DOL-BIP-Critical: a tool chain for rigorous design and implementation of mixed-criticality multi-core systems

Georgia Giannopoulou, Peter Poplavko, Dario Socci, Pengcheng Huang, Nikolay Stoimenov, Paraskevas Bourgos, Lothar Thiele, Marius Bozga, Saddek Bensalem, Sylvain Girbal, et al.

Noname manuscript No.
(will be inserted by the editor)

DOL-BIP-Critical: A Tool Chain for Rigorous Design and Implementation of Mixed-Criticality Multi-Core Systems

Georgia Giannopoulou · Peter Poplavko · Dario Socci · Pengcheng Huang · Nikolay Stoimenov · Paraskevas Bourgos · Lothar Thiele · Marius Bozga · Saddek Bensalem · Sylvain Girbal · Madeleine Faugere · Romain Soulat · Benoît Dupont de Dinechin

Received: date / Accepted: date

Abstract Mixed-criticality systems are promoted in industry due to their potential to reduce size, weight, power, and cost. Nonetheless, deploying mixed-criticality applications on commercial multi-core platforms remains a highly challenging problem. To name a few reasons: (i) Industrial mixed-criticality applications are usually complex reactive applications, which cannot be specified by traditional, e.g., dataflow-based, models of computation. Appropriate mixed-criticality models of computation built upon Vestal's assumptions are missing; (ii) Scheduling such applications on multicores with shared resources, such as memory buses, requires that any timing interference among applications of different criticality is bounded in order to guarantee - the necessary for certification - temporal isolation and to enable incremental design; (iii) The implementation of isolation-preserving mixed-criticality schedulers is itself subject to certification. Hence, it needs to be not only efficient, but also provably correct. This paper proposes, for the first time, a complete design flow covering all aspects from specification, using a novel mixed-criticality aware model of computation (DOL-Critical), to correct-by-construction implementation, using the principle 'what you verify is what you generate' which is based on a novel variant of task automata (BIP). We demonstrate the applicability of our design flow with an industrial avionic test case on the state-of-the-art Kalray MPPA®-256.

Keywords real-time systems · mixed-criticality systems · multi-core scheduling · rigorous design · software synthesis · avionics

Georgia Giannopoulou · Pengcheng Huang · Nikolay Stoimenov · Lothar Thiele
Computer Engineering and Communication Networks Laboratory, ETH Zurich, 8092 Zurich, Switzerland
E-mail: {firstname.lastname}@tik.ee.ethz.ch

Peter Poplavko · Dario Socci · Paraskevas Bourgos · Marius Bozga · Saddek Bensalem
Univ. Grenoble-Alpes, CNRS, VERIMAG, Grenoble, F-38000, France
E-mail: {firstname.lastname}@univ-grenoble-alpes.fr

Sylvain Girbal · Madeleine Faugere · Romain Soulat
THALES Research and Technology, PALAISEAU Cedex, F-91767, France
E-mail: {firstname.lastname}@thalesgroup.com

Benoît Dupont de Dinechin
Kalray S.A., F38330 Montbonnot Saint Martin, France
E-mail: benoit.dinechin@kalray.eu

1 Introduction

With the proliferation of multi- and many-core platforms in the electronics market, the embedded system industry is experiencing an unprecedented trend towards integrating multiple applications into a common platform. The migration from single-core to multi-core designs affects even safety-critical domains, such as avionics and automotive. In such domains, applications are characterized by discrete safety criticality levels, as defined e.g., by the DO-178C avionics standard [17]. Integration of applications with different safety criticality has led to the design of so-called *mixed-criticality systems*, which has been a prominent research topic in recent years [12]. Nonetheless, a complete and sound methodology for successfully integrating mixed-criticality applications on (shared-memory) *multicores* remains by and large an open problem. Some of the challenges are listed below.

Specification. Firstly, the specification of mixed-criticality (MC) applications does not usually fit into traditional streaming models of computation, such as Kahn process networks [35], for which established multi-core scheduling methods exist [57]. MC applications are often reactive control applications, where task activation depends on a combination of data availability (similar to streaming applications), complex (non-periodic) arrival patterns, and dynamic decisions by schedulers which can skip tasks or activate them in degraded mode. As a result, the MC scheduling models widely used in the literature, like Vestal's [64], miss any link to application-level specifications, which calls for new models of computation for the precise representation of real-world MC applications.

