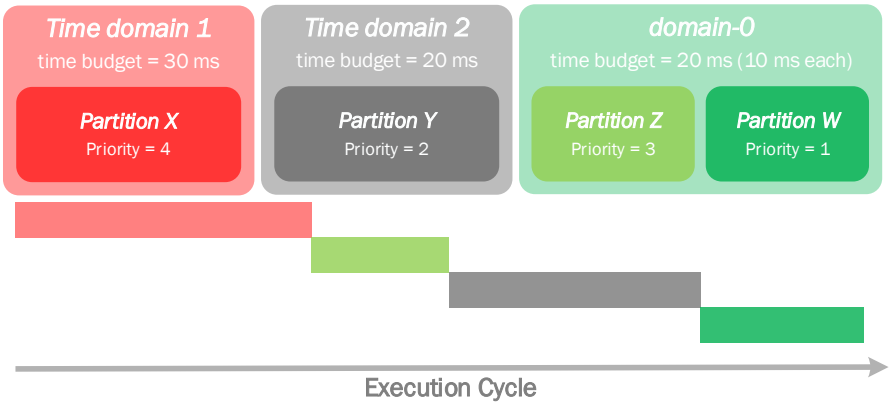


Figure 3.5: Example of an execution cycle, given a set of time domains with their own partitions and respective configuration.

Aiming at providing fast interaction between partitions, some IPC operations are tightly coupled with specific scheduling functionalities. Section 3.1.6 highlights a number of IPC operations that rely on the scheduler: the *ReceiveBlocking* , and the ones with -donate suffix (i.e., *SendReplyDonate* and *ReceiveDonate*). The *ReceiveBlocking* operation results in changing the partitions state to blocked, and then scheduling the next ready partition from domain-0 to perform some background work. Nevertheless, it keeps consuming the former domain's time budget, since it prevails as the active time domain. Hence, by blocking, a partition implicitly donates its execution time to domain-0. The following scheduling point will be triggered according to one of three scenarios: (a) domain-0's internal time slice expires, which results in scheduling the next highest priority partition from domain-0; (b) the active time domain's budget expires, and the next time domain becomes active; (c) the executing partition sends a message to the active time domain's partition, which would change its state to ready and result in scheduling it right away. In summary, upon blocking, a partition remains in this state until it is unblocked by receiving a message on the port it is hanging. If an execution cycle

is completed without a change in the partition's state, partitions from domain-0 are scheduled once more in its place.

The -donate suffixed operations require a more intricate operation from the scheduler, and by invoking them, a partition is explicitly donating its time budget to the recipient port's owner. Hence, it will block until it has the created dependency resolved, i.e., it blocks waiting for the message's recipient to send its response. In case the donator has a higher priority than the donatee server, the latter will inherit the former's priority, augmenting the chances of it to execute and to resolve the dependency sooner. Considering a scenario where two partitions donated their time to low-priority servers running in domain-0, the server that inherits the higher priority will execute first when domain-0's becomes active, or even preempt another time domain's partition, which it previously wouldn't preempt. This enables services to be provided in a priority-based manner, i.e., maintaining the priority semantics of the requesting partitions. This priority inheritance mechanism also mitigates possible priority inversion scenarios. A partition relying on another one, and donating its time domain without any other intervener, constitutes the simplest form of a donation chain. However, a donate operation may be performed to or from a partition that is already part of a donation chain in a transitive manner, constituting a more intricate scenario. Whatever partition is at the tail of the chain, it will be the one to execute whenever one of the preceding partitions is picked by the scheduler. Notwithstanding, only the one following a given partition at the donation chain is able to restore its state to ready, by sending a message to the port on which the donator is waiting for the response to its request. This synchronous, donation-based mechanism is prone to deadlock scenarios, which in our approach is synonymous with a cycle in the donation chain. As such, a mechanism to detect deadlocks is provided, which aimed at being as lightweight as possible, considering its pervasive execution in every donation procedure.

Although not imposed by the implementation, this design was devised so that guest partitions are placed in common time-domains and secure task partitions are placed in domain-0. Since the idea of secure tasks is to encapsulate extended kernel services or shared drivers, these can be configured with lower priorities, executing according to guest needs and based on the latter's priority semantics. In addition, this models allows for the coexistence of event-driven and background partitions in domain-0, while supporting guests with real-time needs and that require a guaranteed execution bandwidth. For example, a driver with the need for

a speedy reaction to interrupts could be split in two cooperating tasks: a high priority task acting as the interrupt handler, which upon interrupt-triggering would message the second lowest priority task that interfaces other partitions, executing only upon a guest request. Even though a mid-priority client guest could be interrupted by the interrupt handler, its execution time within the cycle is guaranteed. Due to possible starvation, only the tasks that act as pure servers should be configured with the lowest possible priorities in domain-0. Other partitions that may be acting as applications on their own right or may have the need to perform continuous background work should be configured with a middle range priority. It is worth mentioning that the correctness of a real-time schedule will depend on time domain budgets, partition priorities and on how partitions use communication primitives. Thus, while the hypervisor tries to provide flexible and efficient mechanisms for different real-time constraints to be met, their effectiveness will depend on the design and configuration of the system.

3.2 Implementation

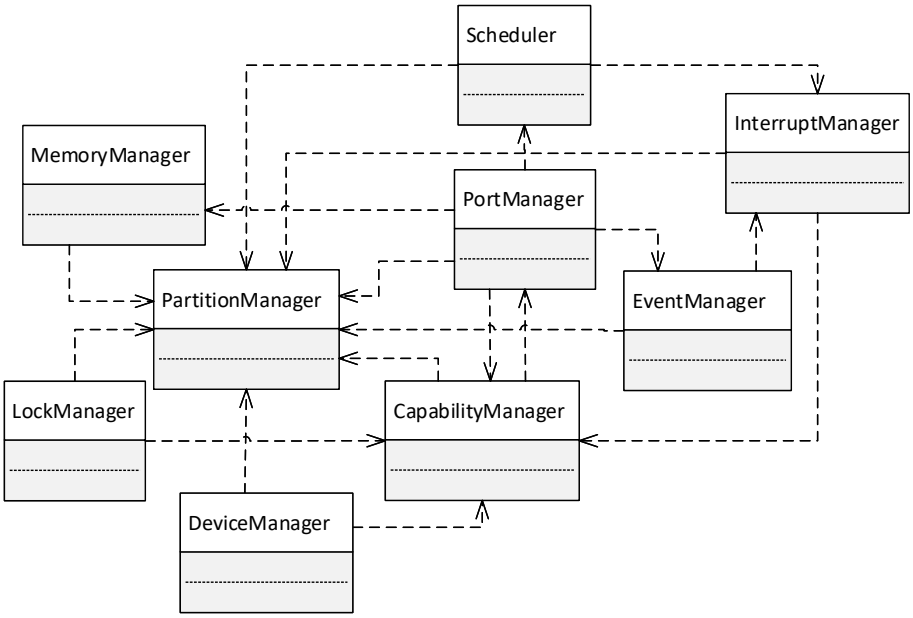


Figure 3.6: μRTZVisor UML diagram: overview of implemented architecture.

Figure 3.6 depicts an UML diagram presenting the main component managers μRTZVisor, and existing dependencies between them. All depicted classes are implemented as singletons. The diagram presents an high-level view where some kernel objects and utility classes are omitted for simplifying the view, however

some of them may be unveil through out this section. At the center of diagram are the *Partition Manager* and *Capability Manager*. The first handles PCBs which may its change state according to operations that might happen on kernel objects related to them in some way, while the latter manages capability objects.

Every system call happens through the same interface, by invoking the *SMC* instruction. Priorly, general purpose registers (r0 to r2) are used to pass arguments to inside the system call according to ARMv7 ABI. As aforementioned, partitions must specify the capability referencing the kernel object that implements the interface they intend to access, in addition to operation-specific parameters. We developed user-level APIs that ease this process.

The handler that attends an SMC interrupt (dubbed *smc_handler*) is tightly coupled with the *CapabilityManager*, given capabilities importance in performing system calls. In this sense, the *smc_handler* is "friend" of the *CapabilityManager* in order to access private properties within the manager. Every manager operating on kernel objects registers a callback that is invoked upon an operation in one of its responsibility, for which the execution is redirected after permissions checks are effectuated. To enhance security, all parameters passed inside the kernel go through a sanity verification, to ensure the value makes sense considering the purpose and meaning for the variable, in addition to verifying that every data pointer passed are indeed pointing to a valid address. These classes are implemented using templates, and are used In order to store kernel resources, we implemented two utility classes *TzPool* and *TzList*. These classes are implemented using templates, and are used to store objects within the kernel, like port messages, capabilities, page tables... Every time one of these cells of memory within these structure is not used anymore, it is cleaned to avoid leaks. Lastly, all managers have associated configuration files, that are a pair of .c and .h files. At μRTZVisor main function (Listing 3.1), each manager initialization function is invoked, traversing configuration structures and creating kernel objects and respective capabilities accordingly. In addition, each manager registers one callback at a table, to which execution flow should be redirected to attend services of its respective objects.

Listing 3.1: Main function with initialization of all managers.

```
1 int32_t main(){
2   /*Managers initialization*/
3   tzvisor::TzListBase::nodePool.Init();
4   tzvisor::PartitionManager::GetInstance().Init();
5   tzvisor::TzCapabilityManager::GetInstance().Init();
6   tzvisor::MemoryManager::GetInstance().Init();
7   tzvisor::TzEventManager::GetInstance().Init();
8   tzvisor::TzPortManager::GetInstance().Init();
9   tzvisor::InterruptManager::GetInstance().Init();
```

```
10    tzvisor::DeviceManager::GetInstance().Init();
11    tzvisor::TzMutexManager::GetInstance().Init();
12    tzvisor::Scheduler::GetInstance().Init();
13
14    tzvisor::Scheduler::GetInstance().Schedule();
15    tzvisor::assembly::ns_world_invoke();
16    /* Should never  return here */
17    while(1);
18 }
```

3.2.1 Access-Control

The access-control functionalities, in addition to the *capability manager*, are implemented in the classes depicted in Figure 3.7. The *Object* class encapsulates information used to reference an object instance to which the access is being mediated and will have its address stored in a 32-bit integer, which is used to fetch the object within its respective pool. The remaining property (dubbed managerInfo) stores the information regarding object type and manager, concatenated in one variable. The manager information is extracted within the *smc_handler* function, in order to figure which manager callback should be invoked, while the object type info is used within the callback to differentiate objects operated within the same manager. This is useful, for example, in the case of the *PortManager* to differentiate from a port and a port group object, as both of them support namesake operations, however, with different behaviors. All public methods *Object* class help to extract these informations whenever necessary.

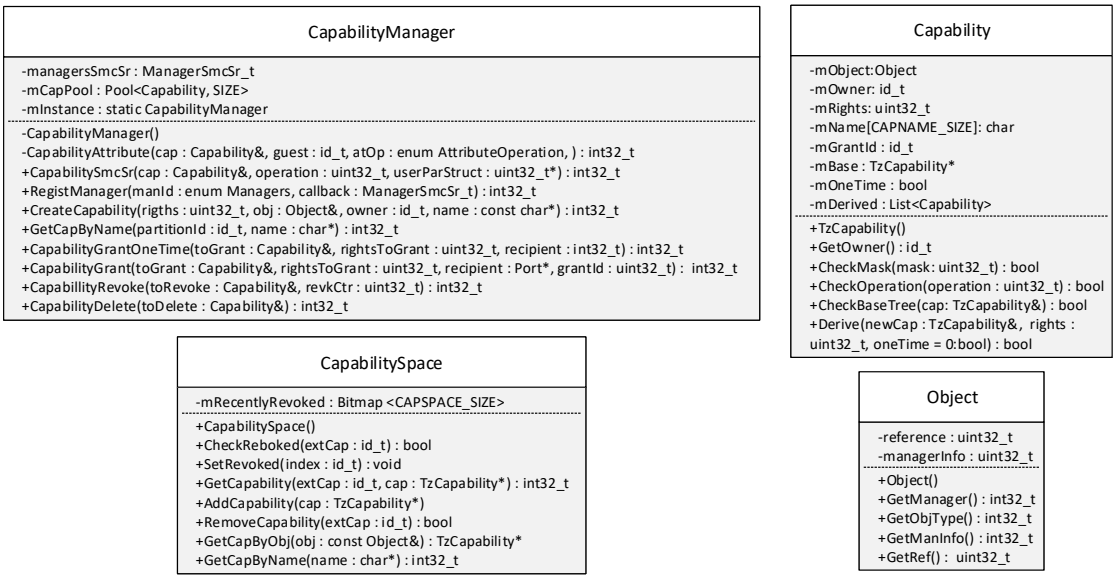


Figure 3.7: Classes that encapsulate access control functionality.

The *CapabilitySpace* class is responsible for managing capabilities owned by the same partition, storing their references in an array. Thus, it encompasses functions to add and remove capabilities, as well as to search them within the capability space, based on name and reference. This is useful, for example, to figure the port that which triggered an event in an event gate. Whenever it runs out of capability slots, a message is sent to the owning partition's kernel port, enabling the recovery either by removing one capability that is not needed, or by using a one-time capability from a client-server communication. Lastly, the *Capability* class encapsulates all elemental operations regarding access control. Permissions are stored in a 32-bit integer, that is used as a bitmap, where each bit represents an operation over the referenced object. These bits have different meanings within a given object type context, however, the grant and revoke operations can also be mapped in this bitmap, occupying the same position regardless of the object type. In addition, *Capability* class provides functionality to check permissions on a mask, to validate a set of operations, and on a single operation. It provides methods for deriving the capability, as well as properties to support the proper grant and revoke of a capability.

3.2.1.1 SMC Handler

As aforementioned, *smc_handler* method is configured as being the handler for *SMC* exception. Thus, this is the entry point for every single operation to happen inside the kernel. As such, its mode of operation is completely transversal to every type of object and manager. Listing 3.2 presents this function's algorithm.

Firstly, the arguments passed from the user-space are fetched: the capability reference from the partition's *capability space*, the operation to be performed and lastly, a pointer used to pass operation-specific parameters. As previously explained, this process is made according to ARMv7 ABI. Following, the capability in use is fetched from the owner capability space, and then checked for permissions regarding the operation to be performed. If the permission is set within the capability, is checked if it is a *grant* or a *revoke*. This because both operations are transversal to all objects and managers, and should approached as a special case within the flow of the *smc_handler*. When the operation is a *grant*, some additional verifications are performed regarding owner permissions to communicate with the recipient partitions, given that it will receive a message informing about the *grant* occurrence. If the operation is not a *grant* nor a *revoke*, from the in-use *capability* is fetched the manager identification. This identification should be used within the manager initialization, for registering a specific callback, to which the

execution should be redirect in case of an operation over one of the manager's objects, on a callback table.

```
Extract function parameters from registers (capability
 reference, operation, other paramenters);
Fetch Capability;
Get operation;
if operation in capability/is valid then
    if operation is a grant then
        if are parameters valid then
            Perform grant();
        else
            return error;
        end
    else if operation is a revoke then
        Perform revoke();
    else
        Get manager from capability;
        Call manager callback;
    end
else
    return error;
end
if is a one-time capability then
    Erase capability;
end
```

Listing 3.2: Algorithm of the *smc_handler*.

