Real-Time Systems (2023) 59:534–567
https://doi.org/10.1007/s11241-023-09410-4

Evaluating virtualization for fog monitoring of real-time applications in mixed-criticality systems

Marcello Cinque[1] · Luigi De Simone[1] · Nicola Mazzocca[1] · Daniele Ottaviano[1] · Francesco Vitale[1]

Accepted: 23 September 2023 / Published online: 1 November 2023

Abstract
Technological advances in embedded systems and the advent of fog computing led to improved quality of service of applications of cyber-physical systems. In fact, the deployment of such applications on powerful and heterogeneous embedded systems, such as multiprocessors system-on-chips (MPSoCs), allows them to meet latency requirements and real-time operation. Highly relevant to the industry and our reference case-study, the challenging field of nuclear fusion deploys the aforementioned applications, involving high-frequency control with hard real-time and safety constraints. The use of fog computing and MPSoCs is promising to achieve safety, low latency, and timeliness of such control. Indeed, on one hand, applications designed according to fog computing distribute computation across hierarchically organized and geographically distributed edge devices, enabling timely anomaly detection during high-frequency sampling of time series, and, on the other hand, MPSoCs allow leveraging fog computing and integrating monitoring by deploying tasks on a flexible platform suited for mixed-criticality software, leading to so-called mixed criticality systems (MCSs). However, the integration of such software on the same MPSoC opens challenges related to predictability and reliability guarantees, as tasks interfering with each other when accessing the same shared MPSoC resources may introduce non-deterministic latency, possibly leading to failures on account of deadline overruns. Addressing the design, deployment, and evaluation of MCSs on MPSoCs, we propose a model-based system development process that facilitates the integration of real-time and monitoring software on the same platform by means of a formal notation for modeling the design and deployment of MPSoCs. The proposed notation allows developers to leverage embedded hypervisors for monitoring real-time applications and guaranteeing predictability by isolation of hardware resources. Providing evidence of the feasibility of our system development process and evaluating the industry-relevant class of nuclear fusion applications, we experiment with a safety-critical case-study in the context of the ITER nuclear fusion reactor. Our experimentation involves the design and evaluation of several prototypes deployed as MCSs on a virtualized MPSoC, showing that deployment choices linked to the

Extended author information available on the last page of the article

monitor placement and virtualization configurations (e.g., resource allocation, partitioning, and scheduling policies) can significantly impact the predictability of MCSs in terms of Worst-Case Execution Times and other related metrics.

Keywords Virtualization · MPSoCs · Fog computing · Monitoring · Anomaly detection · Fusion science

1 Introduction

The tight integration between physical and cyber processes in industrial applications has been facilitated by the technological advances in embedded systems, allowing cyber-physical systems' data collection and analysis (Pivoto et al. 2021).

Modern industrial applications design the aforementioned integration according to the cloud and fog computing paradigms, which organize the overall system as a hierarchy of geographically distributed nodes, whose responsibilities depend on their hierarchical level. These nodes involve data collection, analysis, and exchange with other levels of the hierarchy, leading to dynamics that Bittencourt et al. (2018) label as data flow in the IoT-fog-cloud continuum. The opportunity to distribute data-related tasks to the fog level, which is closer to edge devices tightly linked to physical plants, allows deploying intelligent, latency-sensitive services, such as diagnosis and control of physical phenomena and monitoring of software behavior.

As the criticality of the system being developed increases, the requirements for its applications become more demanding. A running example, and our case-study, is the challenging field of nuclear fusion, which is one of the most promising for highly-efficient energy harvesting, addressing the concerns for energy needs of the next century (EUROfusion 2018). In fact, nuclear fusion requires control systems to combine KHz-level actuation rates and real-time anomaly detection and recovery in order to operate with high assurance. Taking into consideration also other requirements, such as performance, scalability, interoperability, and reconfigurability through fast and efficient deployment, cyber-physical systems must deploy monitoring and anomaly detection for fault tolerance, resorting to increasingly advanced hardware, which requires resource management software to assure not only performance but also isolation and timeliness.

Although traditional monitoring strategies improve dependability through failure detectors and replicas' management (Kshemkalyani and Singhal 2011), network partitioning of distributed applications may impact the capability of nodes to communicate with monitors, hinder timely communication of nodes' conditions, and impair message ordering (Coulouris et al. 2011). By shortening the distance among interacting nodes and addressing connectivity issues, fog-based monitoring (hereafter fog monitoring) can address the aforementioned challenges by leveraging the distribution of monitors across edge devices deployed according to fog computing (Costa et al. 2022).

Being high-performance and heterogeneous embedded systems that provide multiple and diverse processing units (e.g., general-purpose and real-time processors), storage supports, communication protocols, and virtualization support (e.g. Xilinx Zynq Ultrascale+, NXP S32V234), Multiprocessor System-on-Chips (MPSoCs) are spreading in industrial scenarios (Ungurean and Gaitan 2021; Alonso et al. 2021), such as the previously mentioned class of nuclear fusion applications (Avon et al. 2021), and can be successfully deployed for fog monitoring on MCSs. In fact, these boards allow deploying several applications on a common platform, resulting in reduced size, weight, power, and cost (SWaP-C). This supports the development of Mixed-Criticality Systems (MCSs), which involve the deployment of heterogeneous applications, both characterized by different criticality requirements and running on the same physical board. Overall, the deployment of MCSs on MPSoCs opens the opportunity to address key non-functional requirements, such as scalability and interoperability.

1.1 Motivation

Despite the outlined advantages of fog monitoring and MPSoCs, existing research still lacks a model-based development process to design, deploy and evaluate the predictability of fog monitoring of real-time control over MPSoCs. Considering that model-based development allows dealing with complexity, verifying and validating the application's design, and supporting automated deployment (De Saqui-Sannes et al. 2022; Chardet et al. 2018), it may guide developers to correct management of shared hardware resources, limiting runtime interference of mixed-criticality software through the systematic application of design and deployment models and techniques. Please note that, in the following, interference refers to the performance impact that different software tasks experience when a software task's performance is affected by the activities or resource usage of other software tasks that are independent but nonetheless run on the same physical platform.

Promoting the isolation of different application contexts inside Virtual Machines (VMs) and managing multiple accesses to shared hardware resources (Hughes and Awad 2019; Cinque et al. 2021), virtualization is a key design technique to leverage fog monitoring, as it allows mitigating predictability issues when deploying monitoring software on the same board where other real-time tasks are running.

1.2 Contributions

In light of the evidence on challenges in deploying fog monitoring for MCSs on MPSoCs, model-based development benefits, the opportunities that virtualization opens, and the goal of addressing the challenging field of nuclear fusion, which requires predictable control of critical operations, our work provides the two following contributions:

(1) A model-based system development process to design and deploy MCSs on virtualized MPSoCs and a new formalism to model MCS deployment, facilitating the integration of real-time applications and fog monitoring on the same board;

(2) The experimentation with the academic- and industry-relevant ITER case-study, namely real-time stability control of a fully ionized gas, called plasma, within the ITER tokamak, an experimental nuclear fusion reactor under construction in Cadarache, France, where our team is involved in system engineering activities.

Our experimentation shows that the deployment of hypervisor-managed shared memory as the communication channel among VMs that run monitoring software and real-time tasks provides the best trade-off between performance and predictability, as it achieves both low and predictable communication latency for timely anomaly detection and assures the least interference between real-time operation and monitoring. In fact, control and monitoring tasks communication through hypervisor-managed shared memory makes the Worst-Case Execution Time (WCET) of control only 7.14% worse with respect to our reference baseline, which involves control with neither virtualization nor monitoring. On the other hand, the use of network sockets for communication with a remote server causes a worsening of control task WCET by up to 207.14%.

1.3 Paper structure

The rest of this paper is organized as follows: Sect. 2 provides background on model-based development, monitoring, fog computing, and virtualization; Sect. 3 presents the proposed MCS deployment model that we formalize and recommend using during the design of MCSs; Sect. 4 outlines the model-based system development process previously mentioned, which covers three main stages that detail all steps needed to design and deploy fog monitoring for MCSs on virtualized MPSoCs; Sect. 5 shows a practical application of the system development process to the ITER case-study and evaluation of the predictability of the resulting MCS; Sect. 6 analyzes existing work in several areas addressed in this work; and Sect. 7 summarizes the contents and results of the paper, briefly mentioning future research directions.

2 Background

2.1 Model-based development

Having shown promising results in driving the development of dependable systems, model-based development principles can be proficiently applied to the development of MCSs, which require the ability to isolate mixed-criticality software and manage access to shared resources.

The principles that model-based development carries allow practitioners to deal with the complexity of modern computer systems. Indeed, such principles promote modularity as a key design property, leading to system architectures whose requirements' traceability, correctness verification and validation practices, and maintenance routines are improved. Such improvements are due to semi-formal and formal modeling notations, which provide designers with a plethora of options to capture systems' requirements and prove the resulting architecture satisfies such requirements, facilitating the certification of the system with respect to industry standards (De Saqui-Sannes et al. 2022).

Regarding system development processes, there are many proposals in the literature, starting from the traditional waterfall model to the most recent agile development strategies. Still, coping with changes is a concept that developers should take into account, as the target system's requirements may be uncertain, leading to changes in design space options and environmental conditions. Among the available strategies, model-based system prototyping allows developers to demonstrate concepts, try out design options, and find out more about the problem at hand (Sommerville 2016).

In light of the above, the application of model-based development is promising for MCSs on virtualized MPSoCs, though its application is not straightforward: the possibility of fragmenting the design across several different models, mistakes made during modeling, and a lack of automated tools for integrated validation and verification of the resulting design can impair the benefits of model-based development (Quamara et al. 2021).

2.2 Monitoring

There are several ways monitoring can be characterized and classified. For our purposes, we detail here two dimensions: the monitoring placement and abstraction level.

Monitors can be placed as: software running on top of a virtualized layer (e.g., an operating system or a hypervisor); additional pluggable hardware components; and integrated on-chip hardware. These are termed software, hardware, and on-chip monitors, respectively (Watterson and Heffernan 2007). Complex monitoring approaches could leverage combinations of these deployment solutions to reach the desired architecture and meet the system requirements.

The kind of collected data, the way it is analyzed, the results provided, and the possible recovery actions to apply, depend on the abstraction level monitors work at. According to the taxonomy proposed in reference (Taherizadeh et al. 2018), abstraction levels can be: linked to performance of services that the application provides; end-to-end networking information; and hardware resources usage. These are termed application, end-to-end link quality, and VM/container-level monitoring, respectively.

Although Taherizadeh et al. (2018) consider application-level behavior as time metrics linked to services provided by applications, such behavior may also be linked to activities/state transitions that software that implements such services executes/experiences throughout its execution to meet functional requirements. Moreover, behavioral requirements may be described through semi-formal and formal behavioral notations, e.g., UML Activity Diagrams and Petri nets (De Saqui-Sannes et al. 2022).

Monitoring the activities and state transitions to connect the system's runtime behavior to prescriptive requirements involves the placement of logging rules within the deployed software and/or the network. However, such logging rules may not be integrated into modern systems, whose behavior is mainly monitored through time series and requires techniques for the extraction of activities and state transitions (Singh et al. 2022).