Temporal Isolation. Secondly, mixed-criticality design needs to ensure temporal isolation for certification purposes. Namely, applications of different safety criticality levels should not interfere (delay each other), or their interference must be bounded according to safety standards. To achieve isolation on a single core, system designers usually rely on time partitioning mechanisms at platform level, such as the ones specified by the ARINC-653 standard [7]. In contrast to partitioning, in research literature it is commonly assumed that the isolation property is ensured in a non-symmetric way, for efficiency. That is, the interference from lower to higher criticality tasks is eliminated or bounded, but the interference from higher to lower tasks is tolerated. The established MC scheduling model of Vestal [64] represents tasks with multiple worst-case execution time (WCET) bounds at different safety criticality levels. The bounds become more conservative and more probable as the criticality level increases. Most scheduling policies based on this model execute all tasks initially according to their least conservative WCET bounds, and can change the schedule dynamically at runtime if high criticality tasks require more resources (execution time). After the schedule switch, lower criticality tasks may receive less or no service. Inhibiting those tasks prevents unwanted interference to high criticality tasks and improves resource efficiency. This way, non-symmetric isolation is ensured on single cores. However, on multicores one has to consider possible interferences among tasks with different criticality on additional (non-computational) shared platform resources, e.g., shared caches or memory buses. Preserving isolation in the presence of shared resources is not trivial [39]. It requires new industrial specifications, like [7], and an extension of Vestal's original MC model to account for the accessing behavior to shared resources. Existing multi-core scheduling solutions often neglect this source of interference or assume that it has a bounded effect on the individual tasks' execution times [44,46,9,40,16,53,37]. On the contrary, we identify a state-of-the-art approach that preserves temporal isolation [25], and we offer a new rigorous and flexible implementation methodology for it.

Incremental Design. Thirdly, due to the high cost of certification, industry poses the requirement for incremental design of MC systems [8]. A MC scheduling policy should support adding new applications to a system without any impact on the schedule or the real-time properties of higher criticality applications that already existed in the system design. This removes the need for re-certification every time a new application is integrated, thus reducing the overall cost. Industrial standards, such as [7], specify mechanisms for incremental design that are restricted to single cores and symmetric isolation. New incremental design methodologies have to take into consideration non-symmetric isolation and interference of shared resources on multicores. This requirement has received, nonetheless, minimal attention in literature. The implementation methodology proposed in this paper targets at incremental design.

Implementation. Fourthly, the implementation of both MC applications and their supporting mechanisms, such as schedulers and mechanisms for temporal isolation, is itself subject to certification. Given that such mechanisms can include inter-core synchronisation, distributed monitoring of task execution times, dynamic schedule reconfigurations, resource servers, a manual implementation can be challenging and error-prone. Additionally, the runtime overhead of the supporting mechanisms is non-negligible and must be considered at design time for a safe deployment [54]. These challenges call for rigorous approaches for the implementation and validation of MC schedulers and the correct-by-construction MC software synthesis. Implementation paradigms for timing-critical multi-core applications, such as [28], show promising results. However, even though they are rigorous, they are not flexible, *i.e.,* they are restricted to a particular model of computation and hardware architecture.

In this paper we present a complete design flow for mixed-criticality multi-core systems, which addresses all aforementioned challenges. The main contributions can be summarized as follows:

- We apply Vestal's model for MC task sets [64] so as to account, besides WCET, also for shared-resource accesses at different criticality levels, for degraded mode of low-criticality tasks, and for incremental design.
- We extend Vestal's model further to a complete model of computation, with inter-task dependencies and communication requirements. This model is expressed in an architecture description language (ADL), DOL-Critical, which enables the specification of MC applications and schedules complying with the above extensions. This way we demonstrate the new elements that can be potentially included in popular ADLs, such as AADL, to account for mixed-criticality and multi-core designs.
- We present an optimization tool for isolation-preserving multi-core scheduling of MC applications which are specified in DOL-Critical. The optimization tool is integrated with response time analysis that considers task interference on shared resources, and it aims at incremental design. Thus, we propose a method that can handle our Vestal's model extensions in practice.
- For rigorous system design, we extend the timed-automata language BIP [2] to support asynchronous transitions, thus obtaining an enhanced variant of task automata [5, 22]. As a result, we extend the scope of automata as design languages from synchronous to general real-time systems. Traditionally used only for verifying these systems, the automata can now be used to directly express multi-core applications and custom scheduling policies, which leads to the concept *'what you verify is what you generate'* (WYVIWYG). We demonstrate this concept by compiling the DOL-Critical applications and schedules into BIP automata and then performing functional validation and code generation.