3.2.1.2 Grant and Revoke

A *grant* operation, whose algorithm is presented in Listing 3.3, consists on the propagation of permissions over an object. A partition can only provide access to a subset of permissions it has over an object, whose capability has the *grant* bit set. In doing so, the kernel will create another capability and insert it on the recipient's capability space, after ensuring the operation is indeed valid. In addition, it adds the new capability to the list of derived capabilities within the original one, and it sets the pointer for the base capability within the derived one. Whenever invoking the *grant* operations, the running partition specifies a grant ID, that is going assigned to the derived capability, filling *mGrantID* field (check field on Figure 3.7). Each *Capability* class possesses a list to store its derived capabilities. The *mGrantID* is used to identify this capability in case of a revoke operation. The ID is not unique, thus the same can be assigned to more than one derived capability. Lastly, there is a type of message (section 3.2.2.1) used to

communicate the recipient's about the grant, and what is the index of the new capability within its capability space.

```
Get capability to recipient port;
if Has permissions to send then
    if Permissions to grant contained in original capability then
        Update new capability;
        Fill base capability in derived one;
        Extend derived list;
        Invoke manager callback;
        Send message with capability;
    else
        return error;
    end
else
    return error;
end
```

Listing 3.3: Algorithm of the *Grant* operation.

The *revoke* occurs always after a performed *grant*. Listing 3.4 and 3.5 presents the algorithm for the operation for a capability to be revoked. Listing 3.4 presents the algorithm for searching capabilities to be revoked in the derived list of the original capability. This is done recurring to the referred ID, and once there is a match, the function the actually does the revoke is executed Listing 3.5. It removes the capability from its respective capability space, and then from the global capability pool. Whenever revoking a capability all its derived capabilities will be also revoked. This is achieved by invoking the revoke using recursivity over that same function.

```
Get grantID from parameters;
Set iterator to the beginning of derived capabilities list;
while Iterator is different from the end of derived list do
    while Current grantID different from received as paramenter do
        Iterate;
    end
    Invoke recursive function for revoke;
end
```

Listing 3.4: Algorithm of the *revoke* operation.

A change of access over a kernel object may require the respective manager to perform additional operations. For example, upon *grant* or *revoke* of capability referencing a memory object requires the *MemoryManager* to execute a mapping or unmapping functionality, or even to interrupt the operation for sanity-check.

Thus, every manager must also register a callback for *grant* and *revoke* operations, being able to maintain the overall infrastructure coherent and safe.

```
Set iterator to the beggining of derived list from current
 capability;
while iterator is from the end of derived list do
    Invoke recursive call;
    Pop next element from list ;
end
Invoke manager's revoke callback ;
Remove capability from capability space;
Erase capability ;
```

Listing 3.5: Algorithm of the recursive function from the *revoke* operation.

3.2.2 IPC

As explained in Section 3.1.6, μRTZVisor implements IPC mechanisms for synchronization and message passing. For both mechanisms there are different types of kernel objects, managed by different subsystems within the kernel: the *PortManager* for message passing functionalities, and the *LockManager* for synchronization purposes.

3.2.2.1 Message Passing

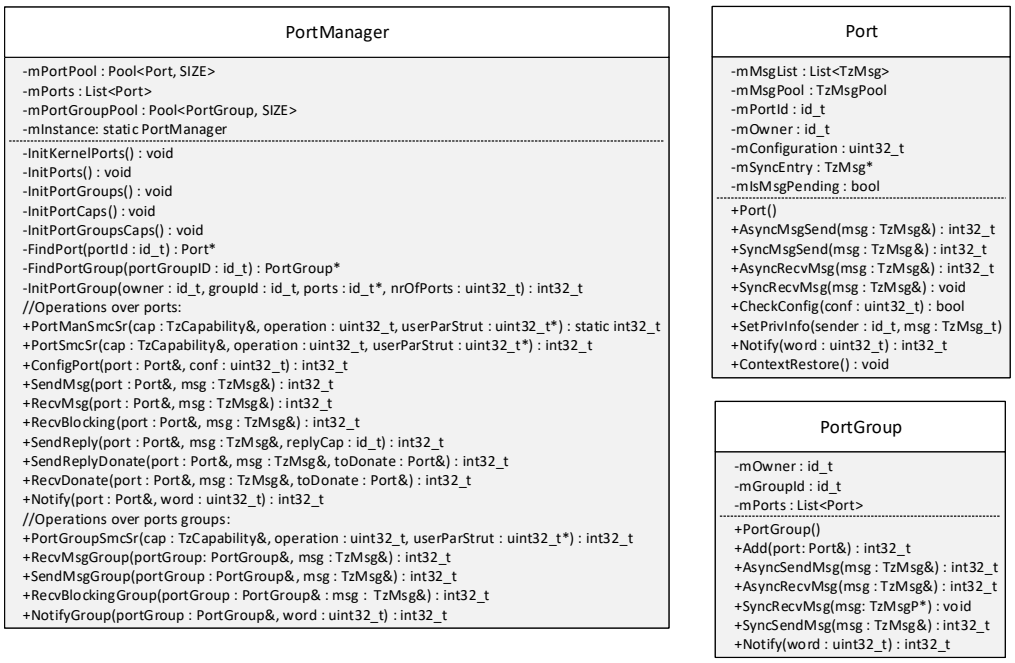


Figure 3.8: Classes that encapsulate message-passing functionality.

Figure 3.8 depicts all classes that deal with message passing functionality. *Port* and *PortGroup* are the kernel objects used for message passing, over which partitions perform their system calls, and whose access is mediated through *PortManager*. All operations of this type will go through method *PortManSmcSr*, which, according to the object type, will redirect to a callback that handles the respective object type.

Following, it is explained how each operation over *Port*s and *PortGroup*s is implemented, starting with a brief description of message's structure.

Listing 3.6: μRTZVisor's message structure.

```
1 struct TzMsg_t {
2   enum TzMsgType {
3     NORMAL=0, NORMAL_REPLY, CAP_GRANT, KERNEL
4   } mType;
5
6   int32_t mPayloadSize;
7
8   struct PropCap_t {
9     int32_t mObjectType;
10    int32_t mExtCap;
11  } mPropCap;
12
13  union payload_t {
14    uint32_t mWords[PAYLOAD_SIZE / sizeof(uint32_t)];
15    uint8_t mBytes[PAYLOAD_SIZE];
16    uint32_t interrupt;
17  } uPayload;
18
19  struct PrivInfo_t {
20    id_t sender;
21  } mPrivInfo;
22 };
```

Listing 3.6 shows μRTZVisor's message structure. The implemented structure encompasses fields to propagate all needed information, which takes into accounts every use-case scenario. Messages may be propagated under different circumstances, as such, we specified a field to identify the messages purpose and context on the *TzMsgType* enumeration. The supported message types are: *NORMAL* that specifies a simple message propagated between partitions; *NORMAL_REPLY* which extends a simple message with a one-time capability; *CAP_GRANT* intended to notify a recipient of a grant operation, providing the respective index within the capability space; and *KERNEL* in case a message comes from the kernel, e.g., to notify about a full capability space. In order to check the index of a propagated capability in both of the aforementioned scenarios, the recipient should check the *struct PropCap_t* typed field, that provides not only the capability position within the respective capability space, but also

the type of object it refers to. For the server to maintain information about the state of a given communication, it can use the *struct PrivInfo_t* type field, which is undefined by default; however, at initialization a port may be configured to provide client's specific information. Lastly, the *union_t payload* is intended to be filled by the sender entity with the information to be sent.

Message Sending The following described implementation, whose algorithm is presented in Listing 3.7, presents the flow of an elemental send operation described in Section 3.1.6. This operation is also invoked whenever performing a compound operation that includes a message sending, over a port object.

```
Extract parameters;
Set message type;
Sanity check on message size;
if Recipient's state is active then
    Invoke port's asynchronous send;
else if Recipient's state is blocked then
    Invoke port's synchronous send;
    Unblock recipient;
    Invoke schedule;
else if Recipient's state is donated then
    if Donated to current sender then
        Invoke port's synchronous send;
        Give back donation;
    else
        Invoke port's asynchronous send;
    end
else
    return error;
end
```

Listing 3.7: Algorithm of the *Send* operation.

Firstly, all the necessary information is decoded, like the message pointer, type and recipient port reference. Then, the recipient's execution state is verified, and it can be *ACTIVE* when executing the normal behavior, *BLOCKED* when waiting for any message in a given *port* or *port group*, or *DONATED* when waiting for a specific partition to send a message in a specified *port*. After this verification, the message can be inserted in the port's message queue if the recipient's is both *ACTIVE*, or *DONATED* and the receiver did not donate to the current

sender. Otherwise, the message will be cached within the port, if the recipient is *BLOCKED*, or *DONATED* to the current sender. Then, the recipient will have its state changed to active, and in the latter scenario, it will invoke services from the scheduler to give back the donation.

There is also a send operation specific for port groups, that will invoke the described procedure for the elemental send over every port within it. In addition, both *SendReply* and *SendReplyDonate* operations, described in Section 3.1.6 also invoke the aforementioned send routine. Both also rely on services from the *Capability Manager* to grant a capability, however, with the one-time feature. The *SendReplyDonate* uses services from the *Scheduler* to perform a time-slice donation to the message's recipient towards reducing the latency of achieving a response. More details about donation schema will be given in further Section 3.2.4.

Message Receiving There are two implementations for an elemental receive. One is completely non-blocking, while the other is blocking. The former (Listing 3.8) leads the kernel to check if there's some message pending within the *port*'s queue. If so, it pops the one on top and copies it to the specified address within the system call parameters.

```
if Queue has messages then
    Pop message;
    Copy it to buffer;
    Remove from message pool;
end
```

Listing 3.8: Algorithm of the recursive function from the *revoke* operation.

The blocking (Listing 3.9) will check for a message on the queue, and if it is not empty, it will behave in the exact same way as the previous call. However, if the queue is empty, it will register an address to be the destination for the next message to be sent over that port, in addition, it will change the partition state to blocked. The *PartitionManager* invokes a routine for the context restore, where a message passing callback is invoked. It checks if there is a message pending to a blocked partition, and if so, as aforementioned, there's a message cached within the port, which is going to be copied for the stored address.

In a non-blocking receive over a *PortGroup*, the operation will occur until a *port* successfully pops a message, while in blocking one it will invoke the described method for every *port* encompassed by the *PortGroup*. The latter, will have the partition's state changed to blocked. Thus, whenever receiving a message in one of

the *ports* encompassed by the *PortGroup*, the partition will become active again. In both operations, the message will also contain the index from the partition's capability space, whose capability is referencing the *port* to which message was directed.

```
Invoke port unblocking receive;
if Have not received any message then
    Register address for receiveing message;
    Block partition execution;
    Invoke schedule;
end
```

Listing 3.9: Algorithm of the recursive function from the *revoke* operation.

Donations and grant one-time The remainder operations encompass a combination of the already described operations with functionalities from both the *Scheduler* and *CapabilityManager* classes. For donations, the kernel changes recipient partition's station and creates a chain of donations, given that a donatee may also donate its time-budget. This will be explained in more detail in Section 3.2.4. Furthermore, -Donate suffixed operations, as well as *SendReply*, encompass the granting of a capability. This is a one-time grant, meaning that upon recipients response, the capability will be removed from its address space.

3.2.2.2 Synchronization

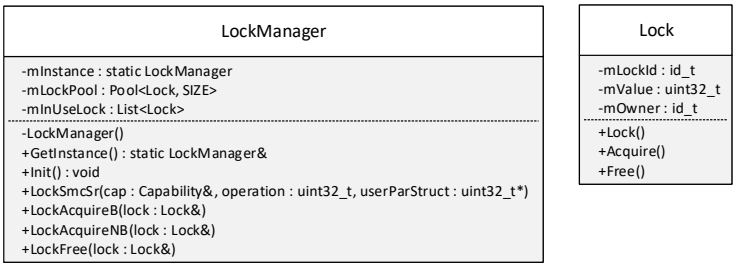


Figure 3.9: Classes that encapsulate synchronization functionality.

For synchronization purposes the *Lock* class was implemented, and is managed by *LockManager* (Figure 3.9). It basically can be used as spin-lock or a mutex. The difference between these mechanisms is on how the failure of trying to acquire the *Lock* will be handled. The failure on acquiring the *lock* happens whenever another partition already owns the *lock*, which means the currently executing should wait until it frees the *lock*. As aforementioned in Section 3.1.7, the *lock*

works as a mutex if the *AcquireBlock* operations is used, or as *spin-lock* if using the *AcquireNonBlocking*.

```
while Lock/Store has not succeeded do
    Invoke instruction LDREX;
    if Previous state was acquired then
        return error;
    else
        Invoke instruction STREX;
    end
end
Set owner information;
```

Listing 3.10: Algorithm of the *Lock* operation.

```
if Is current partition the owner then
    Reset state and owner;
else
    Return error;
end
```

Listing 3.11: Algorithm of the *Free* operation.

Listing 3.10 and listing 3.11 present the algorithm for acquiring and freeing operations over a *lock*, respectively. The first sets the value of *lock*'s state, while the other clears it. If the memory position is set means the *lock* has already been acquired by another partition. The owning partition's ID is stored within the *Lock* class, in addition to the value and *lock* ID used for configuration purposes. Trying to acquire a *lock* is done by requiring to *ARMv7* load and store exclusive instructions. The first is for loading the value state to verify it is set or unset. The store is used to set the value, thus acquiring the *lock*. This process should happen atomically as no other access to the value's memory address should happen between the *LDREX* and the *STREX*. In case other memory access happens between them, the *STREX* will fail. In case the *STREX* fails, the execution of the acquiring procedure should return the *LDREX*. The *Free* operation just checks the ID of the running partition, and if it matching owner's ID, the value is cleared.

3.2.3 Events

Events are delivered to partitions through an instance of an *EventGate* class, which in turn is managed by the *EventManager* (Figure 3.10). As aforementioned, depending on the partition type we have different mechanisms to deliver and alter partition's execution flow. In case of a guest partition, a virtual interrupt is sent,

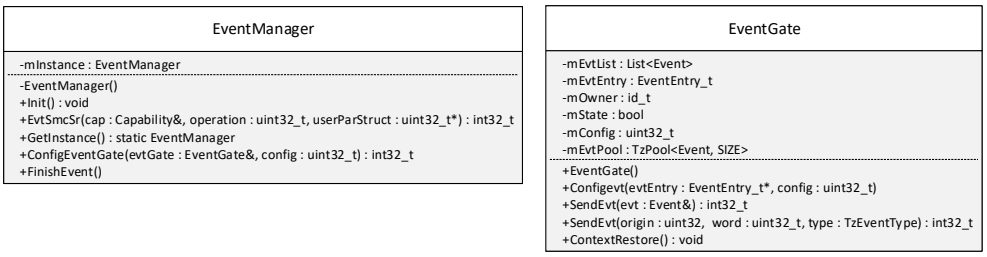


Figure 3.10: Classes that encapsulate events functionality.

while for secure tasks events are delivery as unix-like signals. To implement signals as a virtual interrupt, services from *InterruptManager* class are going to be used. While services from *PartitionManager* are used for the unix-like signals.