In the following, we consider application-level behavior as activities/state transitions that applications execute/experience, application-level data as data linked to activities/state transitions, and application-level monitoring as the collection of activities/state transitions to check their compliance to nominal patterns encoded with semi-formal and/or formal behavioral notations.

2.2.1 Fault tolerance

The development of fault-tolerant computer systems through suitable techniques and technology is a consolidated strategy to meet dependability requirements (Avizienis et al. 2004). Indeed, fault removal through white- and black-box testing cannot prove a system to be fault-free, especially when unexpected runtime conditions may invalidate its behavior (Delgado et al. 2004).

Fault tolerance through redundancy, failure detectors, and, when possible, software mechanisms (e.g., exception handling), have shown to improve system dependability (Coulouris et al. 2011). Moreover, considering the important role of monitoring to achieve fault tolerance, Delgado et al. (2004) survey behavior specification languages, monitor types, recovery actions, and operational issues linked to monitoring. In light of this, modern solutions integrate monitoring for several different abstraction levels to develop fault-tolerant systems and meet dependability requirements (Taherizadeh et al. 2018).

2.2.2 Data-driven anomaly detection

Anomalous behavior may be detected out of data collected through monitoring of computer systems and may indicate the early presence of activated faults, whose timely detection can trigger recovery actions and avoid failures Chandola et al. (2009). More precisely, anomalies “are patterns in data that do not conform to a well-defined notion of normal behavior" Chandola et al. (2009). Therefore, their detection requires a model of the nominal (normal) behavior of the target system, and the runtime comparison of such behavior with data collected as the system is exercised. Both these tasks are often achieved through data-driven techniques, which allow the extraction/parameters-tuning of a behavioral model encoding nominal behavior and comparing the behavior of the system in unknown conditions against the nominal model.

Considering nominal behavior is modeled through data collected as the target system runs in normal conditions, the resulting model may not be a semi-formal or formal model, e.g., an UML Activity Diagram or a Petri net, but may be encoded differently. Although obtained directly from data, in the following, we also consider such models as behavioral.

There are many supervised and unsupervised approaches to data-driven anomaly detection of time series. These could be based on Markov chains (Dong et al. 2018), (deep) neural networks (Ding et al. 2019), and process mining (Hemmer et al. 2021). Despite reaching satisfying detection performance, these approaches often need resource-rich environments to account for their time and memory complexity. In fact, their deployment in online and resource-constrained environments requires optimizations to improve their efficiency, which may hinder their detection performance (Verma et al. 2021).

Finally, although optimizations have been discussed to deploy anomaly detection in resource-constrained environments, its predictability, which is a key challenge for fog monitoring (Sánchez et al. 2022), is rarely addressed.

2.3 Fog computing and platforms

Stemming from requirements of modern cyber-physical systems, which often concern timely service provision in real-time applications (Sánchez et al. 2022), fog computing emerged as a leading architectural style for the development of such systems, as it involves design choices that distribute storage and computation to (fog) nodes close to devices, meeting applications' requirements that require timely service provision (Bellavista et al. 2019).

Therefore, deploying complex layered architectures based on fog computing enables the provision of services with reduced network latency, which are not possible with the standard Cloud-to-Thing networking (Wang et al. 2019);

The deployment of many efficient and well-equipped embedded devices opened the opportunity to move smart computations involving data-driven techniques, such as machine- and deep-learning algorithms, closer to users of the system (Bzai et al. 2022). This led researchers to steer towards studying how modern solutions can be adapted and implemented within these devices to offer services offered exclusively through the cloud.

2.3.1 Multiprocessor system-on-chips

Although developers have many design options when deploying systems according to fog computing, there is a remarkable class of platforms: MPSoCs, which are embedded systems hosting heterogeneous computation resources such as multi-core, device accelerators (e.g. GPUs, FPGAs, DPUs), rich equipment of communication interfaces, efficient management, and hardware security features (Ungurean and Gaitan 2021; Alonso et al. 2021).

The integration of multi-core CPUs ensures a high potential for complex calculations and reliability through redundancy. Moreover, reprogrammable hardware, typically implemented through FPGAs, introduces the advantage to deploy custom hardware design to accelerate time-sensitive tasks. Moreover, MPSoCs can also support fully-fledged operating systems and hypervisors, as well as being fully extendable and compliant with standard software architectures with respect to protocol stacks, devices, and infrastructure.

Meeting the requirements that fog nodes involve, such as substantial computing power, storage capability, and virtualization support (Puliafito et al. 2019), MPSoCs have been employed in industry as fog nodes to integrate several applications, which are usually distributed over many boards, on a single powerful board in order to optimize SWaP-C requirements (Barbalace et al. 2011; Dubbioso 2022).

Migrating from system designs with many interconnected single-core chips to applications consolidated onto a small number of MPSoCs clashes with the complexity given by isolating those applications on the same hardware, especially in the case of applications with safety-critical requirements. Indeed, providing hard real-time guarantees for tasks running on multi-core systems is an open problem in literature (Maiza et al. 2019), which outlines main problems are due to memory hierarchy contention (Yao et al. 2015; Agrawal et al. 2018).

Furthermore, these problems worsen when contention with hardware occurs not only between cores but also because of hardware accelerators. Therefore, researchers have formulated analytical models to reduce the pessimism in the timing analysis of complex heterogeneous systems (Hassan and Pellizzoni 2020; Houdek et al. 2017).

Finally, in order to implement through software the mechanisms for isolation needed by these heterogeneous architectures, virtualization is foreseen as an interesting solution (Sohal et al. 2022; Modica et al. 2018)

2.4 Virtualization

Virtualization is used to provide resource scalability, flexible deployment, and mixed-criticality over MPSoCs. It enables the deployment of several heterogeneous operating systems on the same physical hardware, provided the software architecture integrates a component called Hypervisor, or Virtual Machine Monitor (VMM), which has complete control of the hardware resources and meets the criteria described by Popek and Goldberg about equivalence, safety, and performance (Popek and Goldberg 1974).

All operating systems and applications expect the behavior to be identical to the one of a non-virtualized physical machine; this requires the hypervisor to both keep complete isolation between VMs and guarantee performance that is comparable to that of a non-virtualized environment. If these requirements are met, the use of virtualization can ensure the efficiency, flexibility, isolation, and portability of software applications.

2.4.1 Embedded virtualization

Virtualization is already a well-established technology for cloud computing. However, the current trend is to bring virtualization advantages over embedded systems to accomplish the next-generation requirements of modern industrial applications (Cilardo et al. 2022; Kao 2020).

However, the transition from cloud to embedded virtualization is not easy due to the high heterogeneity of embedded hardware devices and real-time, safety-critical requirements of cyber-physical systems applications. Researchers are putting more and more effort into developing technologies capable of increasing the temporal and spatial isolation provided by the VMM while keeping full utilization of all available resources as much as possible (Modica et al. 2018).

Therefore, virtualization may be a solution to multi-core isolation, as the hypervisor enables the software-based partitioning of shared hardware resources, such as cache, memory bandwidth, and DRAM to reduce interference between VMs. (Kloda et al. 2019).

2.4.2 Real-time hypervisors

The predictability problem through temporal and spatial isolation for the development of MCSs via virtualization has seen strong interest from both academia and industry, leading to a number of open-source and commercial virtualization solutions (Cinque et al. 2021).

A few virtualization solutions are specifically designed for embedded environments and are usually based on micro-kernel (Klein et al. 2009; Steinberg and Kauer 2010) or separation kernel architectures (Siemens 2022; Cotroneo et al. 2021).

Other interesting solutions rely on ARM TrustZone (Pinto and Santos 2019), which guarantees hardware support for isolation between a safe and a non-safe world for security requirements.

Furthermore, two well-known general-purpose hypervisors have been modified to guarantee real-time requirements: KVM (Kivity et al. 2007) and Xen (Barham et al. 2003). These are widely deployed in server environments, but in recent years they have been modified to run on embedded systems and to meet more stringent time requirements. For instance, Xen has been ported to ARM architectures, optimized to achieve a small footprint and be more easily verified (Stabellini 2014). In addition, various real-time schedulers, such as the Real-Time Deferrable Server (RTDS), the null scheduler, and the ARINC-653 cyclic scheduler (Wiki.Xenproject 2019; Linux Foundation 2015) have been included in the Xen project.

For all these reasons, Xen has been chosen as the hypervisor of choice for the experimental campaign carried out during this work.

3 The proposed MCS deployment model

This section presents the proposed MCS deployment model we adopt to describe the deployment of MCSs on virtualized MPSoCs, leveraging fog computing.

Our model describes the deployment of MCSs as architectural scenarios, which are quadruplets (ENV_L, ENV_R, SS, CS) where ENV_L is the local environment, ENV_R is the remote environment, SS is the scheduling scheme, i.e., the scheduling algorithms used to schedule VMs and tasks within VMs, and CS is the inter-VM communication scheme, i.e., the communication channels VMs use to communicate with each other.

Please note that from now on, VMs are considered as sets of tasks, where each VM isolates tasks from other VMs.

The aforementioned quadruplet is obtained by a mapping function *AS_MAP* that combines tasks, computational resources, scheduling algorithms, and inter-VM

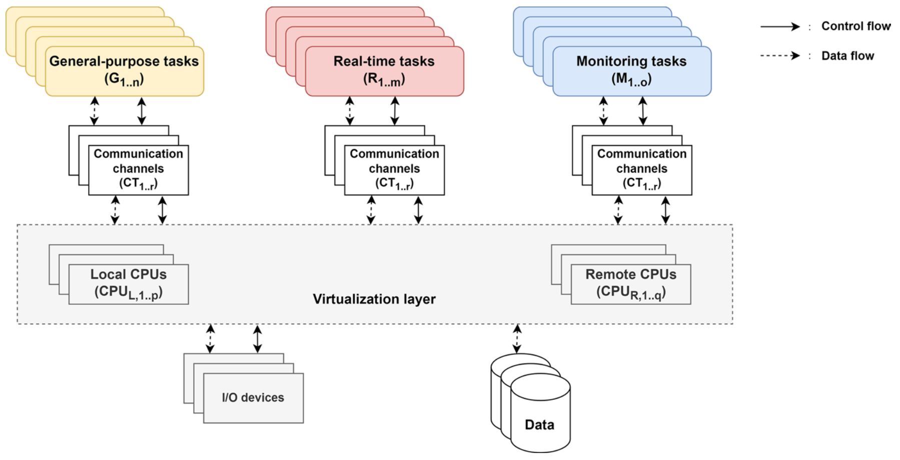

Fig. 1 The elements that the proposed MCS deployment model handles

communication channels together, configuring architectural scenarios. Figure 1 collects all the elements here discussed; the way they are arranged together depends on the *AS_MAP* function, which configures the virtualization layer and the interconnection among tasks, I/O devices, and data repositories.

3.1 Global task set

The global task set (*TS*) is the set of all tasks of an MCS:

$$TS = RT \cup M \cup G$$

where *RT* is the set of real-time tasks, *M* is the set of monitoring tasks, and *G* is the set of all other general-purpose tasks.

3.2 Local and remote environments

Our model splits MCSs into two environments: the local (ENV_L) and remote (ENV_R) environments, where the ENV_L environment is a fog node with substantial hardware resources and virtualization support (e.g., an MPSoC), and the ENV_R environment is the cloud.