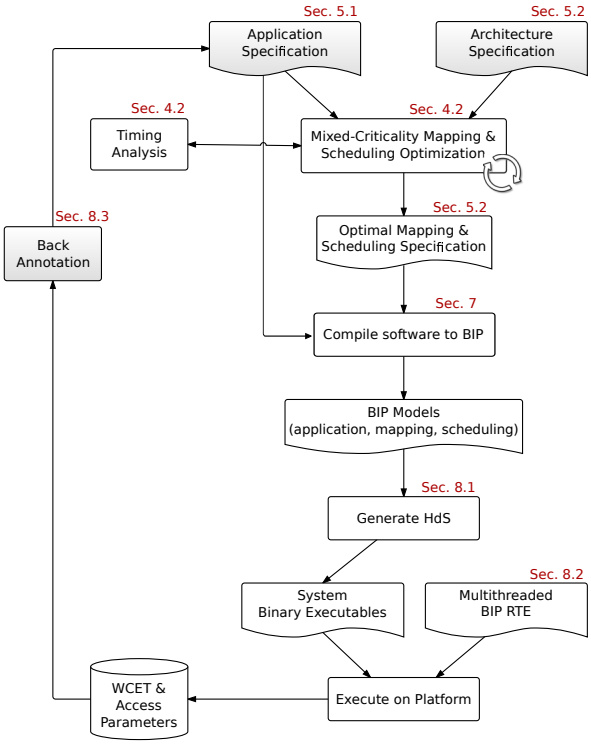


Fig. 1: DOL-BIP-Critical design flow

- We implement a generator from BIP to hardware-dependent software (HdS). The synthesized code preserves the automata semantics up to a bounded clock drift caused by runtime overhead, *e.g.,* thread synchronization. Although in a custom implementation the overhead may potentially be smaller, an automata-based implementation makes the overhead amenable for systematic formal analysis due to the formal automata semantics.
- We integrate all tools, from application specification in DOL-Critical to system modeling in BIP to code generation, into a single tool-chain.
- We show how to integrate runtime overheads characterized after the deployment of the MC application on the target platform back into the optimization tool, reusing its facility to model the shared resources. For this, we introduce a feedback loop in our flow.
- We demonstrate the applicability and utility of our design flow with an avionic test case targeting the Kalray MPPA®-256 platform.

To the best of our knowledge, this is the first seamlessly integrated tool-chain for the specification, scheduling optimization, timing analysis, and correct-by-construction implementation of MC applications on commercial-off-the-shelf multi-core platforms. Note that the model of computation and the respective ADL, the enhanced task automata, the compilation of MC system specifications from ADL into automata for subsequent code generation, and the formal runtime overhead model that is integrated into schedule optimization are presented for the first time in this paper.

Flow overview. The combined DOL-BIP-Critical design flow, which follows the established Y-chart approach [36], is illustrated in Fig. 1. The document shapes represent data (specifica-

tions of application, architecture, mapping in DOL-Critical, BIP models, executable code) and the rectangular shapes represent tools, respectively. The highlighted parts of the flow are user-defined. Namely, the MC application and the target architecture are specified by the system designer. All other steps of the design flow are executed automatically, except for the back annotation of the application specification, which is performed by the system designer after the execution of the MC application on the architecture. The front- and the back-end of the tool-chain are publicly available under [18,49], respectively.

Outline. In the remainder of the paper, Sec. 2 discusses related work. Sec. 3 presents the extensions to Vestal's MC model for resource-sharing multicores and defines the requirements for MC schedulability. Sec. 4 describes a scheduling policy that explicitly considers the effects of resource sharing and ensures temporal isolation, along with an approach for optimizing MC scheduling w.r.t. incremental design. Sec. 5 starts the description of the tool-chain of Fig. 1 by presenting the DOL-Critical language for specifying applications, architectures and schedules. Sec. 6 presents the enhanced task automata language BIP. Sec. 7 and 8 discuss the compilation of an MC application and its optimized schedule into BIP and the deployment of the BIP system representation on the target platform, along with the feedback loop from execution to timing analysis (scheduling optimization). Sec. 9 demonstrates the developed design flow with an avionic test case and Sec. 10 concludes the article.

2 Related Work

Mixed-Criticality Scheduling Models. Scheduling of mixed-criticality (MC) systems has received increasing attention since the original work [64], which introduced the currently dominating model. This model represents MC tasks as periodic (sporadic) real-time tasks with multiple worst-case execution times (WCET), defined at different safety criticality levels. Vestal's model has been applied and extended in several works, [41,21,9,46,20,34] to name a few. For an up-to-date compilation and review of the model extensions and relevant scheduling policies, the interested readers are referred to [12]. In this work, we apply Vestal's model extensions to (i) capture shared-resource accesses, besides WCET, at different criticality levels, (ii) define the degraded mode of lower criticality tasks, and (iii) ensure incremental design.

Temporal Isolation. Although several policies have been suggested for single-core MC systems, fewer solutions exist currently for multicores. One of the main challenges in multicores is satisfying the requirement for *temporal isolation* (or *freedom from interference*), which is dictated by industrial certification standards [17,1]. Since multicores typically feature different types of shared hardware resources, MC scheduling has to explicitly eliminate or bound potential timing interferences on all shared resources. For this purpose, several works advocate the static scheduling or per-core budget assignment on memory buses [59, 70,23], the implementation of novel criticality-aware memory controllers [45,27,29], the privatization of memory banks by cores running single-criticality applications [51,67,69], or the use of virtualization and monitoring mechanisms for isolation among flows of different criticality on a network-on-chip [62]. Such methods allow bounding the effect of resource sharing on the response time of high-criticality applications. However, most of them lack flexibility (e.g., static time-triggered bus scheduling) and/or need special hardware support which limits their applicability to commercial-off-the-shelf platforms.

System-level solutions that target at global temporal isolation via scheduling have been also proposed recently. Anderson *et al.* proposed scheduling MC systems by employing different strategies (partitioned EDF, global EDF, cyclic executive) for different criticality

levels and utilizing a bandwidth reservation server for isolation [6,44]. This work considers mainly the CPU cores as shared resources, but no other platform resources where mixed-criticality applications can interfere. To overcome this limitation, the authors of [25,13] propose scheduling MC applications such that only tasks of the same criticality can be executed, and hence interfere on shared platform resources, at any time. Huang *et al.* formalise this notion under the term Isolation Scheduling and provide optimality results in [33]. In this paper, we employ policies for Isolation Scheduling of MC systems in order to facilitate their deployment on commercial-off-the-shelf platforms without dedicated hardware support. Particularly, we adopt the flexible time-triggered scheduling policy of [25] because (i) it complies with the MC model of Sec. 3, (ii) its dynamic runtime behavior allows efficient resource utilization (Sec. 4), (iii) it enables incremental design, and (iv) timing analysis methods which explicitly consider the effects of timing interference on shared resources are available [26].