EventGates are assigned one per partition during initialization, in addition to the respective capability with no *grant* or *revoke* rights set. Thus, each *EventGate* is confined to the owner partition. Notwithstanding, sibling partitions may trigger an event through asynchronous communication, either by sending a message or through the *notify* operation, over a port. In order to events preempt partition's execution, the respective partition should configure their *EventGate* properly, enabling events and providing an event handler, through means of the *TzEvtEntry_t* structure. Listing 3.12 shows the structure used as an entry point to partitions' space, which holds the information for the last occurred event towards the owning partition and the callback to be invoked. The system call for event gates configuration receives the capability referencing the event gate, in addition to the address of a dedicated structure, where, upon the last event occurrence, its data is written to. The event structure encompasses: a miscellaneous word, whose meaning relates with the event's scope; its origin, which will contain the index of the capability, within owner's capability space, referencing the port that triggered the event; and the type of event, which will be notification or a message arrival in the origin referred port. The type for event handlers is also presented in the Listing 3.12. There is a field within *TzEvtEntry_t* structure to be assigned with the address of the signal's handler. This is going to be used to redirect task's execution to that address. Guests partitions the interrupt specified for the purpose should be properly configured.

For both partition types, whenever an event is handled the *Finish* operation must be performed, to inform the kernel that the partition is ready for receiving another one, otherwise events will be pending on *EventGate*'s events queue. Upon the *Finish* event operation, another event is prompted if there is some pending. Moreover, there is a callback to be invoked at every context restore, that will prompt available events.

Listing 3.12: μRTZVisor's events and events entry structures

```
typedef void TzEventHandler_t(void);

struct TzEvtEntry_t {
  struct TzEvent {
    uint32_t mWord;
    uint32_t mOrigin;
    enum TzEventType {
      NEW_MSG = 0,
      NOTIFICATION
    } mType;
  } curEvt;
  TzEventHandler_t* eventHandler;
};
```

3.2.4 Scheduler

Synchronous IPC implementation is tightly coupled with scheduling functionalities, namely for blocking on a receive, or donate the respective time quantum to a server. As aforementioned, scheduling functionalities that do not directly aim at providing IPC support fall out of the scope of this dissertation, and, consequently, will not be described with higher detail in this section.

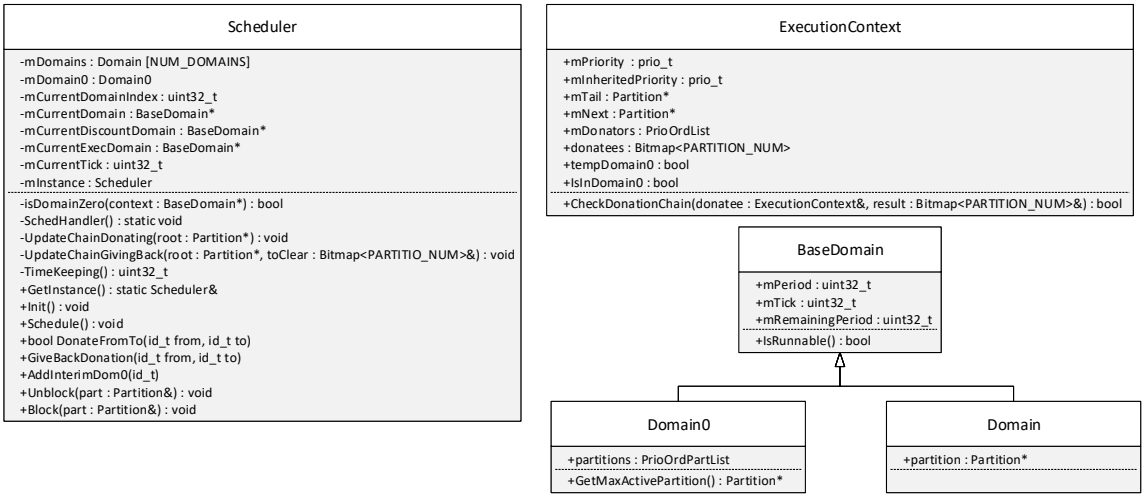


Figure 3.11: Classes that encapsulate scheduling functionality.

Figure 3.11 depicts all classes with scheduling functionality. The *Scheduler* class provides the set of functions referenced by *PortManager*. In addition, it aggregates all necessary structures to support the described scheduling algorithm (section 3.1.9). *BaseDomain* class and its derived ones describe what we call a time domain, which is assigned with a time quantum. The main difference is that *Domain* supports one partition, while *Domain0* supports more than one. In *Domain0* partitions are stored in a prioritized manner, and requires the tick

configuration, which will be the time a given partition within domain0 will execute, and that once expired will incur into a scheduling point. The *ExecutionContext* structure is part of the PCB, and it features a partition with a priority, and data used to support donations and priority inheritance.

To keep track of all donations, we leverage *ExecutionContext* to grow a chain. The tail partition is the one which execute, and that will inherent the highest priority on the chain. Tail and priority information is cached to avoid traversing the chain every time a partition in a donated state is scheduled. Every time a donation operation is issued, it is necessary to traverse the chain to update this information. If the donatee of the operation is inserted at the tail of the execution chain, the traversing procedure should happen backwards, i.e., from tail to head. Another scenario for the chain to grow happens when placing a partition at the chain's head. This happens if a partition donates its execution to a partition that is in *DONATED* stated, which means it already is in a donation chain. If from the newly inserted element onwards, the following partitions have lower priority, they should inherit the new head's priority. Consequently, in this case, the chain must be traversed forwardly. Sometimes, a chain may even become a tree if a given partition is the recipient of more than one donation. If so, every possible tree path should be traversed to update necessary information. Also, to make this feasible and to keep track of the highest donator for a given donatee, *ExecutionContext* is featured with a priority-based list, whose partition on top is the one from which priority is inherited.

Donations should be carefully implemented to avoid possible deadlocks. Deadlocks are created when performing a donation results in a cycle within the donation chain. In this sense, the *ExecutionContext* structure possesses a bitmap, where each bit represents a system's partition. A set bit means that the respective partition has position within the chain between the node it belongs to and the tail. This is means, it is included in the donation chain. Whenever donating, both donator's and donatee's bitmaps are crossed, and in case of match, a cycle is detected. As prevention mechanisms, all partitions within the chain between donator and donatee are awaken, i.e., are passed to active state. Every time traversing a chain for updating tail and priority, this bitmap is also updated.

The algorithm for the donation procedure is presented in listing 3.13. Firstly, the *ExecutionContext* from both donatee and donator are fetched, and checked for matches on the donatees bitmap. In case of a match, a cycle was detected, and, consequently, the recovery procedure is executed. This encompasses iterating over each *ExecutionContext* within the chain and give back the donation, waking all

partitions. In case of a mismatch, the donation can proceed accordingly.

```
Fetch donatee and donator execution contexts;
Invoke method for verifying donation chain;
if Cycles are detected then
    Set iterator equal to donatee;
    Set previous(auxiliar variable) to null;
    while donator different from donator do
        Give back donation from iterator to previous;
        Set previous equal to iterator;
        Set iterator equal to next;
    end
    Return error;
else
    Set donator next to donatee;
    Set donator's donatees bitmap;
    Insert donator into donatee's donators list;
    if The highest priority donator changed then
        Update priority from donatee's on, in domation chain;
    else
        Set donator's tail to donatee execution context;
        Set donator's state to donated;
        Update donation chain;
    end
end
```

Listing 3.13: Algorithm of the donation procedure.

As such, in the donatee is going to be inserted at the tail of the chain, and the donator's bitmap for cycle detection is set with the donatee's bitmap. The donator is also inserted in donatee's list of donators. This is an ordered list, that is used to track the highest priority donator. Every time this changes, the donatee's priority is compared to it, and priority inheritance can happen accordingly. Lastly, the donator's state is changed to *DONATED*, and the tail information is changed. Lastly, chain is updated for a donation scenario, whose procedure is going to be explained further.

Whenever a full cycle of an RPC-like communication happens, the donation should end, which means all aforementioned changes should be undone. The procedure of giving back donation is presented in Listing 3.14. Once again, both execution contexts are fetched, then the donator is removed from donatee's list of donators, its tail's value is reset, and its state is set to *ACTIVE*. Following, the changes are propagated through out the chain. If to the donatee remains no donator on its list, its priority is also reset, otherwise it is verified if the highest priority donator has changed, and if it has higher priority than the donatee. If so, the inherited priority is updated.

```
Fetch donatee and donator execution context;
Remove donator from donatee's donators list;
Set donator's tail pointing to him;
Change donator's state to active;
Update donateion chain;
if Donatee's donators list Is empty then
    Reset donatee's priority;
else
    if There is new high priority donator on top of the list then
        Update inherited priority;
    end
end
```

Listing 3.14: Algorithm for undoing a donation.

Listing 3.15 presents the algorithm for updating the donation chain upon a new donation. It consists on traversing the donations chain from the current donator towards partitions donating to it, i.e., traversing the chain backwards.

```
Set an iterator for traversing donators list;
while Iterator different from donators list end do
    Update tail;
    Set bit in donatees bitmap;
    Update donation chain;
end
```

Listing 3.15: Algorithm for updating donations chain upon a new donation.

This is achieved, by recurring to recursivity, and then iterating the donatee's list of donators and updating tail's value and setting the donatee's, at each node of the chain. For giving back the donation from a donatee (listing 3.16), the procedure is similar, with the slightly difference that every bit set in the cycle detection bitmap of the current donatee is unset from the donator's one, as well as all other iterated *ExecutionContexts*.

```
Clear every bit set in donatee's bitmap of donatees, in
 donator's one;
Set an iterator for traversing donators list of donatee;
while Iterator different from list's end do
    Update tail;
    Clear bit in donatees bitmap;
    Update donation chain;
end
```

Listing 3.16: Algorithm for updating donations chain upon giving back a donation.

3.2.5 Code Verification

The use of C++ for safety critical systems should be thoroughly considered. While C++ provides features that are, on its own, more secure compared with those available in the C language, the interpretation of the language constructs are often ambiguous, since compilers perform some default operations like type casts. Moreover, due to the extension of C++ internals, programmers do not possess the proper knowledge about all C++ constructs, frequently achieving different results from what expected. Also, there are some features that are implementation dependent, which usually compromises software portability [84, 26].

In this sense, the use of coding standards is highly advisable towards the development of safety critical systems, improving code safety, security, maintainability, and portability. Throughout the implementation of μRTZVisor we adopted the MISRA C++ standard, recurring even to code statical analysis. MISRA C++ [85, 84] advocates a set of rules and directives that enforce the proper use of C++ constructs. Herein are listed a set of followed rules, and directives, to provide and idea of those followed through the development of μRTZVisor:

- The use of keyword *const* whenever a given parameter is not changed within the body of a function, or when a method did not alter the properties' state of its object;
- Attributions should be explicitly typed, for example, when attributing a constant value to an *unsigned int* variable, the left-side operand should be explicitly casted accordingly;
- The preprocessor was only used for defining header files, avoiding the use of macros for defining constants. These were defined recurring to *const static* constructs and global variables;
- Every function definition should be paired with the declaration of its signature somewhere in the code;
- The use of conditional operands should explicitly define the order in which the operations should occur by using parenthesis, thus avoid unambiguity;
- Every *if ... else if ...* statement should finish with an *else*, towards a more defensive programming;
- For every function with a return type, there should exist only one *return* statement.

Experienced programmers can use a lot of them as good practice; however, when developing extensive implementations, it is common to have a few code lines unintentionally forgotten in the middle of the code that do not comply with these directives and rules. The use of static code analysis tools allows programmers to identify these points in code, notwithstanding this is a time consuming task, which is often seen as a drawback. Thus, to reduce the verification effort, code should be submitted to analysis throughout the development process. In current μRTZVisor implementation, the code is not verified in its full extension, however this should be conducted in a near future.

3.3 Evaluation

3.3.1 IPC performance

Table 3.2: Asynchronous IPC primitives latency (μs).

Message Size (bytes)	Send	Receive	SendReply
64	4.36	4,17	5,49
128	5.17	4.75	6.31
192	6.00	5.16	7.13
256	6.82	5.72	7.98
320	7.69	6.21	8.80

To evaluate the performance of communication mechanisms, we devised scenarios for both asynchronous and synchronous communication. Table 3.2 shows the times needed to perform the asynchronous *Send*, *Receive* and *SendReply* operations. Despite the performance varying slightly depending on whether the running partition is a guest or a secure task, this variation is not considered to be significant in the case of asynchronous communication. As such, the performed measurements only reflect the time that it takes to perform the respective hypercalls from a guest partition. For a 64 byte message size, the hypercall execution time is of 4.36, 4.17 and 5.49 μs for each operation, respectively. These times increased by about 1 μs for each additional 64 bytes in the message. In all cases, there is one copy to be made. In the *Send* and *SendReply* hypercalls, from the guest's address space into the port's message buffer, and the opposite for the *Receive* hypercall. There is a slight difference between the `Send` and *SendReply* execution times, depending on the recipients' state when the operations are performed. In our implementation, this dictates whether the communication is synchronous or asynchronous. The *SendReply* operation behaves similarly to the *Send* operation,

with the addition of granting a one-time capability to the recipient guest, which only degrades performance by around 1 μs.

To infer about synchronous communication performance, we have prepared three tests, performed under three different scenarios. The first test consists on measuring an elemental *Send* operation, ensuring that the recipient is blocked, resulting in a synchronous operation. The second (One Way test) and the third (Two Way test) tests encompass time-slice donations procedures. In the latter, a full RPC communication cycle is measured, where the client partition waits for the server's response, while the former is used to measure only half of that cycle, i.e., the time that takes for a request to get the a blocked server. Each test scenario was performed between two guests (a) (Table 3.3); two tasks (b) (Table 3.4); and between one guest and one task (c) (Table 3.5). In each scenario, we kept the scheduling configurations simple, with only two partitions running concurrently, in this way reducing scheduling operations to the bare minimum.

Table 3.3: Synchronous IPC communication latency (us), in a guest-guest scenario.

Message Size (bytes)	Send	One Way	Two Way
64	15.21	195.14	385.73
128	16.24	197.23	389.45
192	16.78	199.76	394.18
256	18.58	202.65	398.10
320	18.88	204.66	402.39

We note that the performance of synchronous IPC, as mentioned in Section 3.1.6, heavily relies on the performance of time-slice donation services provided by Scheduler and data transfer services provided by the Memory Manager, which is reflected in the achieved results. The first test evaluates the time that it takes to deliver a message to a blocked recipient, without scheduling involved. In this way, we can measure the latency introduced only by the data transfer. By comparing this value with the other two, we can measure the overhead introduced by our approach to build the donation chain and consequent scheduling operation. Whenever donating, there's a context-switch involved, and as such, the overhead imposed by this operation also applies in this context. Since guest and task partitions are represented in different kernel structures, the needed save and restore operations performed during a context-switch depend on whether the preempted partition or the incoming one are guests or tasks. All of the possible combinations

give rise to four scenarios: guest–guest, task–task, guest–task and task–guest. The last scenario subdivides into two: when switching between a guest, to a task, back to the same guest, as in the case of RPC communication, there is no need for cache and TLB maintenance. However, when a guest is scheduled after a task, and the last active guest was not the same, cache flushing and TLB invalidation need to be performed.

Table 3.4: Synchronous IPC communication latency (us), in a task-task scenario.

Message Size (bytes)	Send	One Way	Two Way
64	5.46	20.17	42.90
128	6.18	20.96	44.35
192	6.88	21.74	45.87
256	7.57	22.50	47.31
320	8.28	23.42	48.83

Thus, in scenario (a); in which there are two guests communicating, results in the largest overhead in every performed test. Consequently, donation scenarios (b) and (c) involve tasks that are far more efficient. Regarding message size variation, we see that with the increase of 64 bytes in message size, the latency increases in just 1 to 2 μs, which is not too significant.

Table 3.5: Synchronous IPC communication latency (us), in a guest-task scenario.