The set of computational resources (*CR*) groups Processing Elements (PEs) from both environments:

$$CR = PE_L \cup PE_R,$$

where a PE is a generic processing unit that MPSoCs ship with, such as CPUs, GPUs, FPGAs, and DPUs (see Sect. 2.3). PE_L and PE_R are the sets of local and remote PEs, respectively.

Given that we abstract VMs as sets of tasks, the global set of VMs $VM_G \subset \mathcal{P}(TS)$ is such that $\bigcup VM_G = TS \wedge \bigcap VM_G = \emptyset$, i.e. no task can belong to more than one VM and the union of all VMs is TS. Moreover, we split VMs into local and remote VMs (VM_L and VM_R, respectively):

$$VM_L \subseteq VM_G, VM_R \subseteq VM_G,$$

where $VM_L \cup VM_R = VM_G$, i.e. the union of local and remote VMs is the global set of VMs

In order to fully define environments, we consider PE pools. A PE pool (P) is an element of power sets of either PE_L or PE_R ($P \in \mathcal{P}(PE_L) \cup \mathcal{P}(PE_R)$), and represents a group of physical PEs isolated from other PEs ($P_i \cap P_j = \emptyset, i \neq j$).

Given $PS_L \subset \mathcal{P}(PE_L)$ and $PS_R \subset \mathcal{P}(PE_R)$ the two sets of local and remote PE pools (i.e., the two local and remote pooling schemes), ENV_L and ENV_R are defined as follows:

$$ENV_L = \{DM_{i,L} = (VM, P) \in VM_L \times PS_L\}$$
$$ENV_R = \{DM_{i,R} = (VM, P) \in VM_R \times PS_R\}$$

where $DM_{i,R}$ is the i-th deployment module of the x environment. If $\|ENV_x\| = 1$, the environment x is made of only one VM and one PE pool, i.e., environment x is non-virtualized.

We here note that during evaluation (Sect. 5) we consider CPUs as the only PEs. Therefore, in the following we consider the set of computational resources as the collection of local and remote CPUs, i.e. $CR = CPU_L \cup CPU_R$.

3.3 Scheduling

Concerning both tasks within VMs and VMs themselves, we split scheduling algorithms according to the hierarchical scheduling taxonomy (Biondi et al. 2015; Lee et al. 2012).

Thus, we split scheduling algorithms into two categories: local scheduling algorithms (S_L) for tasks within VMs, and, global scheduling algorithms (S_G) for VMs assigned to a given CPU pool.

Therefore, the scheduling scheme SS_{DM} for a given deployment module ($DM \in ENV_L$) is a pair ($S_{L_{DM}}, S_{G_{DM}}$), where $S_{L_{DM}}$ and $S_{G_{DM}}$ are the local and global schedulers linked to DM, respectively. The set of all scheduling schemes of all deployment modules is labeled as SS.

Defining the scheduler at deploying time is important for the designer since the choice of the scheduler affects the number of deployable VMs. If a partitioning hypervisor is used, the global scheduling is null and the number of VMs is limited by the number of CPUs. On the other hand, the number of VMs can be higher than the number of CPUs if the hypervisor uses a traditional scheduler.

Starting from the generated deployment model, the designer can later employ an analytical timing analysis method that aligns with the scheduling taxonomy that

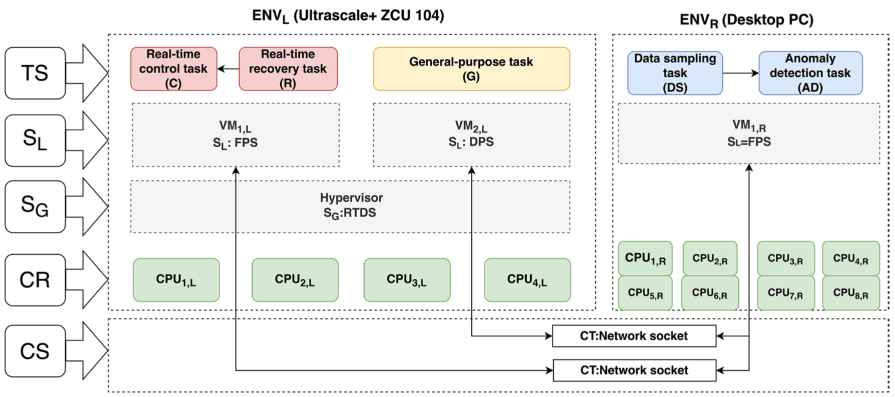

Fig. 2 A sample architectural scenario

has been chosen during the deployment phase (e.g., hierarchical scheduling (Biondi et al. 2015)).

3.4 Inter-VM communication

Inter-VM communication channels (*CT*) are ways VMs communicate with each other. The inter-VM communication scheme *CS* is a set of triples defined as follows:

$$CS = \{(ct, VM_1, VM_2) \in CT \times (VM_L \cup VM_R) \times (VM_L \cup VM_R)\}$$

where *CT* is the set of communication channels, such as hypervisor-managed shared memory or network sockets.

3.5 Mapping function

The mapping function *AS_MAP* is defined as follows:

$$AS = AS\_MAP(TS, CR, S_L, S_G, CT)$$

where $AS = (ENV_L, ENV_R, SS, CS)$, i.e., the architectural scenario, which collects the: local and remote environments (ENV_L and ENV_R); scheduling schemes (*SS*); and inter-VM communication scheme (*CS*).

In order to clarify the application of *AS_MAP*, we depict in Fig. 2 a sample architectural scenario commonly found in distributed applications and described through the MCS deployment model. In this case, the elements are:

- *TS*: One general purpose task (*G*), two real-time tasks (*R* and *C*), and two monitoring tasks (*DS* and *AD*)
- *CR*: Local CPUs of the Ultrascale+ ZCU 104 MPSoC and remote CPUs of a Desktop PC

Table 1 Local and remote VMs, CPU pools, DMs, and scheduling schemes of the sample architectural scenario

Local environment			
VM_L	P_L	DM_L	SS_{DM_L}
$1,L : \{R, C\}$ $2,L : \{G\}$	$1,L : \bigcup_{i=1\ldots p} CPU_{i,L}$	$1,L : (VM_{1,L}, P_{1,L})$ $2,L : (VM_{2,L}, P_{1,L})$	$DM_{1,L} : (RTDS, DPS)$ $DM_{2,L} : (RTDS, FPS)$
Remote environment			
VM_R	P_R	DM_R	SS_{DM_R}
$1,R : \{DS, AD\}$	$1,R : \bigcup_{i=1\ldots p} CPU_{i,R}$	$1,R : (VM_{1,R}, P_{1,R})$	$DM_{1,R} : (-, FPS)$

Table 2 The deployment quadruplet of the sample architectural scenario

Deployment quadruplet			
ENV_L	ENV_R	SS	CS
$\{DM_{1,L}, DM_{2,L}\}$	$DM_{1,R}$	$\{SS_{DM_{1,L}}, SS_{DM_{2,L}}, SS_{DM_{1,R}}\}$	$\{(Socket, VM_{1,L}, VM_{1,R})\}$

- S_L and S_G: the local schedulers of VMs deployed on top of the Xen hypervisor and its global schedulers to orchestrate such VMs to handle their access to shared hardware resources, and the local schedulers of an Ubuntu-based OS
- CT: Network sockets

The way these elements are laid out by *AS_MAP* is described in Tables 1 and 2, which collect local and remote VMs (VM_L and VM_R), local and remote CPU pools (P_L and P_R), local and remote deployment modules (DM_L and DM_R), the scheduling scheme of each local and remote deployment module (SS_{DM}), and the corresponding quadruplet, i.e., the local and remote environments (ENV_L and ENV_R), the scheduling scheme (SS), and the communication scheme (CS).

4 The proposed model-based system development process

Following the driving principles outlined in Sect. 2.1, we provide an overview of the model-based system development process we propose to integrate real-time applications and application-level fog monitoring in virtualized environments, facilitating (1) The deployment of MCSs on virtualized MPSoCs by leveraging the MCS deployment model presented in the previous section and change management in the requirements and (2) Design of the application through an iterative approach.

The process is split into three main steps, which are: system description and specification; system architecture design, when developers may leverage the proposed MCS deployment model; and prototype development. These steps are described in the flow diagram shown in Fig. 3 and are detailed in the following.

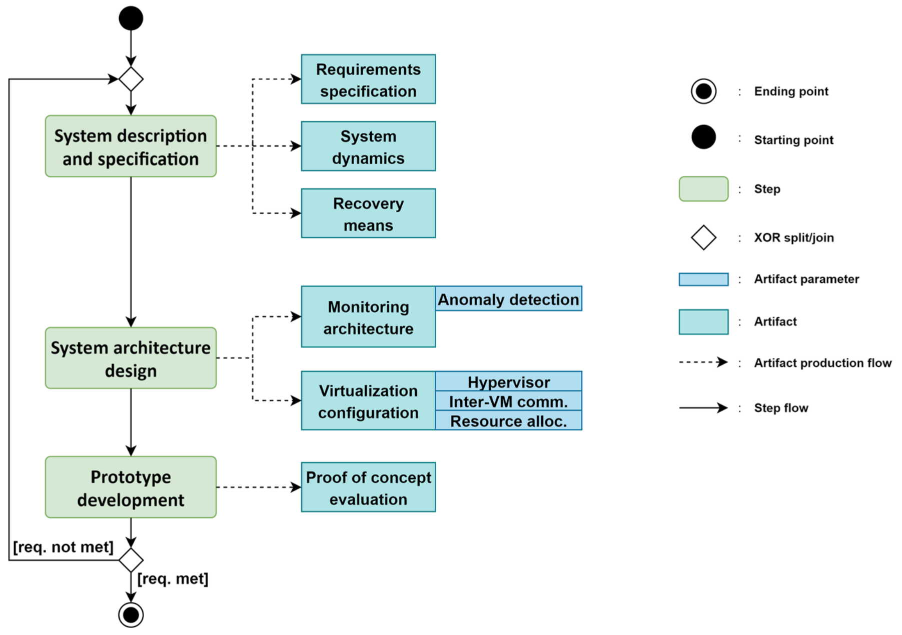

Fig. 3 The proposed system development process steps, flow, and artifacts

4.1 System description and specification

This step requires collecting all specifications about the real-time application. The resulting requirements specification document should focus on the non-functional requirements the application must meet, first and foremost predictability, followed by performance requirements needed for monitoring, both in terms of accuracy and latency.

Typically, the aforementioned requirements are found in standards such as avionics (DO-178C (RTCA 2012), ARINC-653 (AEEC 2010)), automotive (ISO 26262 (ISO 2011)), and others (e.g., ISO 61508 (I.E 1998), EN 50128 (CENELEC 2011), etc.)

Requirements specification must include prescriptions about application-level behavior (see Sect. 2.2). Therefore, these need to be modeled, providing a prescription of the system dynamics that the application is expected to show throughout its execution.

As pointed out in Sect. 2.2, the description of nominal behavior can be carried out through suitable modeling notations, such as Markov chains, Petri nets, and queue networks, or else, be directly obtained from data as the system is exercised in normal conditions. Please, in addition to Sect. 2.2, also refer to Sect. 6.2 for an overview of monitoring taxonomies, anomaly detection techniques for data-driven nominal behavioral modeling and detection of behavioral anomalies, and existing application-level monitoring approaches in the literature.

Once the system's nominal behavior is profiled through semi-formal or formal notations, recovery actions must be defined to manage and respond to critical situations that may invalidate the application's dependability requirements.