Implementation of Mixed-Criticality Systems. The current industrial practice for implementing MC systems on single-core platforms enforces temporal isolation by means of operating system and hardware-level *partitioning* mechanisms, e.g., as specified in the ARINC-653 standard [7]. No existing standards, however, define how isolation is preserved on resource-sharing multicores. Hence to the best of our knowledge, commercial multicores are not used currently for MC deployments in large-scale industrial applications. This highlights the vast need for tools and methodologies for the implementation of multi-core MC systems.

In research, implementation aspects of MC scheduling have started being addressed recently. Herman *et al.* consider the implementation and runtime overhead of multicore MC scheduling in [30], where the scheduling method of [6,44] is implemented in the real-time operating system LITMUS [14]. This policy does not preserve isolation in the presence of shared platform resources. Huang *et al.* develop a framework, where several single-core MC policies are implemented on top of a standard Linux kernel, and their runtime overheads are evaluated on an Intel Core i7 platform [31]. Sigrist *et al.* compare alternative implementations of common multi-core MC mechanisms on top of Linux, and evaluate their overheads on an 4-core Intel Core i5 and a 60-core Xeon Phi [54]. Among others, they consider the overheads of the flexible time-triggered scheduling policy of [25], which is considered in our paper, and show that the implementation overheads can have a tremendous effect on schedulability, hence cannot be neglected. This shows clearly the challenge of implementing multi-core MC systems; rigorous methods are necessary for their scheduling, software synthesis, and timing analysis. This paper achieves a major step in this direction by presenting the first complete design flow for the implementation of isolation-preserving MC systems on commercial multi-core platforms, with explicit consideration of runtime overheads.

Rigorous Design Methods. Rigorous design of timing-critical systems should employ models which possess formal operational semantics and capture the notion of physical time [65]. A relevant class of such models are timed automata, i.e., finite automata with continuous-time clock variables [4]. A literature overview [65] on applying timed automata in real-time systems reveals a large number of tools and a solid mathematical basis. An important extension of the timed automata are *timed automata with tasks*, also known as *task automata* [22]. These models can express and measure the time segments of their execution during which tasks are running. Timed and particularly task automata have many applications in timing analysis and code synthesis, an important example being the task-automata analysis and implementation tool TIMES [5].

Still, timed/task automata alone cannot satisfy all modeling needs, for two reasons. Firstly, they are often not convenient for programmers. Therefore, compilation from high-

level languages, such as UML, to timed automata becomes a common practice, see e.g.,[68]. Secondly, large timed automata suffer from analysis scalability issues. Therefore, for timing-critical system design it may be favorable to employ less expressive, yet better scalable models. Examples are (i) the AADL-based design flow TASTE [48], which employs tools for classical schedulability analysis, and (ii) the design flow CompSoC [28], which employs formal throughput analysis of dataflow graphs.

In this work, we introduce DOL-Critical as a high-level description language and a model of computation for specifying MC applications and multi-core scheduling solutions. The DOL-Critical specifications are fully automatically compiled to an enhanced variant of the BIP language for timed automata [2]. Our rationale for compilation to automata is to reuse their known ability to formally express runtime resource management mechanisms, especially in mixed-criticality settings [55], and to obtain a rigorous methodology for analyzing the runtime overheads. We perform code synthesis for both the application and runtime scheduling directly from the BIP task automata model. To enhance the scalability of timing analysis, we currently rely on a customized high-level analyzer which verifies the system both prior to and after (via a feedback loop) the compilation into BIP automata. We expect that the formal DOL-BIP relation established at compilation can be used to construct, in future work, a formal proof that the analysis can safely bound the runtime overheads.

DOL-Critical is based upon the Distributed Operation Layer (DOL) [60, 32]. A compilation framework from the original DOL to untimed automata in BIP was introduced in [10]. Unlike [10], in our tool-chain, the compilation target automata are timed. Moreover, we enhance the automata to represent real-time tasks and scheduling policies (including MC) explicitly, in a way that they form a homogeneous monolithic system with formal timing-aware semantics that can be validated and synthesized as HdS code for a target platform. We refer to this facility as *what you verify is what you generate* (WYVIWYG). This has led to an essential redefinition of the synergy between DOL and BIP in particular and between ADL and formal-semantics models in general.

3 System Model

This section defines the abstract application and architecture models[1] that are considered in our work as well as the necessary conditions for mixed-criticality schedulability. The application model is based on established assumptions from literature, which are extended to support resource sharing, degraded mode, dependencies, and non-blocking communication, while the architecture model is inspired by commercial many-core architectures. The schedulability conditions represent state-of-the-art methods of capturing temporal isolation and incremental design.