Message Size (bytes)	Send	One Way	Two Way
64	6.01	28.97	63.50
128	7.15	30.30	66.60
192	8.37	30.89	68.63
256	9.48	32.32	71.49
320	10.68	33.25	73.77

Synchronous communication encompasses an increased overhead, regarding, donation and scheduling operations and resulting context-switches. On the other hand, it reduces latency in service provision. As for asynchronous communication, the time involved to perform each elemental operation is smaller. Nevertheless, it should be taken into account that in client–server scenarios, the respective response could take more time than with synchronous communication, since partitions other than the server might be scheduled in the meantime. As such, the more partitions there are in a system, the bigger the latency for the response would be.

3.3.2 Security Analysis

The implemented artifacts rely on the assumption that the separation provided by the kernel is totally reliable:

- A sound management of Arm TrustZone configuration registers is provided, imposing that a non-secure running software can not access, by any means, the data existent on the secure-world. Thus, guest partitions cannot access any kernel internal information, nor secure world tasks' address spaces;

- Secure world tasks are properly confined to each respective address space leveraging the existent MMU. Thus, insuring that the tasks cannot access kernel nor guests' data.

Throughout the development of the hypervisor, a secure-by-design approach was adopted, that aimed at complimenting the provided isolation with supplementary policies and mechanisms. Thus, constraining the use of system calls, namely those supporting IPC functionality. In the remainder of this section, a security discussion is conducted, explaining how the developed work complies or not with the three fundamental elements of CIA [26]: confidentiality, integrity, and availability:

- **Confidentiality** regards to the ability to restrict data to only those with authorized access. In our implementation messages are directed to ports, and can only be accessed by the port's owner. This principle is enforced by restricting capabilities of others than the port's owner to not have the right to perform a *receive* operation. This is achieved either during initialization as well as by blocking a grant operation of a capability referencing a port that violates this premise. Synchronous primitives, in which the partition will block waiting for a message, only an address can be specified within the caller address space, otherwise an error will be prompted. In asynchronous communication primitives, messages are buffered in kernel space, and the only way to access them is through a *receive* operation, whose access is constrained as explained. Identity spoofing is also impossible towards unauthorized access, as partition's information is managed within the kernel, as well as all capabilities, that also possess the owner's identity. Moreover, partitions access to capabilities is mediated by the kernel. The index that partitions provide whenever performing an operation is used to fetch capabilities within its capability space. Given the isolation and indirect access, partitions are forbidden of tampering capabilities for either forging their identity

or maliciously change permissions. Also, all used memory is sanitized after fulfilling its purpose, to prevent the possibility of data leakage.

- **Integrity** regards to the consistency, accuracy, and trustworthiness of data and of the system over its entire life cycle. Given the nature of our system, a given message is only considered trustworthy as long as it is provided by an intended sender, and its content complies with what was accorded by all explicitly authorized parties. Isolation ensures that the content of the message remains unchanged while cached within the kernel, while access-control ensures that the content is provided by intended parties. Thus, integrity is only partially achieved, given that message's content is never verified. This would be particularly useful in a client-server scenario, in which to perform its regular operation the server is expected to receive requests following a certain structure and content. If the message is not compliant with what is expected by the server, it should be discarded as soon as possible, avoiding unintended operations to be performed within the server. System calls' parameters should also be addressed to ensure integrity, thus all of them are submitted to sanity-check. Messages are also limited in size, and the maximum allowed size for a message is specified at design time, and every message copy does not surpass that size. This could be quite limiting in terms of throughput; however, this does not allow for memory to be overwritten.

- **Availability** emphasizes that authorized parties are able to access the information when needed. In our implementation this element must be analyzed under the light of time quantum donation, and its impact on the overall systems. Partitions' availability can be jeopardized if it blocks indefinitely. In Section 3.1.9 it was stated that a partition running on top of μRTZVisor is assigned with a priority and a time quantum. By performing IPC operation with the suffix -donate, a partition is explicitly donating its time quantum to the recipient of the message. In addition, if the recipient has lower priority than the sender, the former will inherit the latter's priority. Our scheduling algorithm does not allow for priority inversion to be a problem. Therefore, for every execution cycle, every time domain executes at least once. This is done according to priority, so high priority partitions will execute first, and lower priority partitions will always execute once the formers have their time quantum expired. The risk of a deadlock is also mitigated through the mechanism implemented to detect cycles within a donation chain (as described in Section 3.2.4). Notwithstanding, great care

should be taken when specifying communication relations with donation features. Capabilities should be leveraged to ensure that critical partitions only rely on trustworthy servers. This is because our system does not prevent a partition to monopolize a donated time quantum, which could be achieved recurring to timeouts. Timeouts were not included in our implementation, given that (as explained in Section 2.3) there is no precise way of specifying them. Availability can also be compromised by resource exhaustion, which could be achieved in the message, and capabilities pools. In case of attack, new messages are prevented to be sent, prompting an error on the sender's side. Regarding capabilities, if an attack occurs by intensively performing spurious grants, at some point capability spaces will get filled. In case this happens, the kernel will send a message for the capability space's owner, directed to his *kernel port*. Thus, it creates the possibility for a recovery, even that probably not so effective. However, the overall system functionality is assured since for every object, there is one specified pool of memory, that never gets surpassed. Thus, the overall system remains functional, enabling some control to be performed towards system's safe recovery, e.g., a simple reboot.

Throughout the discussion, *locks*, *EvtGates* and *PortGroups* were not mentioned, however, much of what aforementioned does also apply for them.

Chapter 4

Hypervisor's Design Automation

In this section, it will be presented the followed approach on the hypervisor's design automation. We combine the SeML (described in Section 2.10) infrastructure with a newly implemented DSL to be used as the front-end. The referred DSL is entitled TrustZone Description Language (TZDL), and aims at describing microkernel-like systems. Having SeML's workflow in mind, firstly it will be explained the first thoughts towards our methodology. Then, the TZDL infrastructure is detailed, namely workflow, language constructs and purpose, as well as the final code generation.

4.1 Methodology and Context

First and foremost, it must be defined what is going to be modeled. μRTZVisor aims at relocating variability from within the kernel to user-space, presuming that the kernel possesses a static and trustworthy implementation. Thus, preventing untrustworthy software from executing with the highest level of privilege. Therefore, we aim at providing a modeling infrastructure towards systems based on μRTZVisor, namely configuring the necessary kernel resources, as well as providing a means for describing user-level applications and servers. Also, we intend to provide a way for describing interfaces, namely services provided among partitions. As result, code generation must consist on generating source code files for the configuration of μRTZVisor infrastructure, which should encompass devices attribution, allocation of communication resources, address-space configuration, and capability distribution and permissions configuration. In addition, the ultimate goal is the generation of user-space code that automate partitions initialization, regarding resources configuration and capability fetching, as well as communication glue code for service provision.

SeML is a semantically enriched meta-modeling framework. It is featured with an upper ontology, that is required to be used as the base for every prescriptive

domain-ontology, to make it compatible with the necessary reasoning within the framework. Its workflow was described in Section 2.10. Basically, the modeling procedure consists on instantiating individuals from the domain ontology, originating the DSL's input ontology, and then connecting those individuals through object properties, using the DSL. For this, firstly it would be needed to use the *Protégé* tool, to extend the SeML owl file with the new entities and individuals, and then save it. Then, it would be used a text editor to resolve variabilities and connect individuals using SeML DSL. The resulted SeML file would be quite verbose, requiring a lot of typing that with time becomes tedious and error prone, taking a lot of time to type it. In addition, not being able to instantiate domain-ontology concepts within the editor divides user's attention between the editor and *Protégé* (the tool we used to conceived ontologies), which on top of aforementioned issues makes the framework not as user friendly as desirable. Notwithstanding, SeML provides a great common-ground, enabling knowledge extendability by aligning different domain ontologies, and also functionality extension through its external tools engine.

Under the light of the above issues, and the short experience we were having in using SeML on its own, we decided to focus a collaborative effort to mitigate the identified drawbacks. The followed approached had to be based on SeML infrastructure to still provide a good means for extendability, leveraging as much as possible of the already implemented engines. Also, it should not hinder the use of SeML as it is, it should, however, provide a simpler dedicated means for describing μRTZVisor-based systems in a user-friendly, effective way.

4.2 TZ Description Language

TrustZone Description Language (TZDL) consist on a semantically enriched DSL, that provides a means for describing μRTZVisor-based systems, aiming at easing the modeling process based on the SeML infrastructure. Figure 4.1 depicts the overall architecture for TZDL, and how it relates to the SeML framework. As can be seen, TZDL is a front-end for SeML modeling framework, given that the artifacts of the former are used as an input for the latter framework. Both aim at providing faster μRTZVisor-based systems configuration, and without SeML, TZDL work is incomplete. We advocate that the CBSE towards embedded systems modeling is the best approach, leveraging components composability and functionality containment. Also, IDL proved to be a plus on defining a interface semantics, and glue code generation.

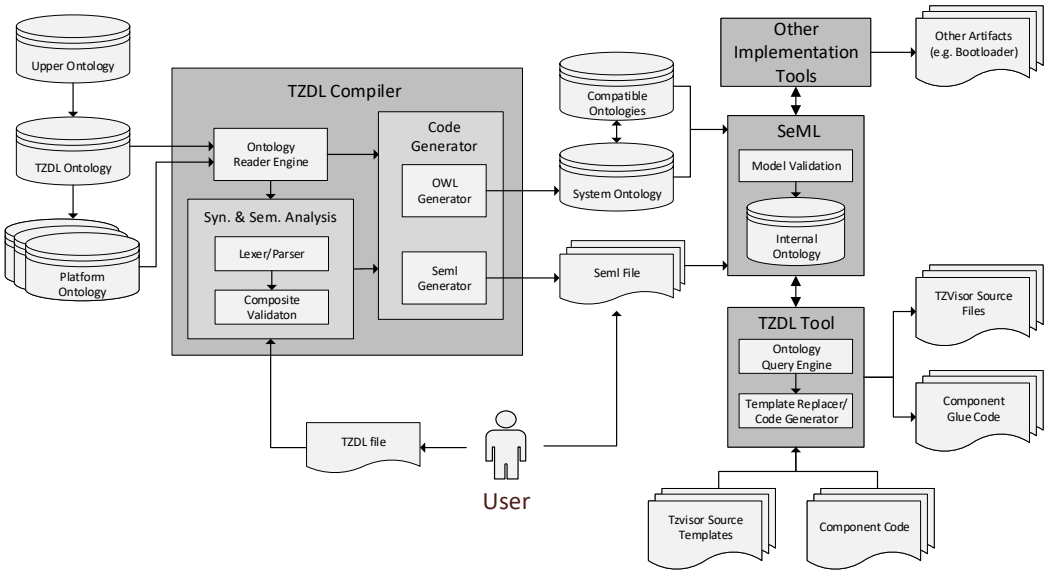


Figure 4.1: Overview of TZDL workflow.

Firstly, it was conceived a domain ontology that contains knowledge for describing microkernel-like systems, namely, with concepts that were deployed within μRTZVisor architecture. This ontology is derived from SeML's upper ontology, and, as such, can be used with this framework on its own. On the TZDL architecture the ontology is used for mapping its DSL constructs to ontology concepts, instantiating individuals, originating the system ontology. In addition, it is generated a SeML file that performs the binding of instantiated individuals. Both the SeML file and system ontology are used as input for SeML framework, that can also be extended with external knowledge to support other functionalities. For the final source code generation, was developed the TZDL Tool, which is an external tool for the SeML framework. This queries the final ontology for fetching the necessary individuals, bindings and configuration informations, necessary during final source code generation.

TZDL is composed of an ADL featured with IDL-like constructs, that imposes a system's description in a component-based architecture. The existing elemental components are described as partitions, namely, guests or tasks, according to μRTZVisor architecture. For composition, most abstract elementals components, called composites, are also available. Elemental components are required to be instantiated within a composite for being deployed in the final system. Also, elemental components provide services to each other. In this sense, a given component *requires* a certain type of interface, that another one provides. Both the requirer and the provider must be instantiated within the same composite, in which both can be binded. So far, only static bindings are supported for a matter

of simplicity. This means that dependencies across partitions would not change, neither during compile time nor at run time.

The elemental components are thought to be easily reusable, which means their features must be defined outside the component's body. We specified that components would only possess properties to specify devices it needs and properties related to scheduling functionality, namely period and priority, according to μRTZVisor's scheduling algorithm. Devices should be binded, the same way as interfaces. Scheduling properties should be specified at instantiation, within the respective composite. According to components description, the binding of interfaces, devices and properties configuration, ontology concepts will be instantiated and binded through object properties, in both aforementioned SeML input artifacts.

As it will be explained in detail in sections bellow, in both DSL constructs and domain ontology, IDL interfaces can be typed with prefixes *RPC*, *Data* and *Notification*. For *RPC* typed interfaces it was specified a scenario in which, for every communication attempt, the client is going to block waiting for the server's response, and, as the server does not always possess permissions to communicate with the client, a one-time capability should be provided solely for the reply (as in a normal RPC scenario). For this, the respective construct allows for the procedure calls description, similar to C syntax. The *Notification* typed interfaces regards to a scenario where the server uses the *notify* primitives to send events to the client. *Data* typed interfaces approach to more open scenarios, in which a message is going to be sent by either recurring to blocking or non-blocking calls. In the latter, it is intended that on the client side there is the possibility to use μRTZVisor's -donate suffixed primitives that include a grant-one time, or either an elemental send to the server.

All the aforementioned procedures are mediated by the represented user, which is someone that aims to develop a system based on μRTZVisor. He/She may be someone with no expertise on ontologies, and if so, he has to use the design automation infrastructure according to policies imposed by TZDL's grammar, which on its own provides enough capacities to describe meaningful systems. Nonetheless, if he is someone with knowledge on how to operate ontologies and SeML files, he will be able to extent its modeled architecture, with a more fine-grained control; however, having also to ensure the source code generating tool is able to support the performed changes.

4.2.1 Domain Ontology

Every time creating an ontology, it is necessary to clearly have in mind its scope, purpose, applicability, and possible use-cases. The ontology we intend to conceive must possess enough concepts to clearly describe a system based on μRTZVisor. Encompassing the terms that were stated previously, to describe partitions and interfaces they provide to each other, as well as, to assign kernel resources to partitions. Moreover, it enables code generation towards design automation, for either resource configuration and communication between partitions running on μRTZVisor. Therefore, the architectural concepts within μRTZVisor must be described in the ontology, as well as other abstract concepts like interfaces which are related with functionality and how it is provided. Also, this ontology must be aligned to SeML's upper ontology to make it compliant with its inner engine, which will be responsible to perform the ultimate code generation. It is also worth to have in mind the approach that is going to be taken for the design automation, i.e., the conception of a DSL to abstract SeML required workflow.

The following competency questions were formulated to help into conceiving a taxonomy, which would be used to create our ontology, where relations between taxonomy's concepts were created:

- What objects does the kernel provide?
- Which platforms will the kernels target?
- Which partitions should be implemented?
- Which resources should be assigned to which partitions?
- What type of objects exists, and which operations they support?
- Which permissions do partitions have over a certain kernel object?
- Which priority should be assigned to each partition?
- How are partitions assigned to time domains?
- Which services do a partitions provide, following which interface?
- Which type of interfaces exists?
- Which services do partitions require from its siblings?
- How are the described interfaces?

- Which resources are allocated to support a given communication, according to their interfaces?
- What policies does an interface comply with?