Finally, as requirements need to be unambiguously validated, the set of metrics to measure in order to check whether the system meets the requirements here outlined are captured and referenced in the last phase of the process, which prescribes carrying out a proof-of-concept evaluation of the system prototype.

For instance, typical metrics used for tasks of real-time applications are their WCET, Best-Case Execution Time (BCET), Average-Case Execution Time (ACET), Tail Latency (TL), and Standard Deviation (std). In the case of an MCS leveraging virtualization, all the interference caused by the chosen hypervisor and the blocking time caused by the scheduling of different VMs must be considered by both analytical considerations and prototype assessment of these metrics (e.g. (Zhao et al. 2018; Biondi et al. 2015)).

4.2 System architecture design

Designing the architecture of a system in a model-based fashion requires the description of several different views, which, in turn, collect multiple models. We recommend using the MCS deployment model presented in Sect. 3, as it promotes model-based development benefits, e.g., modularity, and minimizes issues, e.g., model fragmentation (see Sect. 2.1).

Among the aspects the model allows describing, there are:

(1) The *monitoring architecture*: the way application-level monitoring is performed.
(2) The *virtualization configuration*: the remote (cloud) and local (fog) environments layout, the distribution of VMs across these environments and their scheduling policies, and the communication scheme that VMs follow to exchange data and control signals.

Concerning requirement 1., the monitoring architecture is the way available software and hardware resources are configured in order to implement a monitoring solution that satisfies the requirements collected in the previous step, such as the need for separating communication channels when monitoring and how data should be communicated to monitors.

There are several monitoring architectures proposed in the literature, such as the single process monitor and the distributed process monitor (Goodloe and Pike 2010). The monitor architecture should also take into account the kinds of anomalies targeted, the monitoring level, and the anomaly detection technique to deploy (see Sect. 2.3).

Regarding requirement 2., the virtualization configuration describes the arrangement of resources and VMs to comply with both the requirements collected in the previous step and the monitoring architecture the MCS deploys.

Moreover, the configuration involves choices linked to the embedded hypervisor to deploy, the communication channels that VMs use, and scheduling algorithms (see Sect. 2.4 for further details).

Although we only focus on deployment aspects of system design, there are many other features that developers may take into account when designing MCSs on virtualized MPSoCs, e.g., interference among tasks when accessing shared hardware resources and end-to-end delays introduced by communication channels.

Modeling such features can guide developers to narrow the scope of the design space, limiting the number of architectural scenarios to be evaluated in the Prototype Development step. However, as analytical modeling of MCSs may not be appropriate or feasible at times, our proposal allows practitioners to experiment with several different architectural scenarios iteratively to evaluate which one fits the requirements best.

4.3 Prototype development

Provided with a set of architectural scenarios, the Prototype Development step involves the implementation of prototypes, aiming at validating a given MCS deployment against application-specific requirements. Thus, this validation step leads to a proof-of-concept evaluation, which is critical for several reasons: check whether the monitoring system is able to detect the presence of runtime conditions deviating from nominal behavior; incorrect system dynamics modeling during the system description and specification step; and unsatisfying monitoring and/or real-time processes performance/predictability.

It is worth noting experimentation may not be adequate to validate process predictability, as the sampled statistics for real-time tasks, such as their WCET or execution time variability, could not cover edge cases that were not targeted throughout experimentation.

5 Evaluation

In this section, we present and evaluate a safety-critical case-study that involves a system whose specifications require monitoring its application-level behavior while guaranteeing system predictability. We have considered developing prototypes of the system according to the system development process presented in the previous section.

The goals of the evaluation are:

- Showing a practical application of the model-based system development process and the MCS deployment model;
- Carrying out a proof-of-concept evaluation of system prototypes deployed according to different architectural scenarios, described using the proposed MCS deployment model;

- Showing that different architectural scenarios impact the predictability of the real-time application differently.

5.1 Case-study

The case-study deals with real-time control of a fully ionized gas, called plasma, within a tokamak, which is a nuclear fusion experimental reactor.[1]

As the global need for sustainable low-carbon sources of electricity increases, nuclear fusion represents one of the most promising technologies to meet this need (EUROfusion 2018).

One of the essential problems to be tackled in a tokamak reactor is control of vertically unstable plasma in order to prevent it to collapse on the reactor wall, as such an event may seriously damage the plant. Therefore, plasma must be controlled very precisely and any misbehavior must therefore be properly addressed.

5.1.1 System description and specification

Many challenging control loops must be implemented in order to operate a tokamak. By adjusting the magnetic field created by the currents flowing in numerous external coils, the magnetic control system is responsible for managing the current induced into the plasma, as well as its shape and position. The interested reader may refer to reference (Ariola and Pironti 2016) for more details on plasma magnetic control.

Unfortunately, the elongated shape of high-performance plasma pursued in modern tokamak turns them to be vertically unstable. Therefore, as anticipated, the magnetic control design must contain an active Vertical Stabilization (VS) system.

In this work, we are interested in monitoring the plasma during the application of the VS control algorithm. Specifically, we consider a real-time task executing a control strategy based on Extremum Seeking (ES) to vertically stabilize the plasma (Dubbioso 2022).

Despite the specific control algorithm adopted to realize the corresponding function, the VS system is one of the essential components of any Plasma Control System (PCS). Moreover, for any existing tokamak, including ITER, the PCS is not a standard component that is available off-the-shelf; hence its design represents itself one of the project challenges (Snipes 2021).

Although procedures for the assessment of the PCS performance requirements are envisaged (Walker 2019; De Tommasi 2022), no standards apply to specific safety-critical metrics for real-time control tasks when designing the PCS. However, there is significant interest in high-frequency and fine-grained monitoring of real-time control systems for performance, predictability, and safety to provide countermeasures in due time to address run-time anomalies. This is shown by the intensive exception management study being conducted in fusion projects such as ITER (Raupp 2014) and JET (Snipes 2021; Sohal et al. 2022), and by other works

[1] https://www.iter.org/sci/MakingitWork.

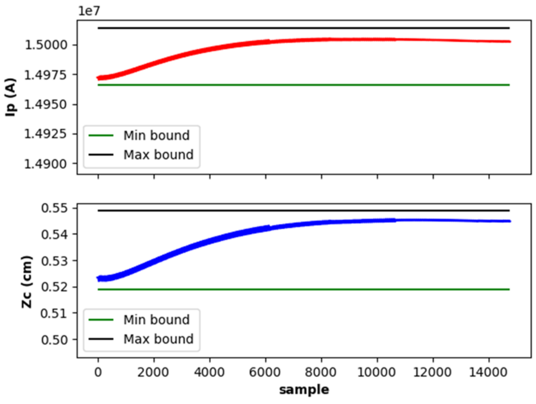

Fig. 4 Normal behavior of the plasma current *Ip* and vertical position of the plasma centroid *Zc*

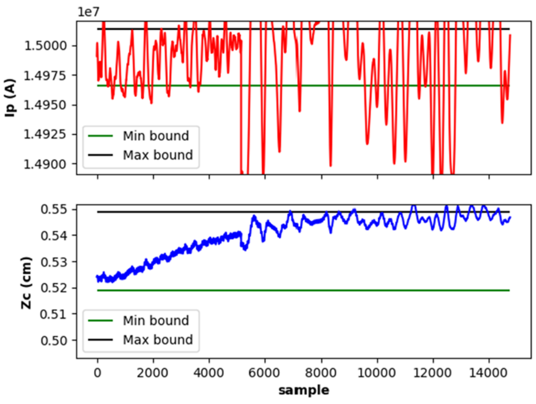

Fig. 5 Anomalous behavior of the plasma current *Ip* and vertical position of the plasma centroid *Zc*

dealing with the monitoring of hard real-time control algorithms for performance guarantees (Barbalace et al. 2011; Neto 2011), or with disruption prediction systems (Murari 2018; Vega et al. 2022).

Regarding VS, this algorithm ensures that the plasma column is kept around a given equilibrium, i.e. the vertical position of the plasma current centroid is bounded in a given range during operation. Moreover, other state variables, mostly involving currents flowing throughout the plasma and actuator circuits, must follow patterns that lead to correct steady-state behavior. Figure 4 shows the typical response of the controlled system to a sudden disturbance, modeled as downward Vertical Displacement Events (VDEs, (Ambrosino et al. 2010)); in particular, the time traces of the vertical position of the plasma centroid Z_c and of the plasma current I_p are shown. When VDEs exceed a given threshold, the plant may enter an operation regime that may lead to a plasma disruption, which, in turn, makes Z_c and I_p exceed normal bounds, seriously damaging the plant. Hence, prompt identification of a potentially dangerous regime should be put in place, to trigger the proper mitigation policies. This is shown in Fig. 5, which depicts anomalous Z_c and I_p dynamics due to VDEs exceeding safety thresholds.

The set of all normal dynamics characterizes the system's normal behavior to be used when comparing it to runtime behavior under unknown (possibly anomalous) conditions.

As Figs. 4 and 5 outline, application-level behavior is monitored through time series. This requires modeling normal behavior and checking the system at runtime for such behavior through data-driven anomaly detection for time series, of which we surveyed possible options from the literature (see Sect. 2.2).

Recovery actions are uniquely prescribed for each of the experimental designs in operation (e.g. ITER and JET (Raupp 2014; De Tommasi et al. 2014; Valcárcel et al. 2014)). In any case, the main concern for these projects is forcing the fail-safe behavior of the plant to prevent catastrophic consequences. As a result, the detection must happen within a time interval that depends on both the specific tokamak and the plasma configuration. This is to ensure that all safety shutdown processes are carried out on time so as to avoid any contact between the hot plasma and the blankets, i.e., the tokamak's walls. Indeed, a sudden shutdown can lead to faulty machine parts that are too costly and difficult to repair. However, it is worth noting that the exact procedures for plasma cooling during a fault have not yet been established for the experiments on ITER.

In order to check whether the requirements coming from fusion needs are met, such as timely application of control inputs, isolation of safety-critical tasks from others, and fault tolerance to runtime errors and failures, there are several metrics that can be evaluated. Among these, there are:

- $WCET_\tau$, the WCET of tasks;
- $ACET_\tau$, the ACET of tasks;
- $BCET_\tau$, the BCET of tasks;
- TL_τ, the TL of tasks, which is the 99th percentile of the execution time of tasks;
- std_τ, the std of tasks.

Due to the iterative nature of the system development process, it is worth noting that requirements can be refined incrementally. In fact, as the design space is explored and prototypes are developed in the System Architecture Design and Prototype Development steps, there may be failures that developers may initially not consider, such as crashes of critical tasks involved in, e.g., the execution of the ES algorithm during VS, application-level data sampling, or the application of recovery actions in response to anomalies. In the iteration we are considering, the concern is about the evaluation of the predictability of tasks in absence of other failures.

5.1.2 System architecture design

In order to comply with the requirements that were identified in the previous step, our system must have a real-time control task C, the ITER industrial plant C must control, and a set of local CPUs ($CPU_{1..p,L}$) that C runs on.

Additionally, we may consider:

- A set of general-purpose tasks $G = \{G_i, i \in \{1, \dots, n\}\}$, a real-time recovery task R, and two monitoring tasks $M = \{DS, AD\}$, in which DS concerns application-level data sampling and AD runs anomaly detection on sampled data
- A set of remote CPUs ($CPU_{1\dots q,R}$)
- A data repository to collect the nominal behavior of the ITER industrial plant
- A set of communication channels ($CT_{1\dots s}$).