3.1 Mixed-Criticality Application Model

We consider mixed-criticality task sets $\tau = \{\tau_1, \ldots, \tau_n\}$ with criticality levels between 1 (the lowest) and L (the highest). The tasks can be periodic or sporadic. A *periodic* task is characterized by a 4-tuple $\tau_i = \{W_i, \chi_i, \mathbf{C}_i, C_{i,deg}\}$, where:

- $W_i \in \mathbb{N}^+$ is the task's period.
- $\chi_i \in \{1, \ldots, L\}$ is the task's criticality level.

[1] These models are used in our tool-chain for timing analysis (Sec. 4.2). The concrete class of applications and targets architectures that can be specified in DOL-Critical is described in Sec. 5.

Task τ_i	Criticality Level χ_i	Type	Period W_i [ms]	Level-1 WCET $e_i^{max}(1)$[ms]	Level-2 WCET $e_i^{max}(2)$[ms]	RTE Access Count $\mu_i^{max}(1), \mu_i^{max}(2)$
Filter	1	periodic	50	32	2	3
SensorInput	2	periodic	100	1	26	3
GPSConfig	2	sporadic	100	1	21	4
HighFreqBCP	2	periodic	100	1	11	3
LowFreqBCP	2	periodic	100	1	11	3
MagnDeclin	2	periodic	100	1	11	3
Performance	2	periodic	100	1	11	3
Z1	2	periodic	100	1	26	3
Z2	2	periodic	100	1	26	3
Cycle_Begin	2	periodic	100	0	0	10
Frame_Begin	2	periodic	50	0	0	4
Subframe_Bar	1	periodic	50	0	0	2

Table 1: System model model example: FMS application.

– $\mathbf{C}_i$ is a size-L vector of execution profiles, where $C_i(\ell) = (e_i^{min}(\ell), e_i^{max}(\ell), \mu_i^{min}(\ell), \mu_i^{max}(\ell))$ represents a lower and an upper bound on the execution time (e_i) and the number of shared resource accesses (μ_i) of τ_i at level $\ell \leq \chi_i$. Note that execution time e_i denotes the computation or CPU time of τ_i, *without* considering the time spent on accessing shared resources. Such decoupling of the execution and communication time is feasible on fully timing compositional platforms [66].
– $C_{i,deg}$ is a special execution profile that can be employed at runtime if a task τ_j ($\chi_j > 1$) consumes more resources than $C_j(\ell')$ for some ℓ' in $\{1, \ldots, \chi_j - 1\}$. In Vestal's model, in this case it is legal to drop all subsequent jobs of tasks τ_i with $\chi_i \leq \ell'$ in order to free resources for the more critical task τ_j. In this work, for compliance with industrial standards, we do not drop tasks, but instead execute them in *degraded mode*, which is characterized by profile $C_{i,deg}$. This corresponds to the minimum required functionality of τ_i so that no catastrophic effect occurs in the system. If execution of τ_i can be aborted without catastrophic effects, then $C_{i,deg} = (0, 0, 0, 0)$.

A *sporadic* task is characterized by a 5-tuple $\tau_i = \{a_i, I_i, \chi_i, \mathbf{C}_i, C_{i,deg}\}$, with the new parameters ($a_i \in \mathbb{N}^+, I_i \in \mathbb{N}^+$) denoting the maximum allowed number of task activations, a_i, within any time interval I_i[2]. For scheduling purposes, a sporadic task is over-approximated by a periodic "server" task that has a sufficiently high execution frequency and tighter deadline to meet the deadlines of the sporadic task that it represents, see *e.g.,* [50].

Periodic and sporadic tasks generate an infinite amount of jobs respecting the corresponding period or task activation per interval parameters. For simplicity, we assume that the first job of all periodic tasks is activated at time 0 and that the relative deadline D_i of τ_i is equal to its period, *i.e.,* $D_i = W_i$. Furthermore, the worst-case parameters of $C_i(\ell)$ are monotonically increasing for increasing ℓ and the best-case parameters are monotonically decreasing, respectively. Namely, the min./max. range of execution times and shared resource accesses in $C_i(\ell)$ is included in the corresponding range of $C_i(\ell + 1)$, for $\ell \in \{1, \ldots, \chi_i - 1\}$. Note that the best-case parameters are only required for a tighter response time analysis. If not available, they are assumed equal to 0.

Example 1 For illustration purposes, Table 1 presents the system model for our case study, a flight management system (FMS), which is discussed in more detail in Sec. 9.1 and is used as a running example throughout the paper. The FMS is a dual-criticality system, *i.e.,* $L = 2$. The second column contains the criticality level $\chi_i \in \{1, 2\}$ of each FMS task τ_i. The period

[2] Conventional sporadic tasks assume $a_i = 1$.

W_i of the sporadic task 'GPSConfig' is in fact its interval I_i, and $a_i = 1$. As the table shows, for high-criticality tasks ($\chi_i = 2$), the level-1 worst-case execution time (WCET), $e_i^{max}(1)$, is lower than the respective level-2 WCET, $e_i^{max}(1)$. Therefore, in the 'emergency' situation where the level-1 WCETs turn out to be insufficient, the high-criticality tasks are eligible to continue their execution up to their level-2 WCET. For low-criticality tasks ($\chi_i = 1$), *e.g.,* 'Filter', the situation is reverse. In the case of 'emergency' (after high-criticality tasks overrun their level-1 WCET), the low-criticality tasks may receive a smaller execution budget than their 'normal' level-1 WCET, in order to free up resources for high-criticality tasks. In Table 1, for convenience, we specify this budget as 'level-2 WCET', $e_i^{max}(2)$. In fact, this budget corresponds to the degraded execution profile $C_{i,deg}$ of low-criticality tasks, *i.e.,* $e_i^{max}(2) = e_{i,deg}^{max}$, if $\chi_i = 1$. The resource access counts, μ_i^{max}, which are the same at all levels, in this example, are shown in the last column. The term 'RTE' describes a shared resource and will be clarified later, in Sec. 8.3. All best-case parameters, e_i^{min} and μ_i^{min}, $\forall \tau_i \in \tau$, are considered zero and hence, omitted in the table.