Figures 4.2 and 4.3 depict the taxonomy created, already derived from SeML's upper ontology. From the later, most of the new concepts derive from *Component* subclasses, namely *Entity*. In Figure 4.2 has some general concepts regarding kernel/microkernels, in addition to some more specific to object oriented implementations, namely with those objects available in μRTZVisor's implementation.

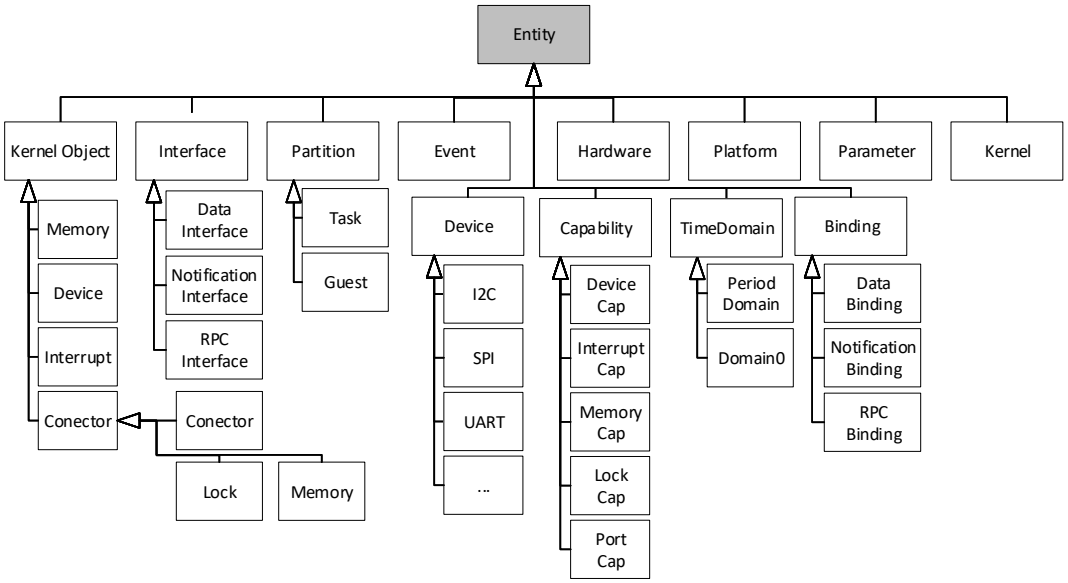


Figure 4.2: Taxonomy for domain ontology, namely *Entity* derived classes.

Partition class is subdivided into *Guest* and *Task* classes, that map into the type of partitions supported by μRTZVisor. *KernelObject* class is subdivided into kernel objects classes that translate into those available in μRTZVisor. Those that can be used for IPC mechanisms are subclasses of *Connector*, namely a *Lock*, *Port* and *Memory* for shared memory with capability grants. *Capability* derived classes represent one aggregating access permissions over a respective object type. *TimeDomain* class maps to the concept described in Section 3.1.9, both *Domain0* and a *PeriodDomain*, where the first can only be assigned to more than one partition, while the latter can only be assigned to one. *Interface* class describes three types of interfaces that will also be available in *TZDL* syntax: the *RPCInterface* that regards to RPC-like communication; the *DataInterface* that regards to a more open communication scenario (as previously explained), and encompasses the description of a message structure, to be used on a send operation; and *EventInterface*

that encompass a set of events that may be sent among partitions. The *Binding* class will aggregate informations for the provided interface, and capabilities referencing objects allocated to fulfill the functioning of that interface.

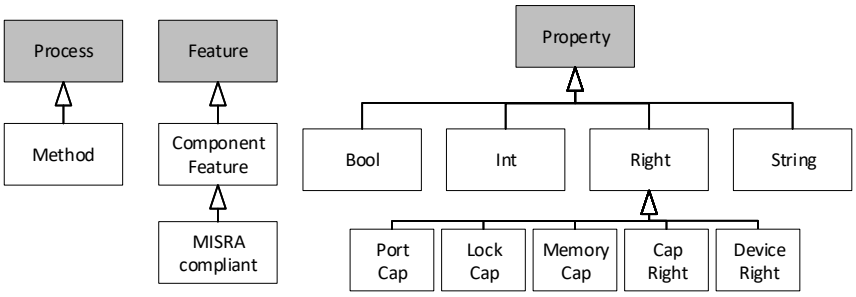


Figure 4.3: Other taxonomies derived from other upper ontology's concepts.

Figure 4.3 depicts concepts that derive from other classes belonging to the upper ontology, which are *Process*, *Feature* and *Property*. The first is used to describe a *Method*, which will be related to *RPCInterface* classes, that will abstract a given functionality with a function call, proper of RPC-like communication. For the class *Feature*, it was derived one possible non-functional requirement to characterize a partition. The derived classes of *Property* are used to describe type values, for either method parameters, capabilities rights and others.

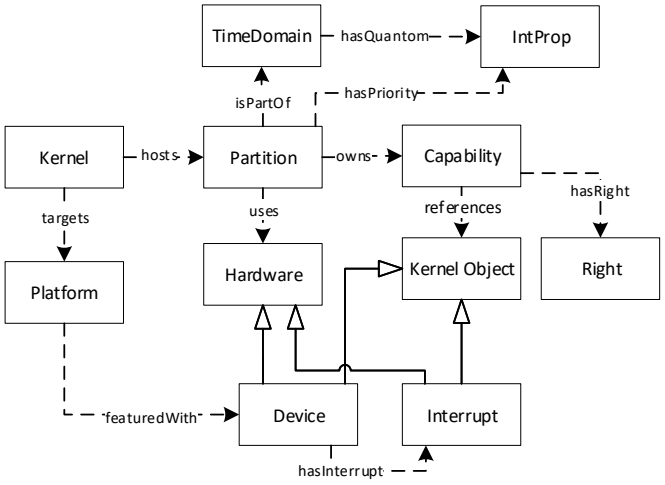


Figure 4.4: Excerpt of kernel's domain ontology.

Following, the exerpt of the resultant ontology is presented in Figure 4.4, depicting overall concepts and their relations. *Kernel* supports a set of *Partitions*, and is implemented to be *Platform*-compliant. This last concept is added to TZDL validation process that must check if the implementation devices required by partitions are part of the *Platform*. *Partitions* are also assigned to a *TimeDomain*,

which is configured with a time quantum. *Capabilities* reference *KernelObjects*, which are possessed by *Partitions*, and have rights according to the type of object, as demonstrated in the previous taxonomy.

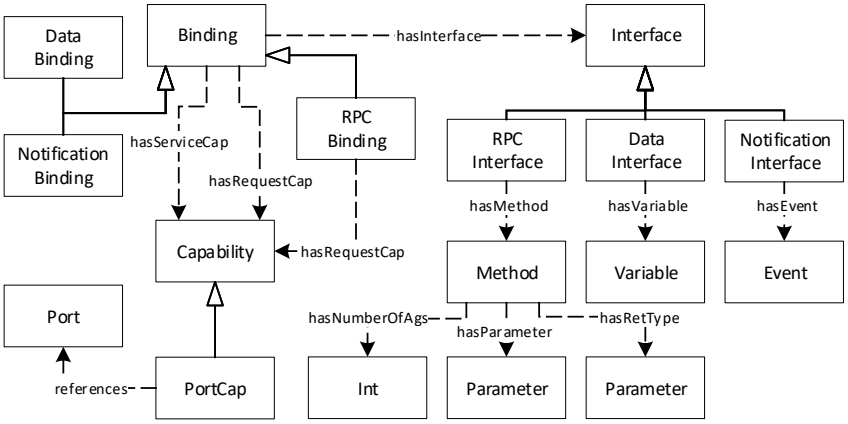


Figure 4.5: Excerpt of kernel's domain ontology, focusing on communication related objects.

Figure 4.5 depicts the excerpt, that relates to communication and interfaces concepts. *Interface* class is subdivided into three types, that are defined by a set of properties. *RPCInterface* is characterized by an aggregation of methods, whose signature is defined in a return type, a set of parameters and a name. A *DataInteface* aggregates a set of variables that should be part of the same structure, that consists on the message flowing within that communication relation. The *NotificationInterface* is defined by a set of *Event* classes, that define the type of events to be discriminated at the operation's recipient. The *Binding* class represents a communication relation and maps it to the needed resources, depending on the interface type. It is worth to point out that for each type of interface we assume different communication policies, and, consequently, communication primitives that will be used at the final code generation. Communication happens through *Port*s, which are referenced by the capabilities used in each operation within the communication relation. Regardless of the binding type, there should exist two capabilities mapped into the binding through *hasServiceCap* and *hasRequestCap*. The first capability represents the one to be attributed to the server for requests reception, while the latter should be attributed to the client for sending the requests, both referencing the same port. Notwithstanding, their rights configuration diverges according to the type upon *Capability* individuals creation in the following section. For the *RPCBinding* there should exist a third capability assigned to the client, referencing a port to be the target for a response message within the communication relation, granted to the server using to the one-time grant feature of μRTZVisor's IPC primitives. Details about the configuration of

the individuals originated from the bindings are going to be explained in the next section.

The domain ontology also possesses some individuals, namely for the subclasses of *Right* to describe permissions available within μRTZVisor's implementation, *Platform* and respective *Device* individuals, to specify which platforms a partition is compatible with, and which devices at the partition requires. The *PortRight* individuals, are for later map *Capability* individuals with the respective permissions.

4.2.2 Grammar

TZDL was implemented using the Xtext framework, that, through grammar specification, builds a syntax and semantics analyzer, execution engine and Eclipse IDE integration. The main purpose for TZDL DSL is to abstract individuals instantiation within SeML workflow, based on domain ontology's concepts. However, TZDL is not featured with constructs that directly map into all domain ontology's concepts, given a few imposed policies, which dictate how classes instances are created. For example with IDL like constructs, *Capability* and *Port* instances are created and configured according to the type of interface. For this purpose some constructs are provided, while others are conceptually more closer to the described ontology concepts.

As aforementioned, TZDL allows for systems description as component-based architectures, where partitions are described as being components that encapsulate functionality accessible to its siblings through well defined interfaces. The syntax allows component and interfaces description in a verbose manner, however very intuitive, that makes the typing very simple and eases the architecture comprehension by reading the code without any prior knowledge about the language. Table 4.1 presents all TZDL's constructs pairing them with description of their purpose within the language. For describing partitions to execute over μRTZVisor there are the *guest* and *task* constructs, which are followed by an ID, and within brackets its description and configuration. For both types it is required to point out the source code location, the compatible platforms and the minimal amount of memory required to allocate them, which is achieved by using the *source at*, *is compatible with*, and *alloc min* constructs, respectively. In addition, by using the *provides* and *requires* construct, also inside the brackets, it is possible to specify which services it implements, allowing the access from external parties, and which services from outside parties it depends on.

Table 4.1: Summary of all TZDL's constructs.

Language Constructs	Description
assemble ... **for** ...	Specifies the root composite for the model.
rpc ... { ... }	Declaration of RPC type interface.
data ... { ... }	Declaration of Data type interface.
event ... { ... }	Declaration of Event type interface.
guest ... { ... }	Declaration of Guest type component.
task ... { ... }	Declaration of Task type component.
composite ... { ... }	Declaration of Composite type component.
source at ...	Specifies the path towards the source of its component.
is compatible with ...	Lists all platforms that a given component is compatible with.
contains ...	Lists composite subcomponents separated by commas.
alloc min ...	Specifies size of a Guest or Task.
provides ...	Lists interface that a given component implements and provide to others.
requires ...	Lists interface dependencies that a given component needs to resolve.
requires device ...	Lists the devices a given component needs.
promote provide ...	Specifies a provided interface by one subcomponent within a composite, that is to be promoted.
promote require ...	To specify a required interface by one subcomponent within a composite, that is to be promoted.
promote device ...	To specify a device required by one subcomponent within a composite, that is to be promoted.
bind ... **to** ...	Connects required and provived interfaces within a composite, that belong to sibling subcomponents and have the same type.
in	Input parameter in a method description within a RPC interface.
out	Output parameter in a method description within a RPC interface.
has priority ...	Specifies component priority upon its instantiation within a composite.
with interrupt	State that a device's respective interrupt is also needed.

Recurring to the *composite* construct it is possible to aggregate other components, i.e., *tasks*, *guests* and *composites*. In every project, it should exist at least a composite that is the top-level one, i.e., that will be the root for the code

generation, specified by using the *assemble for* construct, which also specifies the target for which the code generation will happen. In the case a *task* or *guest* is being instantiated within a composite, its name definition must be followed by priority configuration. Partitions' period can also be added, however if absent the partition will not be assigned its own time domain, and will be assigned to *Domain0* at the generation of SeML artifacts. Then, it is possible to connect required and provided interfaces from the composite subcomponents by using the *bind to* construct. By binding required and provided interfaces, it is created a communication relation between the respective partitions, which will incur into instantiating resources with proper configurations from the domain ontology. In addition, regardless of being a provided or required, interfaces can be promoted within a *composite*, which will result in having the interface available for later binding upon composite instantiation. Required interfaces can only be used once, either on a promote or bind operation, while provided interface can be used multiple times.

Interfaces have constructs that almost perfectly match to domain ontology's concepts. The *rpc* construct defines a type of interface for RPC-like communication, and within brackets, are defined the methods signatures for the procedure call, which encompass the return type, method name, and parameters. Parameters can be tagged with the constructs *in* and *out* to specify if it is used for input or output respectively. For defining a structure to be sent in purely asynchronous client-server communication, the *data* construct is used. Lastly, the *event* construct is used to define interfaces that map to *Notification* class from the domain ontology, and within brackets are defined the events that can flow from a communication relation typed with the specified notification. Whenever describing a *composite*, *task*, *guest* or any interface, it is attributed an ID for the type, that will be referenced to create its respective instances.

4.2.3 Code Generation

There are two moments along the workflow that requires code generation engines to operate. The first happens after compiling a TZDL program, generating the system ontology (.owl file) and SeML files, with classes instances and object properties for relations establishment respectively; the other happens by executing TZDL Tool within SeML tools engine, that will fetch information from the resultant ontology and translate it into source code generation. The following sections explain these procedures respectively, however, as the generation of artifacts are

more specific to each dissertation's scope, it is given more focus on communication resources.

4.2.3.1 TZDL's code generation

This phase consists on, according to the TZDL file with system's description, instantiating individuals of classes from the domain ontology creating the system ontology, as well as the SeML file that connects the latter individuals through defined object properties. The purpose of each class individual, will be later reflected in the final source code generation. For describing a communication relationship on TZDL DSL, the programmer must first declare and describe an interface, that should be provided by a partition and required by another, and later these should be binded through the *Binding* construct. First of all, it will be instantiated an individual from the interface class upon its declaration. In addition, according to the interface's type and definition there are a few individuals to be created, which are summarized in Table 4.2. The interface resulting in more individuals is the *RPC* interface type, that will require the instance of a *Method* class, in addition to its respective parameters and return type. Following, a *Data* typed interface results on instances of *Variable* class, to define the data flowing through a communication relation defined with that type. Lastly, the *Notification* constructed interface will generate instances of the *Event* class, where each definition will provide an event type within the scope of the declared interface. The SeML file will later contain the relation between interfaces individuals and instances of its associated resources.

Table 4.2: Generation summary for each type of *Interface*.

Interface Type	Instances
RPC	*Method* instance for each description within interface construct.
	Parameter instance for each parameter definition within a method description.
	Parameter instance for the return type of each Method.
Data	*Variable* instance for each one defined within within Data construct.
Notification	*Event* instance, for each definition within *Notification* construct.