As mentioned in Sect. 4.2, the design of the system can be driven using a plethora of models that refer to several of its different views. These models may guide designers in choosing deployments that aim to optimize a given goal function.

Taking into account the goal of evaluating changes in the system's predictability due to different deployment scenarios, we model architectural scenarios AS_0, AS_1, and AS_2 by applying the AS_MAP function of the MCS deployment model we presented in Sect. 3.

We extend each architectural scenario with a general-purpose task G deployed on ENV_L, which generates network disturbance by communicating with the Internet through a network socket. This is because a substantial amount of raw data, which reach TBs in ITER, are collected by sensors during the experiments and must be transferred to storage sources for later analysis to further assess the behavior of the plasma. Additionally, general-purpose, signal-processing tasks, which may send data to outbound servers for additional and non-urgent analysis in distributed scenarios, may use the same data collected during real-time control as well. Therefore, it is worthwhile deploying such tasks in the local environment. This leads to the d counterparts of the aforementioned architectural scenarios ($AS_{x,d}, x = \{0, 1, 2\}$), where there is the task G deployed on one of the VMs of ENV_L. We collect local and remote VMs and CPU pools, their corresponding DMs and scheduling, and the resulting quadruplets in Tables 3 and 4.

For the sake of clarity, we depict one of the architectural scenarios (AS_2) in Fig. 6, though it is worth noting the opportunity to formalize the scenario through a well-defined mathematical language makes graphical representations superfluous, eliminating ambiguity and reducing the time required to describe several different scenarios.

The metrics previously mentioned can now be specialized for the C and DS tasks of our design:

- $WCET_C$, $BCET_C$, $ACET_C$, TL_C, and std_C;
- $WCET_{DS}$, $BCET_{DS}$, $ACET_{DS}$, TL_{DS}, and std_{DS}.

As mentioned, these metrics drive the evaluation of the predictability of the prototypes deployed according to AS_0, $AS_{0,d}$, AS_1, $AS_{1,d}$, AS_2, and $AS_{2,d}$.

It is worth noting that access policies to shared memory are strictly dependent on the specific protocol chosen by the designer. Since this is out of the scope of this work, we decided to use a simple asynchronous protocol, which we briefly describe in the following.

The C task asynchronously sends data to the DS task within each of its execution periods. In turn, the DS task reads the newest data sent by C from the

Table 3 Local and remote VMs, CPU pools, DMs, and scheduling schemes of the designed architectural scenario

	Local environments			
	VM_L	P_L	DM_L	SS_{DM}
AS_0	$1,L : \{C\}$	$1,L\colon \bigcup_{i=1\ldots p} CPU_{i,L}$	$1,L : (VM_{1,L}, P_{1,L})$	$DM_{1,L} : (-, FPS)$
$AS_{0,d}$	$1,L : \{C\}$ $2,L : \{G\}$	$1,L\colon \bigcup_{i=1\ldots p} CPU_{i,L}$	$1,L : (VM_{1,L}, P_{1,L})$ $2,L : (VM_{2,L}, P_{1,L})$	$DM_{1,L} : (RTDS, FPS)$ $DM_{2,L} : (RTDS, FPS)$
AS_1	$1,L : \{C, R\}$	$1,L : \bigcup_{i=1\ldots p} CPU_{i,L}$	$1,L : (V, M_{1,L}, P_{1,L})$	$DM_{1,L} : (-, FPS)$
$AS_{1,d}$	$1,L : \{C, R\}$ $2,L : \{G\}$	$1,L : \bigcup_{i=1\ldots p} CPU_{i,L}$	$1,L : (VM_{1,L}, P_{1,L})$ $2,L : (VM_{2,L}, P_{1,L})$	$DM_{1,L} : (RTDS, FPS)$ $DM_{2,L} : (RTDS, FPS)$
AS_2	$1,L : \{C, R\}$ $2,L : \{DS, AD\}$	$1,L : \bigcup_{i=1\ldots p} CPU_{i,L}$	$1,L : (VM_{1,L}, P_{1,L})$ $2,L : (VM_{2,L}, P_{1,L})$	$DM_{1,L} : (RTDS, FPS)$ $DM_{2,L} : (RTDS, FPS)$
$AS_{2,d}$	$1,L : \{C, R\}$ $2,L : \{DS, AD\}$ $3,L : \{G\}$	$1,L : \bigcup_{i=1\ldots p} CPU_{i,L}$	$1,L : (VM_{1,L}, P_{1,L})$ $2,L : (VM_{2,L}, P_{1,L})$ $3,L : (VM_{3,L}, P_{1,L})$	$DM_{1,L} : (RTDS, FPS)$ $DM_{2,L} : (RTDS, FPS)$ $DM_{3,L} : (RTDS, FPS)$
	Remote environments			
	VM_R	P_R	DM_R	SS_{DM}
AS_0	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$AS_{0,d}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
AS_1	$1,R : \{DS, AD\}$	$1,R : \bigcup_{i=1\ldots q} CPU_{i,L}$	$1,R : (V, M_{1,R}, P_{1,R})$	$DM_{1,R} : (-, FPS)$
$AS_{1,d}$	$1,R : \{DS, AD\}$	$1,R : \bigcup_{i=1\ldots q} CPU_{i,L}$	$1,R : (V, M_{1,R}, P_{1,R})$	$DM_{1,R} : (-, FPS)$
AS_2	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$AS_{2,d}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

communication channel. The most important factor is to receive the newest data possible, even if it could imply the loss of some samples. For the same reason, in the scenario where network sockets are used as a communication technique, we decided to use the UDP protocol. Conversely, if two or more applications communicate synchronously, this may result in temporal variations, which should be

Table 4 The deployment quadruplets of the designed architectural scenarios

	Deployment quadruplets			
	ENV_L	ENV_R	SS	CS
AS_0	$\{DM_{1,L}\}$	$\emptyset$	$\{SS_{DM_{1,L}}\}$	$\emptyset$
$AS_{0,d}$	$\{DM_{1,L}, DM_{2,L}\}$	$\emptyset$	$\{SS_{DM_{1,L}}, SS_{DM_{2,L}}\}$	$\emptyset$
AS_1	$\{DM_{1,L}\}$	$\{DM_{1,R}\}$	$\{SS_{DM_{1,L}}, SS_{DM_{1,R}}\}$	$\{Socket, VM_{1,L}, VM_{1,R}\}$
$AS_{1,d}$	$\{DM_{1,L}, DM_{2,L}\}$	$\{DM_{1,R}\}$	$\{SS_{DM_{1,L}}, SS_{DM_{1,R}}\}$	$\{Socket, VM_{1,L}, VM_{1,R}\}$
AS_2	$\{DM_{1,L}, DM_{2,L}\}$	$\emptyset$	$\{SS_{DM_{1,L}}, SS_{DM_{2,L}}\}$	$\{SHM, VM_{1,L}, VM_{2,L}\}$
$AS_{2,d}$	$\{DM_{1,L}, DM_{2,L}, DM_{3,L}\}$	$\emptyset$	$\{SS_{DM_{1,L}}, SS_{DM_{2,L}}, SS_{DM_{3,L}}\}$	$\{SHM, VM_{1,L}, VM_{2,L}\}$

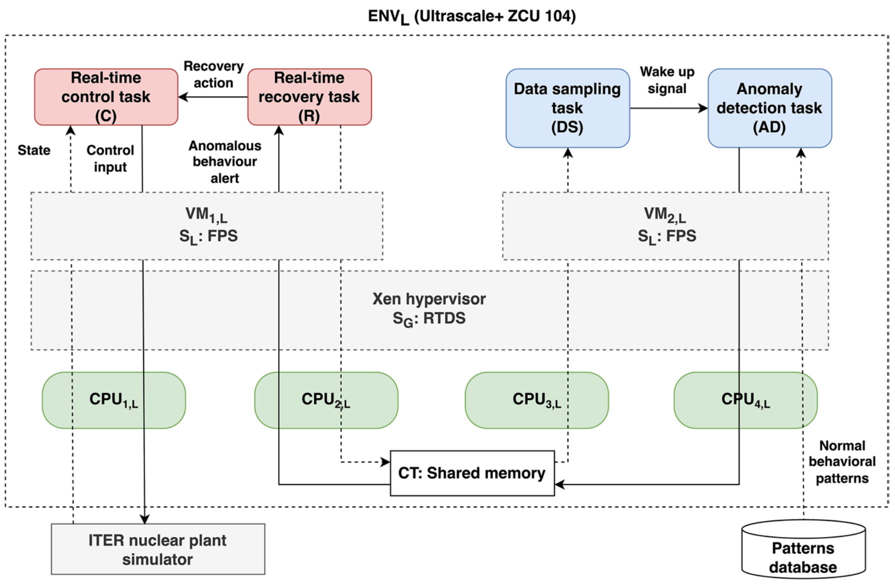
ENV_L (Ultrascale+ ZCU 104)
Real-time control task (C)
Recovery action
Real-time recovery task (R)
Data sampling task (DS)
Wake up signal
Anomaly detection task (AD)
State
Control input
Anomalous behaviour alert
VM_1,L
S_L: FPS
VM_2,L
S_L: FPS
Xen hypervisor
S_G: RTDS
CPU_1,L
CPU_2,L
CPU_3,L
CPU_4,L
CT: Shared memory
Normal behavioral patterns
ITER nuclear plant simulator
Patterns database

Fig. 6 The AS_2 architectural scenario

appropriately handled by the communicating tasks, regardless of the communication channel they send/receive data to/from.

Finally, anomaly detection, performed by task *AD* and based on process mining, allows characterizing the nominal behavior as locally-stored patterns and comparing new time series with such behavior at runtime (Hemmer et al. 2021). However, please note that the goal of our experimentation is not evaluating the predictability of process mining algorithms, which is an open challenge on its own. Rather, in the following, we evaluate the ability of each architectural scenario to isolate the non-deterministic nature of *AD*. In light of this, we do not consider any metric for *AD* in our experimentation.

5.1.3 Prototype development

The prototype for each of the designed architectural scenarios employs a Zynq UltraScale+ MPSoC ZCU104 as the node where ENV_L is deployed, whereas a workstation with an Intel(R) Core(TM) i7-4790 CPU, 16Gb RAM, and 512Gb HDD is used for ENV_R deployment. When virtualized, ENV_L is managed by the Xen hypervisor.