The bounds for the execution times and accesses can be obtained by different tools. For instance, at the lowest level of assurance ($\ell = 1$), the system designer may extract them by profiling and measurement, as in [47]. At higher levels, certification authorities may use static analysis tools, such as the abstract interpretation suite aIT [3], with more and more conservative assumptions as the required confidence increases. The execution profile $C_i(\ell)$ for each task τ_i is derived only for $\ell \leq \chi_i$. For $\ell > \chi_i$, there is no valid execution profile since certification at level ℓ ignores all tasks with a lower criticality level. At runtime, if a task with criticality level greater than χ_i requires more resources than initially expected, then τ_i may run in degraded mode with execution profile $C_{i,deg}$. Note that we forbid the case where a task τ_i consumes more resources than its own criticality level profile $C_i(\chi_i)$.

Dependencies can be defined between tasks with equal periods. We represent these by a directed acyclic graph $\mathcal{D}ep(\mathcal{V}, \mathcal{E})$, where each node $\tau_i \in \mathcal{V}$ represents a task, and an edge $e \in \mathcal{E}$ from τ_i to τ_k implies that within a period the job of τ_i must precede that of τ_k. The dependencies between the FMS tasks of Example 1 will be defined later on.

Our *DOL-Critical model of computation* (MoC) extends the above system model by defining an inter-task communication method realized by means of shared objects, which are called *data channels*. The channels are written and read by tasks in a *non-blocking* fashion. The non-blocking communication is selected to avoid (potentially unbounded) blocking delays, and hence to facilitate scheduling, timing analysis and certification of mixed-criticality systems. Instead of blocking, we use dependencies to ensure functionally deterministic communication. Two tasks (of equal or different criticality levels) that communicate should have a dependency between them, going in the same or in the opposite direction as the flow of data. Recall that, in our model, a dependency implies equal periods. Therefore, to let two different-period tasks communicate, we transform them into equal-period tasks with a common-divisor period and internal skipping of excess activations. The DOL-Critical MoC is further discussed in Sec. 5.1.

The MC model described above extends Vestal's model [64] by: (i) Introducing the shared resource access bounds, which are required for timing analysis on shared-resource multicores; (ii) Defining the degraded mode for lower criticality tasks. Guaranteeing a minimal functionality for such tasks (instead of dropping them as in the original model) has been also advocated in [52,58,11]; (iii) Introducing a consistent MoC where applications, such as the flight management system of Example 1, can be programmed.

3.2 Shared-Resource Multi-core Architecture Model

We consider a set $\mathcal{P}$ of m processing cores, $\mathcal{P} = \{p_1, \ldots, p_m\}$. Here, the cores are identical but our approach can be generalized to heterogeneous platforms. The mapping of a task set τ to the cores in $\mathcal{P}$ is defined by function $\mathcal{M}_\tau : \tau \rightarrow \mathcal{P}$. In our work, $\mathcal{M}_\tau$ is *not* given, but it is calculated by our optimization approach in Sec. 4.2.

Each core in $\mathcal{P}$ has access to a private cache memory and to a shared general-purpose memory. The code and data of the tasks in τ as well as the data channels used for the inter-task communications are assumed to fit in the shared memory. This abstract model gives a partial view of commercial many-core platforms, for instance the Kalray MPPA®-256 [15] and the STHorm/P2012 [42]. These platforms are on-chip networks of shared-memory clusters, with 16 cores per cluster. Currently, our model is restricted to a single cluster, since exploiting more on-chip clusters would require network-on-chip management, which is outside the scope of this paper.

For timing analysis, we need to consider shared resources which are accessed synchronously, namely which cause execution on the cores to stall until any pending access requests are served. We assume that such resources, for instance a memory bus, can be accessed by only one core at a time, and that once granted, a resource access is completed within a fixed time interval, T_{acc}. Access to the shared resources can be arbitrated according to any event- or time-triggered scheme, e.g., round-robin or time-division-multiple-access. To enable safe timing analysis under resource contention, we consider hardware platforms without timing anomalies, such as the fully timing compositional architecture defined in [66], where execution and communication times can be decoupled. Note that the MPPA®-256 cores have been shown to be fully timing compositional [15].

3.3 Mixed-Criticality Schedulability Conditions

Under the above system assumptions, we seek a *feasible* schedule for the MC task set τ on the cores $\mathcal{P}$, which enables *temporal isolation* among criticality levels and *incremental design*. Below we define the properties of feasibility, isolation and incremental design. The feasibility conditions follow from Vestal's schedulability conditions, by considering shared resource accesses and degraded mode. The isolation and incremental design conditions are introduced to capture the certification-induced requirements in safety-critical domains.