The *Binding* class is also subdivided into classes, that directly map to the interfaces types. The one to be instantiated is identified by analyzing the *Binding* construct and verifying the used interface type. A *Binding* class is used to map the resources that are allocated according to the policies imposed by the interface type, i.e., communication resources, namely *ports*, referencing *capabilities*,

and respective configurations. Table 4.3 summarizes all instantiated individuals according to their types. The *RPC*, that requires two ports, one assigned to the server for receiving requests, and another for the client for receiving responses. Accordingly, *Capability* individuals are created referencing the service port, one for the server with permissions to perform the receive blocking, and another assigned to the client for sending the requests. In addition, one capability assigned to a client is created to be granted in a IPC primitive. For the *Data* typed binding, a service port is created for the client to send messages to the server. According to what was described previously, two *Capability* individuals are created, one assigned to server with permissions for receiving messages, and another assigned to the client for message sending. Both permission description aims at fulfilling the flexible scenario described previously. Lastly, the *Notification* typed binding, encompass an instantiation from *Port* class for the server to send events to the client, in addition to *Capability* classes instances, one for the server to send the event, and another for the client to configure the service port. It is worth to mention, that the described *Capability* individuals will be mapping the respective *PortRight* individuals, through the *hasRight* object property.

Table 4.3: Generation summary for each type of *Binding*.

Binding Type	**Instances**
RPC	Service *Port*, attributed to the server for receiving requests.
	Service *Capability* owned by the server to perform receive operations over the service port.
	Request *Capability* owned by the client to send requests, referencing the service port.
	Reply *Port*, attributed to the client for receiving responses.
	Reply *Capability* owned by the client to get responses, and to grant it for the server to send responses.
Data	Service *Port* attributed to the server for receiving requests.
	Request *Capability* owned by the client to send requests.
	Service *Capability* owned by the server to perform receive operations over the service port.
Notification	Service *Port* attributed to the client for receiving Events.
	Service *Capability* owned by the server to send events over the service port.
	Request *Capability* owned by the client to configure the port.

Lastly, it is worth to mention that every service provider will be featured with a *PortGroup* instance, that will encompass all the ports created under each interface. Thus allowing to a receive blocking call to be performed in all of them making possible the attendance to incoming requests from every communication relation.

4.2.3.2 TZDL Tool

This is a java tool that is going to be incorporated into SeML tools engine through java reflection, and is responsible for the final source code generation. Given the scope of this dissertation, it will only be described a part of this tool that is related to communication resources. This consists on generating the configuration to be deployed within μRTZVisor ports, port groups and locks, as well as respective capabilities. Note that, so far, locks have not been mentioned, as they are not included within TZDL constructs; however, they can be described by the user in the middle of the depicted workflow, namely accessing generated artifacts prior to SeML framework operation. The tool should be used, whether TZDL has been used for the modeling process or not. That is, this tool works on the SeML framework on its own.

In order to fulfill its purpose, this tool must access the system ontology to access individuals, their relations and properties. To make this feasible the Pellet API was used, which defines an interface for accessing a reasoner querying it about specific individuals, individuals based on relations, object properties, data properties and others. In addition, the developed code was written in Xtend, which is, a language derived from java, aiming to provide a means for simpler and faster typing. Xtend is featured with language constructs specific for code generation, referred to as multi-line template expressions [86].

Listing 4.1: Ports and Capabilities' configuration structures.

```
1 struct PortConfig {
2   const int32_t mConfigId;
3   const int32_t mOwner;
4   const uint32_t mConf;
5 };
6 struct PortGroupConfig {
7   const int32_t mConfigId;
8   const int32_t mOwner;
9   const int32_t nrOfPorts;
10  const uint32_t *mPorts;
11 };
12 struct CapabilityConfig {
13  const int32_t mConfigId;
14  const int32_t mOwner;
15  const uint32_t mRights;
16  char const * const mName;
17 };
```

Listing 4.1 depicts the structures for both ports and capabilities configuration. Figure 4.6 depicts the excerpt of ontology relevant for the port configuration procedure. Using the aforementioned API, all *Port* instances are fetched. Given the depicted relations, it is possible to fetch the remaining information. That is,

the *Capability* instances referencing each *Port*, in addition to the respective *Right* instances, and owning *Partitions*. A similar procedure is carried for *PortGroup* instances.

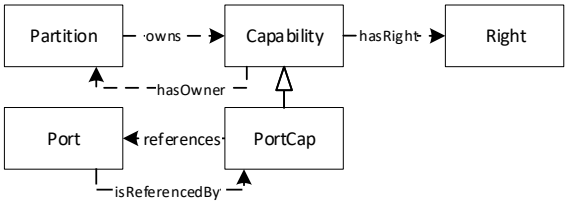


Figure 4.6: Excerpt of kernel's domain ontology, focusing on communication related objects.

As mentioned before, through the DSL it will be configured more than the communication resources, which follows a similar process to what was described. Figure 4.7 presents all generated configuration files for a μRTZVisor-based system. The described generation procedure only encompass the *PortConfig.cpp* file. The ultimate goal is to generate interfaces glue code, with the already specified communication semantics, providing even higher increases in development productivity. This was not achieved yet, but it will be implemented in a near future. Notwithstanding, with the final code generation is possible to infer the benefits of using TZDL.

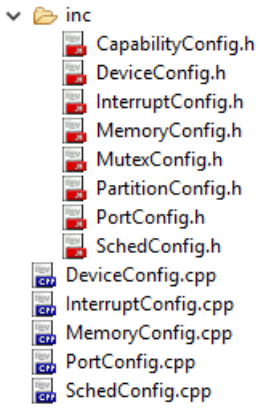


Figure 4.7: Source code organization/hierarchy of all generated configuration files.

4.3 Simple Use-Case

To provide a better understanding on the language's constructs and overall functioning, the example depicted on Figure 4.8 was implemented. It consists on the abstract modeling of a Publish-Subscriber scenario, encompassing *Publisher1*

and *Publisher2*, that publish topics through a event-typed interface assigned to each one of them. The aforementioned topics are delivered to the *Mediator* composite, that is responsible for controlling subscriptions, and delivering the topic accordingly. As such, the *Mediator* aggregates *Control* and *Subscriptions* components. The latter should store the subscriptions information, which are done through *Data_SubUnsub* interface, while the former delivers the topics to subscribers; however, it first checks subscriptions state with *Subscriptions* component through and *RPC*-typed interface. Both *Subscriber1* and *Subscriber2* receive the topic on a *event*-typed interface, but they should first specify to which topics they subscribe through *Data_SubUnsub* interface.

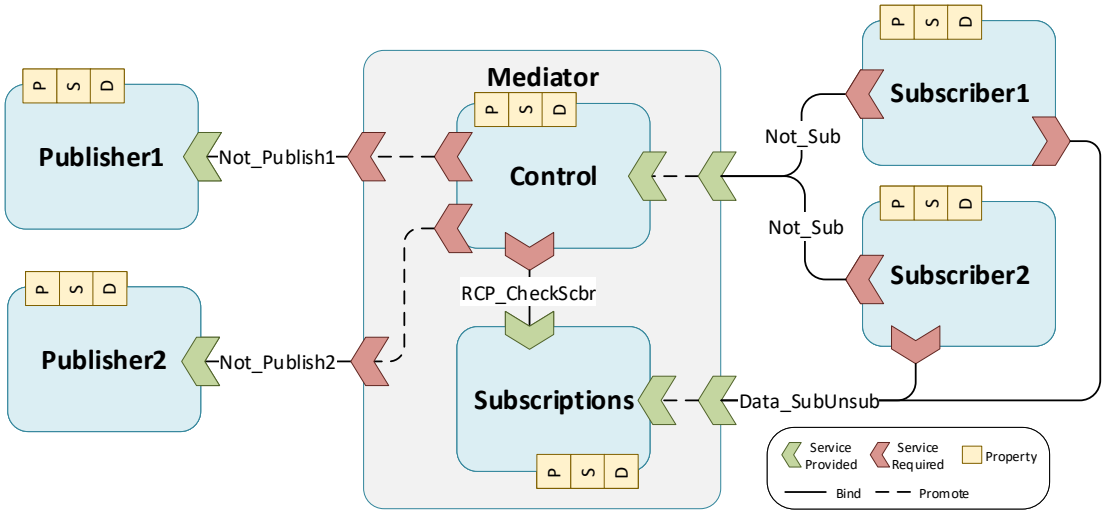


Figure 4.8: Component-based architecture for a Publisher-Subscriber scenario.

4.3.1 TZDL Program

Not all the code will be provided due to its extension, however, it will be presented the code for describing one type of each interface, some elemental components and a composite. Following, are the generated SeML artifacts and excerpt of ports' configuration file. Listing 4.2, presents the code for the description of a partition to be deployed as secure world task *Publisher1*, and another as guest *Subscriber1*. It is observable that both required interfaces from *Subscriber1*, in a form of a list. Both should be binded to a provider, otherwise an error will be prompted. The required amount of memory for each partition is the same, and will be used for configuring each other's address space. *Publisher1 provides* one interface for publishing topics, and *requires* one *I2C* device. So far, these are just descriptive as the glue code is not yet implemented.

Listing 4.2: Elemental components code example in TZDL.

```
task Publisher1{
  source at "<PATH>"
  is compatible with zynq, imx6
  alloc min 4KB

  provides Publish1 pPub1
  requires device I2C i2c
}

guest Subscriber1 {
  source at "<PATH>"
  is compatible with zynq
  alloc min 4KB

  requires Subscription rSub,
  SubUnsub rSubUnsub
}
```

Listing 4.3 presents the code for describing one type of each interface. The *CheckSub* interface is of type *rpc* and implements two procedure calls, to be abstracted in a RPC-like communication scenario. The specified parameters for procedure calls, need to be sent to the server in a specified protocol. This would encompass the creation of a message format with a specified opcode representative of each procedure call, that the server would interpret, operate accordingly and return to the client with the response. This response, for the *GetSub* function, might be solely an *ack*, while in the other should be the *string sub* parameter, tagged with the *out* keyword. *Publish2* interface defines the type of events, that should flow within a communication relation of that type. To each one of them should be assigned an opcode, interpreted within event handler on the client side. Lastly, the *SubUnsub* interface defines the structure for the message to flow within a communication relation of that type. It encompasses a set of variables that permits the subscription of a subscriber to a topic. The subscriber would specify its respective identification, the topic representative opcode, and the flag with the meaning of subscription or unsubscription, that the *Subscriptions* component would interpret and perform operations accordingly.

Listing 4.3: Interfaces code example in TZDL.

```
rpc CheckSub{
  uint32_t GetSub(in int topic, out string sub);
  uint32_t IsSub(in int topic, in string sub);
}
event Publish2{
  topic_c;
  topic_d;
}
event Subscription{
```

```
10   topic_a;
11   topic_b;
12   topic_c;
13   topic_d;
14 }
15 data SubUnsub {
16   bool subUnsub; //true->subscribe, false -> Unsubscribe
17   int topicId;
18   string subId;
19 }
```

The *Mediator* composite is presented in Listing 4.4. It is possible to check a section defining its contained components, followed by their properties configuration. By configuring this properties outside the respective component's, it is eased the components reuse. It is specified to which platform the composite aims, which should be compliant with what described within its components, i.e., all its contained components should be compliant to *zynq* platform. Within a composite it is also possible to specify interfaces it *requires* and *provides*; however, these definitions should be used to *promote* contained components' interfaces of the same type. As a result, these would become "visible" outside the composite for a binding, upon its instantiation. From line 13 to 17, it is possible to see all promote operations within mediator, and on line 19 it is possible verify the syntax for a binding.

Listing 4.4: Composite code example in TZDL.

```
1 composite Mediator{
2   contains Control controller has priority 5 and period 60;
3   Subscriptions subscriptions has priority 4;
4
5   is compatible with zynq
6
7   requires Publish1 prom_rPub1,
8   Publish2 prom_rPub2
9
10  provides SubUnsub prom_pSubUnsub,
11  Subscription prom_pSub
12
13  promote require controller.rPub1 as prom_rPub1
14  promote require controller.rPub2 as prom_rPub2
15
16  promote provide subscriptions.pSubUnsub as prom_pSubUnsub
17  promote provide controller.pSub as prom_pSub
18
19  bind controller.rCheckSub to subscriptions.pChechSub
20 }
```

Lastly, the system composite is presented in Listing 4.5. This encompasses the elements from Figure 4.8 and binds them according the specified interfaces. Also, the last line of code specifies the more abstract component, from which model description starts, thus dictating the result of next stages.

Listing 4.5: Composite code example in TZDL.

```
1 composite PublishSubscribe_example {
2   contains Publisher1 pub1 has priority 4;
3   Publisher2 pub2 has priority 4;
4   Subscriber1 sub1 has priority 5 and period 60;
5   Subscriber2 sub2 has priority 6 and period 70;
6   Mediator med;
7
8   is compatible with zynq
9
10  bind med.prom_rPub1 to pub1.pPub1
11  bind med.prom_rPub2 to pub2.pPub2
12
13  bind sub1.rSub to med.prom_pSub
14  bind sub2.rSub to med.prom_pSub
15
16  bind sub1.rSubUnsub to med.prom_pSubUnsub
17  bind sub2.rSubUnsub to med.prom_pSubUnsub
18
19  bind device pub1.i2c to zynq.i2c0
20 }
21
22 assemble PublishSubscribe_example for zynq
```

4.3.2 Code Generation

As explained in Section 4.2, SeML artifacts are generated from a TZDL program. This means a system ontology would be generated, composed with all individuals required, in addition to a SeML file connecting instances according to the model. In Section 4.2.3, it was explained which ontology instances are created for each type of *Interface* and each type of *Binding*. Due to the extension of the SeML file, Listing 4.6 solely exposes the mapping between individuals created for *Controller* component (from line 1 to 5), to provide *Subscription* interface of event type. Following, the mapping between individuals for the binding between the former and *Subscriber1* (from line 7 to 21).

Listing 4.6: Composite code example in TZDL.

```
1 // Server Objects Binding*/
2 med.controller ownsCapability med.controller.pSub.portgroupcap
3 med.controller.pSub.portgroupcap referencesObject
      med.controller.pSub.portgroup
4 med.controller.pSub.portgroupcap hasRight send
5 med.controller.pSub.portgroup hasOwner med.controller
6
7 // Event Binding from sub1.rSub to med.controller.pSub */
8 tzvisor hasBinding sub1.rSub-bindsto-med.controller.pSub
9 sub1.rSub-bindsto-med.controller.pSub hasInterface Subscription
10
11 sub1.rSub-bindsto-med.controller.pSub hasReplyCap
      sub1.rSub.replycap
```

```
sub1 owns sub1.rSub.replycap
sub1.rSub.replycap referencesObject sub1.rSub.port
sub1.rSub.port hasOwner sub1
sub1.rSub.replycap hasRight config

sub1.rSub-bindsto-med.controller.pSub hasServiceCap
    med.controller.pSub.portgroupcap
med.controller owns sub1.rSub.servcap
sub1.rSub.servcap hasRight send
sub1.rSub.servcap referencesObject sub1.rSub.port
med.controller.pSub.portgroup hasPort sub1.rSub.port
```

Providing both artifacts to the SeML infrastructure, this could invoke the implemented *TZDL tool* that generates configuration files. Listing 4.7 contains the excerpt from the *PortConfig.cpp* file exposing the configuration required for the aforementioned scenario. As would be expected, the configuration follows the structure explained in Section 4.2.3.2. Firstly, we see capability configuration for the port over which communication will occur. To the server were given permissions to perform the *notify* operation, as it is an *event* type interface, whilst to the client was given the right to configure the port. Configuring the port usually means enabling the reception of events, either by an asynchronous *send* or a *notify* from the sender. So in this scenario, the client would be able to enable/disable the events as needed, which is helpful for the client to perform some atomic operation. Following the port configuration, where it is assigned to the client and has its configuration reseted. The *mConfigId* helps to identify the port, and to associate the aforementioned capabilities with it.