In order to generate normal and anomalous system dynamics, we used an ITER nuclear plant simulator that allows injecting VDEs. During the offline phase, VDEs are injected within normal ranges, allowing the storage of normative patterns as the ES algorithm runs and controls the plant to its equilibrium. These normative patterns are used during online monitoring and as the simulator runs together with the *C* task running the ES algorithm; throughout online monitoring, the nuclear plant is

Table 5 Metrics per scenario related to tasks C and DS

	$WCET_\tau(\mu s)$	$ACET_\tau(\mu s)$	$BCET_\tau(\mu s)$	$TL_\tau(\mu s)$	$std_\tau(\mu s)$
AS_0	$C: 14.00$ $DS: N/A$	$C: 13.07$ $DS: N/A$	$C: 11.00$ $DS: N/A$	$C: 14.00$ $DS: N/A$	$C: 0.90$ $DS: N/A$
$AS_{0,d}$	$C: 15.00$ $DS: N/A$	$C: 13.41$ $DS: N/A$	$C: 11.00$ $DS: N/A$	$C: 15.00$ $DS: N/A$	$C: 0.91$ $DS: N/A$
AS_1	$C: 43.00$ $DS: 14.00$	$C: 41.87$ $DS: 3.94$	$C: 11.00$ $DS: 1.00$	$C: 43.00$ $DS: 13.00$	$C: 0.97$ $DS: 3.27$
$AS_{1,d}$	$C: 69.00$ $DS: 33.00$	$C: 45.29$ $DS: 6.24$	$C: 12.00$ $DS: 1.00$	$C: 67.00$ $DS: 27.00$	$C: 4.12$ $DS: 6.31$
AS_2	$C: 15.00$ $DS: 3.00$	$C: 13.09$ $DS: 1.38$	$C: 10.00$ $DS: 1.00$	$C: 14.00$ $DS: 3.00$	$C: 0.88$ $DS: 0.59$
$AS_{2,d}$	$C: 20.00$ $DS: 15.00$	$C: 15.01$ $DS: 4.28$	$C: 11.00$ $DS: 1.00$	$C: 19.00$ $DS: 14.00$	$C: 1.85$ $DS: 3.28$

N/A: Not Applicable

injected both with normal and anomalous VDEs, so as the DS task collects samples from the C task, checking whether there are anomalies or not.

In each experiment, we have performed 50 runs per architectural scenario, where, for each run, the C task is a periodic task with period $1000\mu s$ and acquires, for each task instance, a sample from the ITER nuclear plant simulator while it runs under unknown conditions. Each sample is a vector of 5 floating point values, including the previously cited I_p and Z_p state variables.

In AS_0, these samples are not sent to any other task, whereas in AS_1 and AS_2 they are sent to the DS task through a network socket and shared memory, respectively.

Once 2000 samples are collected by DS, the AD task executes and classifies the behavior according to the normal patterns that were characterized during the offline phase. For each run, a total of 60000 samples are collected from the simulated plant, thus there are 60000 execution times collected from both the C and DS tasks.

Table 5 collects the metrics per scenario. These are computed globally, which means all execution times from all runs are considered at once and all metrics are computed accordingly. For instance, in order to compute $WCET_C$ for scenario AS_0, execution times from all 50 runs are collected and the maximum is computed.

In Fig. 7 the metrics per scenario are visualized as histograms to highlight how the distributions of execution times of tasks C and DS shift from one setup to another.

Figures 8a and b show violin plots of tasks C and DS per scenario in order to provide another view of the results with a greater focus on the variability of the data across scenarios.

The results presented show that each architectural scenario impacts the predictability of the system differently. As shown in Figs. 7 and 8, the execution times are generally higher when data between C and DS are sent across the network (AS_1). Furthermore, network disturbance impacts the predictability of the MCS slightly more when deploying network sockets for inter-VM communication (AS_1) than hypervisor-managed shared memory (AS_2).

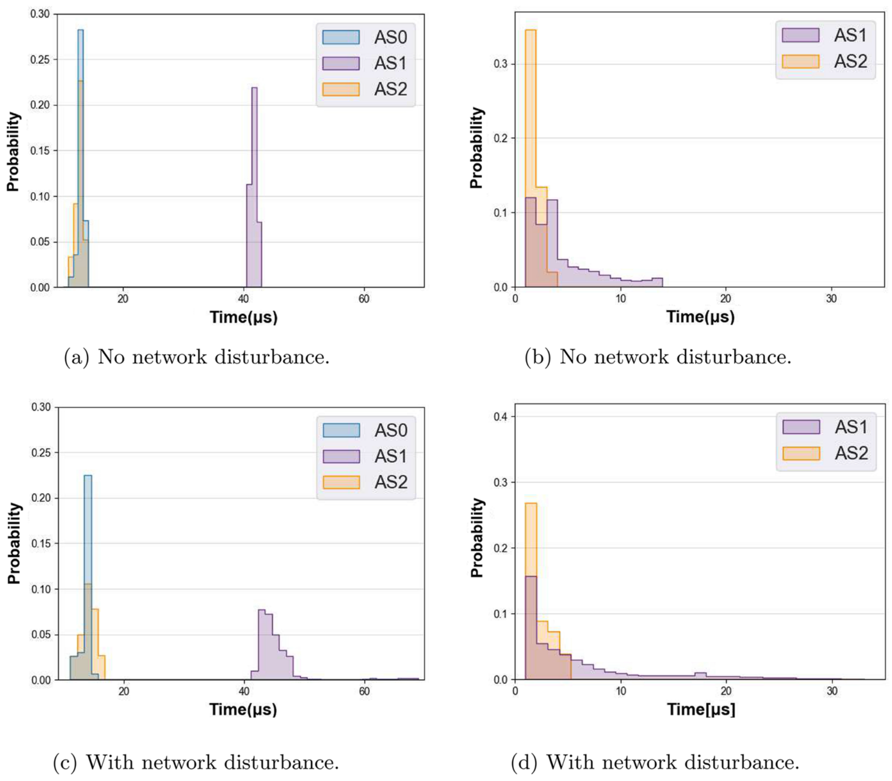

Fig. 7 Density function plots of execution times of tasks *C* **a**, **c** and *DS* **b**, **d**, per scenario

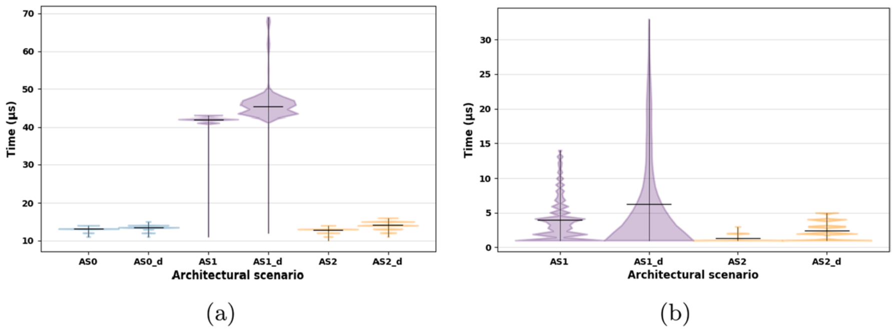

Fig. 8 Violin plots of execution times of tasks *C* **a** and *DS* **b**, per scenario

Firstly, the observed $WCET_C$ and $ACET_C$ in each scenario AS_1 and AS_2 (see Fig. 7a) suggests that hypervisor-based shared memory (AS_2) should lead to execution times that are considerably closer to the non-virtualized baseline (AS_0). In fact, compared to the baseline, shared memory only worsens $WCET_C$ and $ACET_C$

by 7.14 and 0.15% (AS_2), whereas network sockets worsen $WCET_C$ and $ACET_C$ by 207.14 and 220.35% (AS_1).

Secondly, as Table 5 highlights, there are $WCET_C$ shifts when adding network disturbance to all architectural scenarios (see Figs. 7a and c). Specifically, network disturbance make $WCET_C$ worse by 7.14, 60.46, and 33.33% in architectural scenarios AS_0, AS_1 and AS_2, respectively. Similarly, network disturbance makes std_C worse by 1.11, 324.74, and 110.22%.

Furthermore, by directly comparing scenarios AS_1 and AS_2 under network disturbance we can notice an increase in time variability when network sockets are used. In fact, the value of std_C from scenario AS_2 to AS_1 worsens by 122.7%. Clearly, such worsening is due to the interference of tasks when accessing shared network resources.

It is worth noting that, as Figs. 7b and d show, also execution times of *DS*, both with and without network disturbance, show less variability when deployed as a VM on the same board where *C* is running (AS_2) compared to the scenario where these two tasks communicate over the Internet through network sockets (AS_1). Specifically, in case of network disturbance, the value of std_{DS} in AS_1 is 92.37% higher than the one in AS_2.

Although shared memory is the best option with respect to reducing *WCET* and *std* worsening of tasks, virtualization still causes a slight variation in execution times in case of network disturbance, regardless of the deployed communication channel between *C* and *DS*. This is because, despite resource partitioning and deployment of real-time schedulers, there still is interference caused by both context switches between VMs and interrupt handling.

Despite hardware support for virtualization speeds up the process due both to the presence of a hypervisor-dedicated execution mode (e.g. ARM EL3) and a set of additional registers, context switches still lead to non-negligible delay.

Furthermore, each time a device sends an interrupt, as the network device deployed in our experiment does, the hypervisor intercepts it to determine which VM has to serve it. This behavior causes a delay to the running VM. Though modern hardware interrupt managers are extended to support virtualization allowing direct access to VMs when possible (e.g. the ARM GICv4), currently, such support is limited to MSIs and IPIs and does not avoid triggering the hypervisor.

In conclusion, in spite of the problems we have outlined, and given the experimental results, we believe that virtualization is the best choice to implement high-frequency monitoring of control algorithms via shared memory, as non-virtualized communication via network sockets not only impacts the predictability of the system more, it also makes performance worse due to use of the software network stack and the more intensive use of the I/O.

6 Related works

6.1 Model-based deployment

Although the plethora of semi-formal and formal modeling notations opens many opportunities to practitioners for the model-based deployment of complex systems, there is no one-size-fits-all solution in which language should be used, as modeling may require elements whose semantics are not found in any existing notation (De Saqui-Sannes et al. 2022).

Commonly, existing modeling languages allow designers to extend their semantics (e.g., Unified Modeling Language profiles such as SysML and MARTE or extended Petri nets (De Saqui-Sannes et al. 2022)). However, addressing domain-specific requirements drove the development of ad-hoc languages, such as the Architecture Analysis and Design Language (Mkaouar et al. 2020), adopted in avionics, automotive, and robotics domains, or the Dynamic STate machines proposed by Benerecetti et al. (2017) to appropriately model requirements of railways standards.

To the best of our knowledge, there is no deployment model able to describe the integration of real-time applications and application-level fog monitoring in virtualized MPSoCs, leading to our proposal in Sect. 3. It is worth noting our deployment model does not aim to analytically model deterministic and/or non-deterministic interference and time requirements, as these can be described using other well-known models. Rather, our goal is to drive the design and deployment of MCSs on virtualized MPSoCs in a model-based fashion, proposing a flexible notation that facilitates developers in modeling architectural choices.

6.2 Application-level monitoring

Several monitoring strategies with different placement and abstraction levels have been proposed and deployed over desktop and embedded systems. We focus on reviewing application-level software monitors, as our work considers this class when dealing with the case-study in Sect. 5.1.

There are many application-level software monitors based on data-driven techniques that allow anomaly detection in cyber-physical systems based on data collected from sensors distributed in the environment.

Molka-Danielsen et al. (2015) highlight the insightful information about environmental conditions that the large-scale integration of wireless sensor network technologies provides through big data analytics.

Naeem et al. (2020) outline that the usage of image visualization and deep learning models can leverage the big amount of data that sensors collect for malware detection.

Other than using data-driven techniques, there also are other monitors based on source code instrumentation for runtime monitoring of software and/or hardware faults (Kadar et al. 2019; Pike et al. 2013).