Definition 1 (Execution Scenario) At runtime, the tasks follow a *level-ℓ scenario* in a given time interval if, within this interval, the resource demand for all executing jobs of tasks τ_i with criticality $\chi_i \geq \ell$ complies with the execution and access bounds of profiles $C_i(\ell)$. If $\ell > 1$, there must be at least one job of a task τ_j, for which the resource demand violates the bounds of $C_j(\ell - 1)$.□

The term *resource*, in this context, refers to both processing time and shared-resource access. Initially, during a sufficiently small time interval, the tasks follow a level-1 scenario. When we extend this interval, the first job of a task τ_j, whose resource demand exceeds $C_j(1)$, switches the current scenario to level 2. Later, a job of the same or another task $\tau_{j'}$, whose resource demand exceeds $C_{j'}(2)$, switches to level 3, and so on. The currently assumed scenario level (as well as the reference interval) is regularly reset back to level 1 at specific – for the given policy – time instances, when all cores and shared resources should be idle.

Definition 2 (Feasibility) A schedule is *feasible* if for any level-ℓ scenario ($\ell \in \{1, \cdots, L\}$), it guarantees the conditions:

– the jobs of each task τ_i, satisfying $\chi_i \geq \ell$, receive enough resources between their activation time and deadline to meet their real-time requirements according to execution profile $C_i(\ell)$,
– the jobs of each task τ_i, satisfying $\chi_i < \ell$, receive enough resources between their activation time and deadline to meet their real-time requirements according to execution profile $C_{i,deg}$.□

Example 2 For the FMS application of Example 1, if a high-criticality task from the upper part of Table 1 exceeds its $e_i^{max}(1) = 1$ ms, then the tasks switch from a level-1 to a level-2 scenario. If only the level-1 scenario was possible ($e_i^{max}(1)$ was never exceeded), all tasks could easily meet their deadlines while executing on a single core, even if we assume that RTE accesses add a reasonably small overhead[3]. However, due to the large level-2 WCETs, $e_i^{max}(2)$,of high-criticality tasks, multiple cores are required for a feasible schedule even when the low-criticality tasks run in degraded mode. Note that when running on multiple cores, the tasks will experience *interference* upon simultaneous RTE accesses.

Definition 3 (Temporal Isolation) A schedule satisfies non-symmetric *temporal isolation* if all tasks of criticality level ℓ suffer no interference from tasks with lower criticality level, for all $\ell \in \{1, \ldots, L\}$. Namely, the execution and access activities of a task τ_i do not delay in any way any task with criticality level higher than χ_i. □

Definition 4 (Incremental Design) A scheduling algorithm enables *incremental design* if adding new tasks of lower criticality into the system can be done without altering the schedule for the existing tasks.□

Note that the property of incremental design is based upon non-symmetric temporal isolation. The two properties imply that if the schedule of a task set τ is certified as feasible, the certification procedure will not need to be repeated if new, lower-criticality tasks are added later to the system. This is highly desirable, since repeating the certification process of already certified tasks if the system is gradually incremented results in excessive costs [8].

4 Mixed-Criticality Scheduling on Resource-Sharing Multicores

The previous section presented the abstract models of mixed-criticality applications and multi-core architectures that can be specified in DOL-Critical. Here, we focus on determining the *mapping*, i.e., the binding of the application tasks to processing cores, and *scheduling*, i.e., the execution order of the tasks on the cores. For the problem of mixed-criticality multi-core scheduling, policies that explicitly address the effects of interference on shared resources need to be considered. For this, we select the Time-Triggered scheduling policy with Synchronization points (TTS) [25], which is designed for temporal isolation and incremental design. Temporal isolation is achieved by allowing only a statically known subset of tasks in τ with the *same* criticality level to be executed across the cores $\mathcal{P}$ at any time. This is necessary for deployments on commercial-off-shelf-platforms which do not provide special support for criticality isolation on their shared resources. Allowing a static subset of tasks to be executed in parallel enables, additionally, tight worst-case timing analysis, which is also crucial for certification.

Sec. 4.1 presents the main principles of the TTS scheduling policy from [25], assuming that a TTS schedule for a particular task set and platform is *given*. We show how to determine

[3] RTE specifies a shared resource, as described in Sec. 8.3.

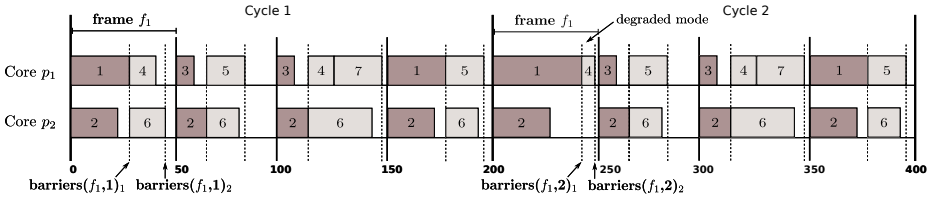


Fig. 2: TTS schedule example: 2 cycles (dark annotation: crit. level 2, light: crit. level 1)

a TTS schedule in Sec. 4.2. The design space exploration method of Sec. 4.2 is implemented in the tool suite for DOL-Critical language [18]. This tool suite is used both to provide the input and to analyze the output (via a feedback loop) of the automata-based compilation framework DOL-BIP-Critical.