Listing 4.7: Generated configuration files for the depicted example.

```
//Ports----
const CapabilityConfig portsCapabilities [] = {
  {//Server (Controller component) Capability
    mConfigId : 5,
    mOwner : 0,
    mRights : (0x01 << TzPortOperations::NOTIFY) ,
    mName : "rSub"},
  {//Client (Subscriber1 component) Capability
    mConfigId : 5,
    mOwner : 5,
    mRights : (0x01 << TzPortOperations::CONFIG),
    mName : "rSub"}
}
const PortConfig portsConfig[] = {
  {//Port configuration
    mConfigId: 5,
    mOwner : 5,
    mConf : 0 }
}
const PortGroupConfig portGroupConfigs[] = {
  {//PortGroup configuration
    mConfigId : 3,
    mOwner: 0,
```

```
24      nrOfPorts: 2,
25      mPorts : (uint32_t[]) { 5, 7 } }
26 }
```

4.4 Discussion

Once finalized this part of the work, it is possible to make some remarks. If confined to using SeML for the modeling procedure, the user would have to create class individuals from the domain ontology, and then connect them through object properties. This, on its own, is a more tedious procedure than modeling from a normal DSL, which gets worse since the user's attention has to be divided between two different tools. Moreover, the SeML DSL solely allow for connecting the instantiated individuals through a set of 3 consecutive words (like shown in the previous example). By solely reading the code, it is difficult to capture the overall architecture, and, as such, this becomes a poor documentation artifact. Even by using the ontology to retrieve information is harder than read an architecture on a DSL with specialized syntax. One point favoring SeML DSL is its shape-shifting syntax, as keywords are the imported relations from the domain ontology. As such, the more expressive these are, more expressive the DSL becomes.

TZDL adds a desirable abstraction, with considerate gains in expressiveness, ease of conception and maintainability. The DSL's constructs are more expressive to hypervisor's domain, and, as such, easier to understand. In addition, models are described in a more centralized environment and a more structured fashion. Consequently, the development process becomes more user friendly and therefore less error prone. Leveraging component-based syntax, the overall program structure gets easy to understand, read and maintain. As a result, the program becomes a valuable piece of documentation. However, compared to similar tools TZDL still has a lot to improve. For starters, only static bindings between components are provided, i.e. these are specified at design time, and a powerful feature would be to allow for bindings to happen at compile time according to parameters. In addition, bindings are also limited to providing a 1-to-1 mapping between components. If within a *composite* a given component provides a service to more than a sibling component, the user is forced to type more than one line of code. This limitation could be bypassed by allowing to define a list of components that require the same interface for latter bind to the same provider, reducing the number of code lines. Also, no extra properties are allowed to be defined in a TZDL program, which may be limiting. The supported properties are inherit to language syntax, but

those that are not static, like priority and period, are configured upon component instantiation within a composite. This is good for simplicity and component reuse. The component-based semantics of TZDL eases the desired confinement for a secure communication environment. That is, partitions that are critical and trustworthy can be easily composed within the same composite, and others that are not so secure nor critical can be composed separately. However, it should not exist any linkage, through interface binding, between them. Thus, imposing a proper capability configuration towards confinement between communication domains.

Some of the identified limitations of TZDL, can be overcame in the modeling workflow (remember Figure 4.1), because the user still has access to the generated SeML artifacts. The referred extra properties can be created at this point. In addition, given the control over low granularity of elements inherit to modeling using the ontology, the user is able to not follow or extend the imposed abstraction on TZDL constructs. For example, for each binding resources it will be configured according to the type of interface. However, the user may want to create a completely custom communication relation between two partitions. Through TZDL syntax this is not possible, however using SeML artifacts it is by creating individuals of *Port*, *Capability* and *Right* class, and associate them in the DSL. This possibility of extending what TZDL provides leveraged by experienced users, will increase the overall modeling flexibility. Also, the SeML external tool engine allows for this changes on the model to be noticed on the final source code.

The example provided in the previous section not only allow us to verify some of the aforementioned points, as it provides a means to infer about the gains of the code generation process. For instance, by comparing the amount of code written on TZDL to the one deployed in the μRTZVisor. So far, this is only reflected in configuration files, as the generation of interfaces and bindings respective glue is not yet generated. This should be addressed in future work, leveraging the taken steps towards this goal. Nonetheless, with what was implemented so far it is already possible to draw some conclusions. The architecture was described in one file, with approximately 85 lines of TZDL code. This code was used to generate a SeML file with approximately 190 lines of code, in addition to the individuals that were instantiated within the ontology. At this point it is possible to concretely state that TZDL development already incurs into productivity gains. However, the ultimate goal of the modeling process and the combination of both tools is the generation of the code to be deploy in μRTZVisor. This should constitute the real gain. From those depicted in Figure 4.7 (from Section 4.2.3.2), there are 5 source

code files that a developer should alter for properly configuring a μRTZVisor-based system. In addition, the effort for synchronizing configurations according to partitions should also be considered. For example, *PartitionConfig.h* is composed by the configuration of each partition, while *PortsConfig.cpp* possesses information for configuring communication resources. For ownership purposes, partitions ID must be aligned with the owner ID within its respective ports' configuration. If solely considering the generated code for communication resources it encompasses 248 lines of code. In addition to *PartitionConfig.h* that included 30 lines of code, the remaining did not contribute to the point we are trying to prove. This because the example was very simple, with special emphasis on communication. Notwithstanding, it already proves that the providing infrastructure for configuring systems to be deployed in μRTZVisor, provides great productivity results. These just will tend to increase, as meaningful and demanding the system becomes.

Chapter 5

Conclusion

Virtualization technology, although becoming a game-changer in the embedded space, on its own, cannot be seen as the solution for all security problems, which is often mistaken due to the provided isolation. As such, virtualization solutions must be conceived following a secure-by-design approach and relying on other security technologies. The in-house developed RTZVisor provides a virtualization solution, leveraging the Arm TrustZone technology. RTZVisor conception placed special focus on real-time systems, following a monolitic approach. Monolitic solutions are prone to hide bugs, and to incorporate untrustworthy software modules, namely device-drivers which often constitute a system's single point of failure. Moreover, the real-time features of RTZVisor are dependent on the guest OS solely implementing a cyclic scheduling algorithm, and RTZVisor is not featured with IPC mechanisms. Thus, a collaborative effort was conducted to enhance RTZVisor architecture with microkernel-like principles, to enforce overall kernel's security, while extending its support for real-time. μRTZVisor was conceived, and, as characteristic of microkernel-like architectures, following the principle of minimality, for limiting TCB size and functionality, in addition to a secure-by-design approach. This, while achieving a close to full-virtualization approach. The core implementation only encompasses services for spacial and temporal isolation, in addition to IPC mechanisms and a lightweight access-control facility. Leaving other functionalities to be implemented in user-space, with well defined interfaces for service provision.

The contribution of this thesis, towards the developed hypervisor artifacts, lays on the implementation of the IPC infrastructure and the access-control facilities, as well as some other complementary functionalities, like synchronization objects and so on. Communication was implemented based on the notion of an end-point, dubbed *port*, from which messages can be written to and read from. A vast number of IPC are provided, to support as many scenarios as possible; however, complying with the principle of minimality. The main limitation found regards to

performance, since on other reliable microkernel implementations communication operations can be performed in less than 1μs. In Section 3.3.1 the performance of the implemented IPC infrastructure is evaluated. The worst-case scenario is achieved when using synchronous primitives to communicate between two guests. Its poor performance is due to cache maintenance required when scheduling a new guest. To prevent data leakages, cashes must be flushed upon the execution of a new guest (as explained in Section 3.1.3). This procedure was noticed to be costly in terms of performance. In addition to *ports*, *portgroups* objects where implemented for aggregating *ports* under the same abstractions. Thus, making possible to perform elemental operations over that group, for example, to send a broadcast message. This feature was important to improve partitions communication, making it more flexible, which is useful when implementing servers, as these could often be providing services to more than one partition. Thus, it can wait for a message from all its clients, by aggregating the respective ports under the same abstraction.

In Section 3.3.2 a security discussion was conducted, approaching how the μRTZVisor implementation complied or not with each CIA (Confidentiality, Integrity and Availability). Most of the security guarantees rely on the provided spacial isolation being properly guaranteed. However, the former is properly complimented with the capability-based access-control system, system calls' parameters sanity-check, and memory pools sanitization. Still, great care should be taken in capability configuration, which dictates how communication relations are established. Partitions that implement critical functionality should only rely on trustworthy servers, which in turn should not be related to any untrustworthy client. In this sense, capability distribution should be thoroughly used to enforce confinement according to criticality and trustworthiness. Thus, closing possible bridges between partitions that are prone to external attacks, from those that implement critical functionalities, towards mitigating the chance of some partition having its availability jeopardized.

μRTZVisor architecture imposes functionality segregation towards a secure environment, which pairs with high-level of systems configuration, namely for establishing communication relations, and resource multiplexing. MDE bases software development on model representation, that provides a more abstract and, therefore, simplified view of the system. This approach is often paired with DSL development, that enhances models description with generative functionalities. The SeML infrastructure was developed towards hypervisor's design automation,

by leveraging ontology representation. In this sense, an upper ontology was provided, used for creating the domain ontology, that, in turn, should be used for the conception of a system's model. The final system representation also required the use of a DSL for mapping individuals within the system's ontology. The inherent workflow was still seen as tedious and error prone, providing few benefits as desirable when comparing its results to manually configuring μRTZVisor. However, the SeML infrastructure was a great means for model and functionality extendability. Thus, under the scope of this thesis a collaborative effort was conducted towards conceiving a DSL, dubbed TZDL, used as the front-end for the SeML infrastructure. In addition, a SeML external tool was developed for final source code generation, that possessed μRTZVisor's resources configuration. TZDL enabled model conception following a component-based approach, in which the model is composed of building-blocks with well defined interfaces between them. As explained in Section 4.2.3, a TZDL program was used to generate SeML artifacts. The ultimate goal of the developed work, was to provide an efficient way to configure μRTZVisor resources, in addition to generate interfaces glue code, with template functionality according to the interface's type. In Section 4.3 a simple use-case scenario was presented was presented, in which both TZDL and SeML infrastructure were used according to the described workflow (Section 4.2). From the former was possible to conclude that TZDL provided benefits into modeling conception, compared with using SeML infrastructure on its own, namely understandability, ease of use and maintainability. Notwithstanding, TZDL functionality was still far from being completed. In Section 4.4 some limitations were identified, for example, the impossibility of specifying other properties than priority and time budget. Despite that the generation of interfaces glue code was not yet achieved, the final source code generation results, regarding productivity gains, are quite promising. Notwithstanding, in the near future a more thorough evaluation must be conducted, by programming more complex architectures.

5.1 Future Work

Hypervisor Implementation The Inter-Partition Communication (IPC) performance is clearly the point that focusing on would create the most benefits. Firstly, hardware resources available on Arm's architecture should be thoroughly analyzed, namely DMA peripherals could be leveraged for faster messages' transmissions. On its own, a more thorough exploration of the memory subsystem (i.e. caches operation, MMU) could be conducted in order to improve the actual IPC's

implementation. Notwithstanding, for an efficient design of IPC mechanisms reliant on a DMA peripheral, it is required to have a sound comprehension on the core memory architecture, placing focus on TrustZone separation and cache maintenance. Another possibility, would be to offload IPC functionalities to hardware recurring to digital-hardware design. For every taken approach, tests should be conducted to infer about the incurred benefits, comparing results with the actual state of implementation.

In Section 4.4 a security discussion was conducted; however, this was merely theoretical. In the near future the implemented artifacts should be submitted to thorough testing, creating even attack scenarios. This would expose hidden vulnerabilities, as well as to test in practice everything that was advocated. We advocate that capabilities should be carefully distributed, confining communication relationships according to criticality and servers trustworthiness. However, other than capabilities being manually configured, there is no practical technique for ensuring that the desired security domains are not violated, i.e., that an untrustworthy party acquires permissions for communicating with partitions it is not intended to. For this, a monitoring tool could implemented. It should be prescribed with expected states for capabilities distribution over time. During runtime the tool should be responsible for aborting *grant* operations if they violated the prescribed information. This is useful to help mitigate risk of a DoS attack.

Throughout this work, part of the developed code was certified according to MISRA C++ standard. However, due to its extension and time constraints, the code has not been yet fully certified. In the future, an effort should be conducted towards full certification.

Design Automation The most demanding feature that would incur in a greater productivity improve is the extension of the *TZDL tool* to support the generation of interfaces glue code. This would avoid programmers having to type extensive repetitive code, namely for configuring the overall system according to the specification. Focusing on communication resources, these should include the configuration of endpoints (i.e. *ports* and *portgroups*), as well as the retrieval of respective capabilities references within its capability space. In addition, it should be automated as much as possible from the communication operations. For *event* interfaces the generated code should encompass, on the requirer side, the template code for identifying the event, and to redirect execution flow to the respective handler. The handler should also be provided as a stub function, for the programmer to fill as necessary. On the provider side, an API should be generated to send

each one of the described events. For *rpc* and *data* interfaces should be automated the marshaling and unmarshaling to generate a message structure, in addition to the stub code for performing operations regarding to the intended semantics of an interface. Namely, to implement the desired behavior for a procedure call defined within a *rpc* interface.

Under the light of what was explained in the previous section, some other features should be added to TZDL overall operation. Namely, allowing for bindings to be resolved on compile time according to parameters configuration. Also, allowing for extra component properties to be defined within TZDL syntax. As explained previously, this problem can be circumvented by defining properties manually in SeML artifacts. However, this requires ontology knowledge and having this on TZDL syntax is much more convenient. In addition, removing the necessity of manually altering SeML artifacts, in some other ways can be beneficial. Namely, allowing for describing elemental constructs like *ports* and *capabilities*, in addition to associate them with partition. Thus, the programmer can expand system functionality, beyond what was imposed by TZDL constructs.

Regarding the conceived domain ontology, in the future it could be expanded for supporting different microkernel-like implementations, as well as different platforms. In doing so, another source code generation tool should also be developed, to generate code compliant with the new target platform. Thus, by leveraging the already implemented infrastructure, the amount of effort needed to develop another modeling tool is reduced. In addition, towards greater level of flexibility, it provides a means for moving servers functionality to inside the kernel. Despite that this raises security questions, and it could only be applied to trustworthy servers, those that could also have been certified. This is due to the fact that sometime performance can be critical in a certain service provision. For this, server's interfaces should be of RPC type.

Bibliography

[1] G. Heiser, "The role of virtualization in embedded systems," in *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, pp. 11–16, ACM, 2008.

[2] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[3] M. Bruer, "Design of a semantic connector model for composition of metamodels in the context of software variability," *PhD diss., Technische Universität Dresden*, 2007.

[4] R. Kaiser, "Complex embedded systems-a case for virtualization," in *Intelligent solutions in Embedded Systems, 2009 Seventh Workshop on*, pp. 135–140, IEEE, 2009.