Application-level monitoring through pluggable hardware components has also shown its utility in verifying the correctness of the behavior of commercial-off-the-shelf components. Moreover, on-chip integration of LTL checkers has been investigated to address the ISO 26262 standard safety guidelines (Heffernan et al. 2014).

Although effective, these monitors lack evaluation of their predictability, their impact when deployed in a hard real-time context, and their implementation in distributed environments. Indeed, as introduced in Sect. 1, the literature focuses on reaching good accuracy levels for challenging anomaly types and addressing problems poorly linked to predictability issues when deploying monitors that handle application-level data from real-time processes (Costa et al. 2022).

6.3 Monitoring via virtualization

Leveraging hypervisors to monitor the activities of a system is an idea recently investigated by several works in literature. However, these works differ both in terms of the monitor's placement, which could be, e.g., integrated into the hypervisor code or placed on separate VMs, and in terms of their objectives, e.g., power efficiency, security, and fault tolerance.

In Poggi et al. (2018) the Xtratum hypervisor is modified to provide power monitoring services that can obtain information on the power consumption of the board on which it is running to optimize it. Other works focus on security through the development of VMM-level frameworks for malware and intrusion detection (Kumara and Jaidhar 2018; Kwon et al. 2018; Kadar et al. 2019).

Our approach stands out since it focuses on monitoring anomalies of applications running high-frequency real-time processes, with a particular emphasis on the trade-off between the temporal predictability of the monitoring and the temporal intrusion of monitoring on monitored applications.

7 Conclusions and future work

This paper reviewed the state-of-the-art model-based development and fog monitoring of real-time applications, proposed a model-based approach to design, deploy, and evaluate Mixed-Criticality Systems (MCS) on Multiprocessor System-on-Chips (MPSoCs), and evaluated the predictability of several architectural scenarios for the ITER case-study, deploying concurrent application-level monitoring of system behavior in time series collected from real-time control within the ITER case-study.

Considering the evidence of literature gaps in (1) Evaluating the predictability of systems when deploying application-level monitoring of real-time applications through fog computing and data-driven anomaly detection, and (2) Applying model-based principles for the development of MCS on virtualized MPSoCs, this paper proposed the application of model-based development for the design, deployment, and evaluation of MCSs on virtualized MPSoCs.

In light of the aforementioned, a model-based system development process, which leverages a well-defined MCS deployment model, was proposed and applied to develop prototypes of the industry-relevant ITER case-study. This involves plasma Vertical Stabilization (VS) throughout nuclear fusion, whose predictable control and timely monitoring are critical. By means of model-based system development and deployment of the resulting prototypes, we assessed the predictability of different architectural choices when designing application-level monitoring of real-time applications deployed on virtualized MPSoCs.

The results obtained highlighted that the use of hypervisor-managed shared memory on a virtualized MPSoC leads to the least impact on WCET and ACET of tasks with respect to the reference scenario, which involves neither virtualization nor monitoring. In fact, hypervisor-managed shared memory worsens such metrics only by 7.14 and 0.15%, whereas network sockets worsen them by 207.14 and 220.35%.

Our experimentation also assessed the impact of network disturbance on execution times, regardless of the deployed communication channel and virtualization. Remarkably, network disturbance due to general-purpose tasks impacts execution times of tasks in all cases, be it non-virtualized or virtualized deployment, worsening WCET of control up to 60.46% when network sockets are used and up to 33.33% when shared memory and virtualization are used. This last impact is due to several isolation flaws when hypervisors handle interrupts and context switches due to access to shared network resources.

As the experiments have been performed with reference to the industry-relevant ITER case-study, which, as mentioned, involves the safety-critical real-time control of plasma during nuclear fusion through control tasks, our work showed the feasibility of model-based design, deployment, and evaluation of the predictability of MCSs that implement fog monitoring of real-time control.

Future work should: (1) Consider the assessment of system predictability when virtualizing the local environment with a partitioning hypervisor, such as Jailhouse or Bao, which are able to ensure better isolation guarantees between virtual machines; (2) Address the predictability of several anomaly detection techniques, as they often are non-deterministic, and should, therefore, be carefully chosen when applying the system development process we have proposed in this paper; (3) Automatize the proposed system development process to speed up the deployment of architectural scenarios, aiding the developers with tools that process architectural scenarios and automatically configure the corresponding prototypes; and (4) Delve deeper into the analysis of software predictability when taking into account end-to-end delays due to data transactions and its aging across the communication and computation chain, strongly influenced by the choice of communication protocols.

Acknowledgements This work has been carried out within the framework of the EUROfusion Consortium, funded by the European Union via the Euratom Research and Training Programme (Grant Agreement No 101052200 - EUROfusion). The work by L. De Simone was supported by European Union - FSE-REACT-EU, PON Research and Innovation 2014-2020 DM1062/2021 contract number 18-I-15350a-6. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Commission. Neither the European Union nor the European Commission can be held responsible for them.

Funding Open access funding provided by Università degli Studi di Napoli Federico II within the CRUI-CARE Agreement.

Data availability The data that support the findings of this study are available from the authors upon request.

References

AEEC (2010) ARINC-653: Avionics application Software standard interface part 1. Technical report

Agrawal A, Mancuso R, Pellizzoni R, Fohler G (2018) Analysis of dynamic memory bandwidth regulation in multi-core real-time systems. In: 2018 IEEE real-time systems symposium (RTSS), IEEE, pp. 230–241

Alonso S, Lázaro J, Jiménez J, Bidarte U, Muguira L (2021) Evaluating latency in multiprocessing embedded systems for the smart grid. Energies 14(11):3322

Ambrosino G, Ariola M, De Tommasi G, Pironti A (2010) Plasma vertical stabilization in the ITER tokamak via constrained static output feedback. IEEE Trans Control Syst Technol 19(2):376–381

Ariola M, Pironti A (2016) Magnetic control of Tokamak plasmas, 2nd edn. Springer, London

Avizienis A, Laprie J-C, Randell B, Landwehr C (2004) Basic concepts and taxonomy of dependable and secure computing. IEEE Trans Dependable Secure Comput 1:11–33

Avnet I Top 6 Autonomous Vehicle Use Cases You Need to Read (White Paper). https://www.avnet.com/wps/wcm/connect/onesite/5e738e6a-a181-46b0-b49b-941ce36fed98/Xilinx-Automotive-eBook-APAC-Eng.pdf?MOD=AJPERES &CVID=na56dje &attachment=false &id=1591370911158

Avon G, Buscarino A, Neto AC, Sartori F (2021) Marte2 embedded signal processing unit for the iter magnetics diagnostics. In: IECON 2021–47th annual conference of the IEEE industrial electronics society, IEEE, pp. 1–6

Barbalace A, Manduchi G, Neto A, De Tommasi G, Sartori F, Valcarcel DF (2011) Performance comparison of EPICS IOC and MARTe in a hard real-time control application. IEEE Trans Nucl Sci 58:3162–3166

Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A (2003) Xen and the art of virtualization. ACM SIGOPS Oper Syst Rev 37:164–177

Bellavista P, Berrocal J, Corradi A, Das SK, Foschini L, Zanni A (2019) A survey on fog computing for the internet of things. Pervasive Mob Comput 52:71–99

Benerecetti M, De Guglielmo R, Gentile U, Marrone S, Mazzocca N, Nardone R, Peron A, Velardi L, Vittorini V (2017) Dynamic state machines for modelling railway control systems. Sci Comput Program 133:116–153

Biondi A, Buttazzo GC, Bertogna M (2015) Schedulability analysis of hierarchical real-time systems under shared resources. IEEE Trans Comput 65(5):1593–1605

Bittencourt L, Immich R, Sakellariou R, Fonseca N, Madeira E, Curado M, Villas L, DaSilva L, Lee C, Rana O (2018) The internet of things, fog and cloud continuum: integration and challenges. Internet of Things 3:134–155

Bzai J, Alam F, Dhafer A, Bojovic M, Altowaijri SM, Niazi IK, Mehmood R (2022) Machine learning-enabled internet of things (IoT): data, applications, and industry perspective. Electronics 11(17):2676

CENELEC (2011) EN 50128. Technical report

Chandola V, Banerjee A, Kumar V (2009) Anomaly detection: a survey. ACM Comput Surv 41(3):1–58

Chardet M, Coullon H, Pertin D, Perez C (2018) Madeus: a formal deployment model. In: 2018 international conference on high performance computing & simulation (HPCS), pp. 724–731

Cilardo A, Cinque M, De Simone L, Mazzocca N (2022) Virtualization over multiprocessor systems-on-chip: an enabling paradigm for the industrial Internet of Things. Computer 55(10):35–47

Cinque M, Cotroneo D, De Simone L, Rosiello S (2021) Virtualizing mixed-criticality systems: a survey on industrial trends and issues. Future Gener Comput Syst 129:315–330

Commission, I.E (1998) Software requirements. Technical report

Costa B, Bachiega J, Carvalho LR, Rosa M, Araujo A (2022) Monitoring fog computing: a review, taxonomy and open challenges. Comput Netw 215:109–189

Cotroneo D, De Simone L, Natella R (2021) Timing covert channel analysis of the vxworks mils embedded hypervisor under the common criteria security certification. Comput Secur 106:1–13

Coulouris GF, Dollimore J, Kindberg T (2011) Distributed systems: concepts and design, 5th edn. Pearson, Boston

De Saqui-Sannes P, Vingerhoeds RA, Garion C, Thirioux X (2022) A taxonomy of MBSE approaches by languages, tools and methods. IEEE Access 10:120936–120950

De Tommasi G (2022) System-Engineering approach for the ITER PCS design: the correction coils current controller case study. Fusion Eng Des 185:1–6

De Tommasi G, Maviglia F, Neto A, Lomas P, McCullen P, Rimini FG (2014) Plasma position and current control system enhancements for the JET ITER-like wall. Fusion Eng Des 89:233–242

Delgado N, Gates AQ, Roach S (2004) A taxonomy and catalog of runtime software-fault monitoring tools. IEEE Trans Software Eng 30(12):859–872

Ding K, Ding S, Morozov A, Fabarisov T, Janschek K (2019) On-line error detection and mitigation for time-series data of cyber-physical systems using deep learning based methods. In: 2019 15th European Dependable Computing Conference (EDCC), pp. 7–14

Dong X, Jin B, Tang B, Tang H (2018) On real-time monitoring on data stream for traffic flow anomalies. In: 2018 IEEE International conference on parallel & distributed processing with applications, ubiquitous computing & communications, big data & cloud computing, social computing & networking, sustainable computing & communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom), pp. 322–329

Dubbioso S (2022) Vertical stabilization of tokamak plasmas via extremum seeking. IFAC J Syst Control 21:100203

EUROfusion (2018) European Research Roadmap to the Realisation of Fusion Energy. https://www.euro-fusion.org/fileadmin/user_upload/EUROfusion/Documents/2018_Research_roadmap_long_version_01.pdf, Accessed November 2022

Goodloe A, Pike L (2010) Monitoring distributed real-time systems: a survey and future directions. Technical report, NASA Langley Research Center

Hassan M, Pellizzoni R (2020) Analysis of memory-contention in heterogeneous cots mpsocs. In: 32nd Euromicro conference on real-time Systems (ECRTS 2020). Schloss Dagstuhl-Leibniz-Zentrum für Informatik

Heffernan D, MacNamee C, Fogarty P (2014) Runtime verification monitoring for automotive embedded systems using the ISO 26262 functional safety standard as a guide for the definition of the monitored properties. IET Softw 8:193–203

Hemmer A, Abderrahim M, Badonnel R, François J, Chrisment I (2021) Comparative Assessment of Process Mining for Supporting IoT Predictive Security. IEEE Trans Netw Serv Manage 18:1092–1103

Houdek P, Sojka M, Hanzálek Z (2017) Towards predictable execution model on arm-based heterogeneous platforms. In: 2017 IEEE 26th international symposium on industrial electronics (ISIE), IEEE, pp. 1297–1302

Hughes A, Awad A (2019) Quantifying performance determinism in virtualized mixed-criticality systems. In: 2019 IEEE 22nd international symposium on real-time distributed computing (ISORC), pp. 181–184

ISO (2011) Product development: software Level. Technical report

Kadar M, Tverdyshev S, Fohler G (2019) System calls instrumentation for intrusion detection in embedded mixed-criticality systems. In: 4th international workshop on security and dependability of critical embedded real-time systems (CERTS 2019)

Kao CH (2020) Survey on evaluation of IoT services leveraging virtualization technology. In: Proceedings 2020 5th international conference on cloud computing and Internet of Things, pp. 26–34

Kivity A, Kamay Y, Laor D, Lublin U, Liguori A (2007) kvm: the Linux virtual machine monitor. In: Proceedings of the Linux symposium, vol. 1, pp. 225–230

Klein G, Elphinstone K, Heiser G, Andronick J, Cock D, Derrin P, Elkaduwe D, Engelhardt K, Kolanski R, Norrish M (2009) seL4: Formal verification of an OS kernel. In: Symposium on operating systems principles, pp. 207–220

Kloda T, Solieri M, Mancuso R, Capodieci N, Valente P, Bertogna M (2019) Deterministic memory hierarchy and virtualization for modern multi-core embedded systems. In: 2019 IEEE real-time and embedded technology and applications symposium (RTAS), IEEE, pp. 1–14

Kshemkalyani AD, Singhal M (2011) Distributed computing: principles, algorithms, and systems. Cambridge University Press, Cambridge

Kumara A, Jaidhar C (2018) Automated multi-level malware detection system based on reconstructed semantic view of executables using machine learning techniques at VMM. Future Gener Comput Syst 79:431–446

Kwon D, Oh K, Park J, Yang S, Cho Y, Kang BB, Paek Y (2018) Hypernel: a hardware-assisted framework for kernel protection without nested paging. In: Proceedings of the 55th annual design automation conference, pp. 1–6

Lee J, Xi S, Chen S, Phan LT, Gill C, Lee I, Lu C, Sokolsky O (2012) Realizing compositional scheduling through virtualization. In: 2012 IEEE 18th real time and embedded technology and applications symposium, IEEE, pp. 13–22

Maiza C, Rihani H, Rivas JM, Goossens J, Altmeyer S, Davis RI (2019) A survey of timing verification techniques for multi-core real-time systems. ACM Comput Surv (CSUR) 52(3):1–38

Mkaouar H, Zalila B, Hugues J, Jmaiel M (2020) A formal approach to AADL model-based software engineering. Int J Softw Tools Technol Transf 22:219–247

Modica P, Biondi A, Buttazzo G, Patel A (2018) Supporting temporal and spatial isolation in a hypervisor for arm multicore platforms. In: 2018 IEEE international conference on industrial technology (ICIT), IEEE, pp. 1651–1657

Molka-Danielsen J, Engelseth P, Wang H (2015) Large scale integration of wireless sensor network technologies for air quality monitoring at a logistics shipping base. J Ind Inf Integr 10:20–28

Murari A (2018) Adaptive predictors based on probabilistic SVM for real time disruption mitigation on JET. Nucl Fusion 58(5):056002

Naeem H, Ullah F, Naeem MR, Khalid S, Vasan D, Jabbar S, Saeed S (2020) Malware detection in industrial internet of things based on hybrid image visualization and deep learning model. Ad Hoc Netw 105:1–12

Neto AC (2011) A survey of recent MARTe based systems. IEEE Trans Nucl Sci 58:1482–1489

Pike L, Wegmann N, Niller S, Goodloe A (2013) Copilot: monitoring embedded systems. Innov Syst Softw Eng 9:235–255

Pinto S, Santos N (2019) Demystifying arm trustzone: a comprehensive survey. Comput Surv 51:1–36

Pivoto DGS, de Almeida LFF, da Rosa Righi R, Rodrigues JJPC, Lugli AB, Alberti AM (2021) Cyber-physical systems architectures for industrial internet of things applications in industry 4.0: a literature review. J Manufact Syst 58:176–192

Poggi T, Onaindia P, Azkarate-askatsua M, Grüttner K, Fakih M, Peiró S, Balbastre P (2018) A hypervisor architecture for low-power real-time embedded systems. In: 2018 21st Euromicro conference on digital system design (DSD), pp. 252–259

Popek GJ, Goldberg RP (1974) Formal requirements for virtualizable third generation architectures. Commun ACM 17:412–421

Puliafito C, Mingozzi E, Longo F, Puliafito A, Rana O (2019) Fog computing for the internet of things: a survey. ACM Trans Internet Technol 19(2):1–41

Quamara M, Pedroza G, Hamid B (2021) Multi-layered model-based design approach towards system safety and security co-engineering. In: 2021 ACM/IEEE international conference on model driven engineering languages and systems companion (MODELS-C), pp. 274–283

Raupp G (2014) Event generation and simulation of exception handling with the ITER PCSSP. Fusion Eng Des 89:523–528

RTCA (2012) DO-178C - Software Considerations in Airborne Systems and Equipment Certification. Technical report

Sánchez JMG, Jörgensen N, Törngren M, Inam R, Berezovskyi A, Feng L, Fersman E, Ramli MR, Tan K (2022) Edge computing for cyber-physical systems: a systematic mapping study emphasizing trustworthiness. ACM Trans Cyber-Phys Syst 6(3):1–28

Siemens AG (2022) Jailhouse. https://github.com/siemens/jailhouse

Singh P, Saman Azari M, Vitale F, Flammini F, Mazzocca N, Caporuscio M, Thornadtsson J (2022) Using log analytics and process mining to enable self-healing in the Internet of Things. Environ Syst Decis 42:1–17

Snipes J (2021) ITER plasma control system final design and preparation for first plasma. Nucl Fusion 61:1–9

Sohal P, Tabish R, Drepper U, Mancuso R (2022) Profile-driven memory bandwidth management for accelerators and CPUs in QoS-enabled platforms. Real-Time Syst 58(3):235–274

Sommerville I (2016) Software Engineering, 10th edn. Pearson Education Limited, Boston

Stabellini S (2014) Xen ARM with virtualization extensions white paper

Steinberg U, Kauer B (2010) NOVA: a microhypervisor-based secure virtualization architecture. In: Proceedings of the 5th European conference on computer systems, pp. 209–222

Taherizadeh S, Jones AC, Taylor I, Zhao Z, Stankovski V (2018) Monitoring self-adaptive applications within edge computing frameworks: a state-of-the-art review. J Syst Softw 136:19–38

The Linux Foundation: ARINC 653 Scheduler - Xen (2015). https://wiki.xenproject.org/wiki/ARINC653_Scheduler

Ungurean I, Gaitan NC (2021) Software architecture of a fog computing node for industrial internet of things. Sensors 21(11):3715

Valcárcel DF (2014) The JET real-time plasma-wall load monitoring system. Fusion Eng Design 89:243–258

Vega J et al (2022) Disruption prediction with artificial intelligence techniques in tokamak plasmas. Nat Phys 18(7):741–750

Verma G, Gupta Y, Malik AM, Chapman B (2021) Performance evaluation of deep learning compilers for edge inference. In: 2021 IEEE international parallel and distributed processing symposium workshops (IPDPSW), pp. 858–865

Walker M (2019) Assessment of controllers and scenario control performance for ITER first plasma. Fusion Eng Des 146:1853–1857

Wang Z, Sun D, Xue G, Qian S, Li G, Li M (2019) Ada-things: an adaptive virtual machine monitoring and migration strategy for internet of things applications. J Parallel Distrib Comput 132:164–176

Watterson C, Heffernan D (2007) Runtime verification and monitoring of embedded systems. IET Software 1:172–179

Wiki.Xenproject (2019) Xen Wiki–RTDS-Based-Scheduler. https://wiki.xenproject.org/wiki/RTDS-Based-Scheduler

Yao G, Yun H, Wu ZP, Pellizzoni R, Caccamo M, Sha L (2015) Schedulability analysis for memory bandwidth regulated multicore real-time systems. IEEE Trans Comput 65(2):601–614

Zhao Q, Gu Z, Zeng H, Zheng N (2018) Schedulability analysis and stack size minimization with preemption thresholds and mixed-criticality scheduling. J Syst Archit 83:57–74

Marcello Cinque is an Associate Professor at the Department of Computer and Systems Engineering, Federico II University of Naples, Italy. He received the Graduate (Hons.) degree and Ph.D. degree from the Federico II University of Naples, Italy, in 2003 and 2006, respectively. His interests include field failure data analysis of distributed systems, and virtualization solutions for real- time mixed-criticality systems. He is the Chair and/or TPC member of several techni- cal conferences and workshops on dependable systems, including IEEE Dependable Systems and Networks and ACM International Conference Proceeding Series.

Luigi De Simone is an Assistant Professor at the University of Naples Federico II, Italy. His research interests include dependability benchmarking, fault injection testing, virtualization reliability and its application on safety- and secure- critical systems. He contributed, as author and reviewer, to several top journals and conferences on dependable computing and software engineering, and he has been organizing multiple editions of the international workshop on software certification (WoSoCer) within the IEEE ISSRE conference.

Nicola Mazzocca is a Full Professor of Computer Systems at the Department of Electrical Engineering and Information Technology of the University of Naples Federico II. Since 1994, he has held numerous university courses and has been involved in several professional training activities on different topics, including high-performance systems, distributed and embedded systems, security, and reliability. His research activities concern computer architecture, distributed systems, high-performance computing, and safety-critical applications. He is author of more than 250 publications in international journals, books, and conference proceedings.

Daniele Ottaviano is a PhD student enrolled since 2021 in the Fusion Science and Engineering (FSE) Doctoral Programme at the University of Padova - University of Naples Federico II. His research activities focus on virtualization technologies for real-time and embedded systems. He is especially interested in hypervisor-level resource management to realize dependable mixed-criticality systems over next-generation Multiprocessor Systems-on-Chip for control systems in fusion reactors.

Francesco Vitale is a PhD student in Information Technology and Electrical Engineering at the Department of Electrical Engineering and Information Technology of the University of Naples Federico II, where he has earned his M.Sc. degree in Computer Engineering in 2021. He has been a Visiting researcher at the Chair of Process and Data Science (PADS) - RWTH Aachen University in 2023. His research activities mainly address Industry 4.0, Anomaly Detection, and Process Mining.

Authors and Affiliations

Marcello Cinque[1] · Luigi De Simone[1] · Nicola Mazzocca[1] · Daniele Ottaviano[1] · Francesco Vitale[1]

✉ Francesco Vitale
francesco.vitale@unina.it

[1] DIETI, Università degli Studi di Napoli Federico II, Via Claudio, 21, 80125 Naples, Italy