[5] J. Shuja, A. Gani, K. Bilal, A. U. R. Khan, S. A. Madani, S. U. Khan, and A. Y. Zomaya, "A survey of mobile device virtualization: taxonomy and state of the art," *ACM Computing Surveys (CSUR)*, vol. 49, no. 1, p. 1, 2016.

[6] G. Heiser, "Hypervisors for consumer electronics," in *Consumer Communications and Networking Conference, 2009. CCNC 2009. 6th IEEE*, pp. 1–5, IEEE, 2009.

[7] J. Shuja, K. Bilal, S. A. Madani, M. Othman, R. Ranjan, P. Balaji, and S. U. Khan, "Survey of techniques and architectures for designing energy-efficient data centers," *IEEE Systems Journal*, vol. 10, no. 2, pp. 507–519, 2016.

[8] G. Heiser and B. Leslie, "The okl4 microvisor: Convergence point of microkernels and hypervisors," in *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, pp. 19–24, ACM, 2010.

[9] G. Heiser, K. Elphinstone, I. Kuz, G. Klein, and S. M. Petters, "Towards trustworthy computing systems: taking microkernels to the next level," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 4, pp. 3–11, 2007.

[10] A. S. Tanenbaum and A. S. Woodhull, *Operating systems: design and implementation*, vol. 2. Prentice-Hall Englewood Cliffs, NJ, 1987.

[11] J. Liedtke, *On micro-kernel construction*, vol. 29. ACM, 1995.

[12] J. Liedtke, K. Elphinstone, S. Schonberg, H. Hartig, G. Heiser, N. Islam, and T. Jaeger, "Achieved ipc performance (still the foundation for extensibility)," in *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*, pp. 28–31, IEEE, 1997.

[13] J. Liedtke, "Improving ipc by kernel design," *ACM SIGOPS Operating Systems Review*, vol. 27, no. 5, pp. 175–188, 1993.

[14] K. Elphinstone and G. Heiser, "From l3 to sel4 what have we learnt in 20 years of l4 microkernels?," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 133–150, ACM, 2013.

[15] R. Kaiser and S. Wagner, "Evolution of the pikeos microkernel," in *First International Workshop on Microkernels for Embedded Systems*, p. 50, 2007.

[16] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Countering ipc threats in multiserver operating systems (a fundamental requirement for dependability)," in *Dependable Computing, 2008. PRDC'08. 14th IEEE Pacific Rim International Symposium on*, pp. 112–121, IEEE, 2008.

[17] J. S. Shapiro, "Vulnerabilities in synchronous ipc designs," in *Security and Privacy, 2003. Proceedings. 2003 Symposium on*, pp. 251–262, IEEE, 2003.

[18] J. Liedtke, N. Islam, and T. Jaeger, "Preventing denial-of-service attacks on a/spl mu/-kernel for weboses," in *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*, pp. 73–79, IEEE, 1997.

[19] A. Lackorzynski and A. Warg, "Taming subsystems: capabilities as universal resource access control in l4," in *Proceedings of the second Workshop on Isolation and Integration in Embedded Systems*, pp. 25–30, ACM, 2009.

[20] J. S. Shapiro and S. Weber, "Verifying the eros confinement mechanism," in *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*, pp. 166–176, IEEE, 2000.

[21] I. Crnkovic, "Component-based software engineering for embedded systems," in *Proceedings of the 27th international conference on Software engineering*, pp. 712–713, ACM, 2005.

[22] T. A. Henzinger and J. Sifakis, "The embedded systems design challenge," in *International Symposium on Formal Methods*, pp. 1–15, Springer, 2006.

[23] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "sel4: from general purpose to a proof of information flow enforcement," in *Security and Privacy (SP), 2013 IEEE Symposium on*, pp. 415–429, IEEE, 2013.

[24] D. Mellado, E. Fernández-Medina, and M. Piattini, "A common criteria based security requirements engineering process for the development of secure information systems," *Computer standards & interfaces*, vol. 29, no. 2, pp. 244–253, 2007.

[25] H. Löhr, A.-R. Sadeghi, C. Stüble, M. Weber, and M. Winandy, "Modeling trusted computing support in a protection profile for high assurance security kernels.," in *TRUST*, pp. 45–62, Springer, 2009.

[26] D. Kleidermacher and M. Kleidermacher, *Embedded systems security: practical methods for safe and secure software and systems development*. Elsevier, 2012.

[27] J. Alves-Foss, P. W. Oman, C. Taylor, and W. S. Harrison, "The mils architecture for high-assurance embedded systems," *International journal of embedded systems*, vol. 2, no. 3-4, pp. 239–247, 2006.

[28] S. Tverdyshev, "Security by design: Introduction to mils," in *this workshop*, 2017.

[29] A. Aguiar and F. Hessel, "Embedded systems' virtualization: The next challenge?," in *Rapid System prototyping (RSP), 2010 21st IEEE International symposium on*, pp. 1–7, IEEE, 2010.

[30] Z. Gu and Q. Zhao, "A state-of-the-art survey on real-time issues in embedded systems virtualization," *Journal of Software Engineering and Applications*, vol. 5, no. 04, p. 277, 2012.

[31] F. Armand and M. Gien, "A practical look at micro-kernels and virtual machine monitors," in *Consumer Communications and Networking Conference, 2009. CCNC 2009. 6th IEEE*, pp. 1–7, IEEE, 2009.

[32] G. Heiser, "Virtualizing embedded systems: why bother?," in *Proceedings of the 48th Design Automation Conference*, pp. 901–905, ACM, 2011.

[33] N. Penneman, D. Kudinskas, A. Rawsthorne, B. De Sutter, and K. De Bosschere, "Formal virtualization requirements for the arm architecture," *Journal of Systems Architecture*, vol. 59, no. 3, pp. 144–154, 2013.

[34] J. Smith and R. Nair, *Virtual machines: versatile platforms for systems and processes*. Elsevier, 2005.

[35] X. Wang, R. Habeeb, X. Ou, S. Amaravadi, J. Hatcliff, M. Mizuno, M. Neilsen, S. R. Rajagopalan, and S. Varadarajan, "Enhanced security of building automation systems through microkernel-based controller platforms," in *Distributed Computing Systems Workshops (ICDCSW), 2017 IEEE 37th International Conference on*, pp. 37–44, IEEE, 2017.

[36] M. Hohmuth, M. Peter, H. Härtig, and J. S. Shapiro, "Reducing tcb size by using untrusted components: small kernels versus virtual-machine monitors," in *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, p. 22, ACM, 2004.

[37] J. Liedtke, "Toward real microkernels," *Communications of the ACM*, vol. 39, no. 9, pp. 70–77, 1996.

[38] Y. Ren, L. Liu, Q. Zhang, Q. Wu, J. Guan, J. Kong, H. Dai, and L. Shao, "Shared-memory optimizations for inter-virtual-machine communication," *ACM Computing Surveys (CSUR)*, vol. 48, no. 4, p. 49, 2016.

[39] K. Kim, C. Kim, S.-I. Jung, H.-S. Shin, and J.-S. Kim, "Inter-domain socket communications supporting high performance and full binary compatibility on xen," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pp. 11–20, ACM, 2008.

[40] F. Diakhaté, M. Perache, R. Namyst, and H. Jourdren, "Efficient shared memory message passing for inter-vm communications.," in *Euro-Par Workshops*, pp. 53–62, Springer, 2008.

[41] J. S. Shapiro, D. J. Farber, and J. M. Smith, "The measured performance of a fast local ipc," in *Object-Orientation in Operating Systems, 1996., Proceedings of the Fifth International Workshop on*, pp. 89–94, IEEE, 1996.

[42] C. Gebhardt and A. Tomlinson, "Challenges for inter virtual machine communication," tech. rep., Technical Report RHUL-MA-2010-12. Royal Holloway, University of London, Department of Mathematics, 2010.

[43] U. Steinberg, J. Wolter, and H. Hartig, "Fast component interaction for real-time systems," in *Real-Time Systems, 2005.(ECRTS 2005). Proceedings. 17th Euromicro Conference on*, pp. 89–97, IEEE, 2005.

[44] I. Kuz, G. Klein, C. Lewis, and A. Walker, "capdl: A language for describing capability-based systems," in *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, pp. 31–36, ACM, 2010.

[45] X. Xiong and P. Liu, "Silver: Fine-grained and transparent protection domain primitives in commodity os kernel," in *International Workshop on Recent Advances in Intrusion Detection*, pp. 103–122, Springer, 2013.

[46] R. S. Sandhu and P. Samarati, "Access control: principle and practice," *IEEE communications magazine*, vol. 32, no. 9, pp. 40–48, 1994.

[47] J. S. Shapiro, J. M. Smith, and D. J. Farber, *EROS: a fast capability system*, vol. 33. ACM, 1999.

[48] L. I. Pesonen, D. M. Eyers, and J. Bacon, "A capability-based access control architecture for multi-domain publish/subscribe systems," in *Applications and the Internet, 2006. SAINT 2006. International Symposium on*, pp. 7–pp, IEEE, 2006.

[49] J. L. Hernandez-Ramos, A. J. Jara, L. Marın, and A. F. Skarmeta, "Distributed capability-based access control for the internet of things," *Journal of Internet Services and Information Security (JISIS)*, vol. 3, no. 3/4, pp. 1–16, 2013.

[50] L. Gong, "A secure identity-based capability system," in *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*, pp. 56–63, IEEE, 1989.

[51] U. Steinberg, "Quality-assuring scheduling in the fiasco microkernel," *Master's thesis, Dresden University of Technology*, 2004.

[52] E. Schierboom, *Verification of Fiasco's IPC implementation*. PhD thesis, Master's thesis, Radboud University, Computing Science Department, 2007.

[53] T. Smejkal, A. Lackorzynski, B. Engel, and M. Völp, "Transactional ipc in fiasco. oc," *OSPERT 2015*, p. 19, 2015.

[54] U. Steinberg and B. Kauer, "Nova: a microhypervisor-based secure virtualization architecture," in *Proceedings of the 5th European conference on Computer systems*, pp. 209–222, ACM, 2010.

[55] T. Xia, J.-C. Prévotet, and F. Nouvel, "Mini-nova: A lightweight arm-based virtualization microkernel supporting dynamic partial reconfiguration," in

Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International, pp. 71–80, IEEE, 2015.

[56] C. Baumann, B. Beckert, H. Blasum, and T. Bormer, "Better avionics software reliability by code verification," in *Proceedings, embedded world Conference, Nuremberg, Germany*, 2009.

[57] P. Varanasi and G. Heiser, "Hardware-supported virtualization on arm," in *Proceedings of the Second Asia-Pacific Workshop on Systems*, p. 11, ACM, 2011.

[58] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Minix 3: A highly reliable, self-repairing operating system," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 3, pp. 80–89, 2006.

[59] T. Alves, "Trustzone: Integrated hardware and software security," *White Paper*, 2004.

[60] S. Pinto, J. Pereira, T. Gomes, M. Ekpanyapong, and A. Tavares, "Towards a trustzone-assisted hypervisor for real time embedded systems," *IEEE Computer Architecture Letters*, 2016.

[61] A. Laarman and I. Kurtev, "Ontological metamodeling with explicit instantiation," in *International Conference on Software Language Engineering*, pp. 174–183, Springer, 2009.

[62] J. Hutchinson, M. Rouncefield, and J. Whittle, "Model-driven engineering practices in industry," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 633–642, ACM, 2011.

[63] V. García Díaz, N. Valdez, E. Rolando, J. P. Espada, P. G. Bustelo, B. Cristina, J. M. Cueva Lovelle, and C. E. Montenegro Marín, "A brief introduction to model-driven engineering," *Tecnura*, vol. 18, no. 40, pp. 127–142, 2014.

[64] B. Selic, "The pragmatics of model-driven development," *IEEE software*, vol. 20, no. 5, pp. 19–25, 2003.

[65] T. Stahl and M. Volter, *Model-driven software development: technology, engineering, management*. J. Wiley & Sons, 2006.

[66] D. Gaševic, D. Djuric, and V. Devedžic, *Model driven architecture and ontology development*. Springer Science & Business Media, 2006.

[67] U. Aßmann, S. Zschaler, and G. Wagner, "Ontologies, meta-models, and the model-driven paradigm," in *Ontologies for software engineering and software technology*, pp. 249–273, Springer, 2006.

[68] A. Van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *ACM Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[69] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM computing surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[70] K. Czarnecki, U. Eisenecker, R. Glück, D. Vandevoorde, and T. Veldhuizen, "Generative programming and active libraries," in *Generic Programming*, pp. 25–39, Springer, 2000.

[71] U. Zdun, "Concepts for model-driven design and evolution of domain-specific languages," 2005.

[72] I. Kuz, Y. Liu, I. Gorton, and G. Heiser, "Camkes: A component model for secure microkernel-based embedded systems," *Journal of Systems and Software*, vol. 80, no. 5, pp. 687–699, 2007.

[73] J.-G. Schneider, *Components, Scripts, and Glue: A conceptual framework for software composition*. PhD thesis, Ph. D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, 1999.

[74] O. Lobry, J. Navas, and J.-P. Babau, "Optimizing component-based embedded software," in *Computer Software and Applications Conference, 2009. COMPSAC'09. 33rd Annual IEEE International*, vol. 2, pp. 491–496, IEEE, 2009.

[75] I. Crnkovic, J. Stafford, and C. Szyperski, "Software components beyond programming: From routines to services," *Ieee software*, vol. 28, no. 3, pp. 22–26, 2011.

[76] C. Atkinson, C. Bunse, C. Peper, and H.-G. Gross, "Component-based software development for embedded systems–an introduction," in *Component-Based Software Development for Embedded Systems*, pp. 1–7, Springer, 2005.

[77] M. Pinto, L. Fuentes, and J. M. Troya, "Daop-adl: an architecture description language for dynamic component and aspect-based development," in *International Conference on Generative Programming and Component Engineering*, pp. 118–137, Springer, 2003.

[78] M. Nolin, J. Fredriksson, J. Hammarberg, J. Huselius, J. Hakansson, A. Karlsson, O. Larses, M. Lindgren, G. Mustapic, A. Moeller, *et al.*, "Component based software engineering for embedded systems-a literature survey," *MRTC Report, Maelardalen University, ISSN 1404*, vol. 3041, 2003.

[79] N. Feske, "A case study on the cost and benefit of dynamic rpc marshalling for low-level system components," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 4, pp. 40–48, 2007.

[80] O. Corcho, M. Fernández-López, and A. Gómez-Pérez, "Ontological engineering: principles, methods, tools and languages," in *Ontologies for software engineering and software technology*, pp. 1–48, Springer, 2006.

[81] D. Oberle, *Semantic management of middleware*, vol. 1. Springer Science & Business Media, 2006.

[82] H.-J. Happel and S. Seedorf, "Applications of ontologies in software engineering," in *Proc. of Workshop on Sematic Web Enabled Software Engineering"(SWESE) on the ISWC*, pp. 5–9, 2006.

[83] H. Knublauch, "Ontology-driven software development in the context of the semantic web: An example scenario with protege/owl," in *1st International workshop on the model-driven semantic web (MDSW2004)*, pp. 381–401, Monterey, California, USA.[WWW document] http://www. knublauch. com/publications/MDSW2004. pdf, 2004.

[84] M. I. S. R. Association *et al.*, *MISRA C++: 2008: guidelines for the use of the C++ language in critical systems*. MIRA, 2008.

[85] S. Pinto, J. Martins, J. Lopes, M. Abreu, and A. Tavares, "Secssy hypervisor: Security-safety synergy for aerospace,"

[86] L. Bettini, *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.