4.1 TTS Scheduling

The non-preemptive TTS scheduling policy combines time- and event-triggered task execution. The tasks are mapped statically to cores and no migrations are allowed. A TTS schedule repeats itself over a *scheduling cycle* equal to the hyper-period H of the tasks in τ (least common multiple of periods). The scheduling cycle consists of fixed-size *frames* (set $\mathcal{F}$), and each frame is divided further into L flexible-length *sub-frames*. A sub-frame contains only jobs of the same criticality level, and the sub-frames are ordered within a frame in decreasing order of criticality. Within a sub-frame, tasks are scheduled sequentially on each core following a predefined order, namely every task is triggered upon completion of the previous one. The jobs executed in a sub-frame have been generated at or before the respective frame start and have deadline at or after the frame end. The beginning of frames and sub-frames is synchronized among all cores in $\mathcal{P}$. The (fixed) frame lengths can differ, but they are upper bounded by the minimum period in τ. Each sub-frame (except the first of a frame) starts once all jobs of the previous sub-frame complete execution across all cores. Synchronisation is achieved dynamically at runtime via a barrier mechanism, for the sake of efficient resource utilization.

Example 3 An illustration of a TTS schedule is given in Fig. 2 for a dual-criticality set of seven tasks, with hyper-period $H = 200$ ms. Fig. 2 depicts two consecutive scheduling cycles. The solid lines define the frames and the dashed lines the sub-frames, i.e., potential points, where barrier synchronisation is performed at runtime. The TTS scheduling cycle ($H = 200$ ms) is divided into four frames of equal lengths (50 ms). Each frame has $L = 2$ sub-frames: the first for criticality 2 (high) and the second for criticality 1 (low), respectively. At runtime, the length of each sub-frame varies based on the different execution times and memory accessing patterns that the concurrently executed tasks exhibit. For example, the first sub-frame of f_1 finishes earlier when τ_1, τ_2 run according to their level-1, i.e., low-criticality execution profiles (cycle 1) than when at least one task runs according to its level-2, i.e., high-criticality profile (cycle 2).

Despite the dynamic runtime behavior, the sub-frame worst-case lengths can be computed offline for a given TTS schedule by applying timing analysis under shared-resource interference. Function $barriers : \mathcal{F} \times \{1, \ldots, L\} \rightarrow \mathbb{R}^L$ defines a vector with the worst-case length of all sub-frames of a frame when a particular scenario ℓ is followed. We denote the worst-case length of the k-th sub-frame of frame f for the level-ℓ scenario

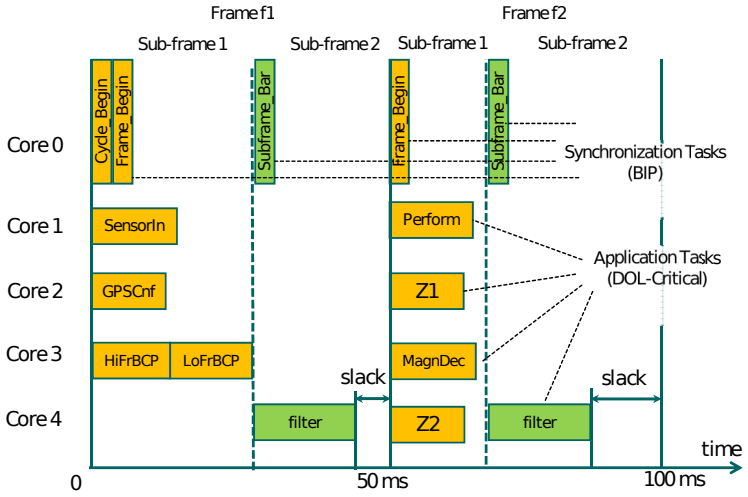


Fig. 3: TTS schedule generated for the FMS application in DOL-BIP-Critical flow

as $barriers(f, \ell)_k$. Note that the k-th sub-frame of f contains tasks of criticality level $\ell' = (L - k + 1)$. Also, ℓ' corresponds to the highest level execution profile that the tasks in subframe k exhibit at runtime: $\ell \leq \ell'$. For $\ell' > 1$, execution in later sub-frames of f may be degraded.

Example 4 Fig. 3 shows the TTS schedule that is generated in our DOL-BIP-Critical flow for the FMS application from Example 1, when we assume five available cores. In our flow, we add to the scheduler a model of runtime overhead of the TTS scheduling policy. The model consists of so-called *synchronization tasks*, which are exclusively executed on Core 0. The execution profiles of those tasks are extracted from the implemenentation of the TTS schedule in BIP automata language. As their names suggest, they represent synchronization of a TTS cycle, frame and sub-frame barrier. High-criticality tasks are depicted in orange and are executed in the first sub-frame, $k = 1$ ($\ell' = 2$), of each frame $f \in \{1, 2\}$. The actual length of this sub-frame depends on execution scenario $\ell \in \{1, 2\}$ and is bounded by $barriers(f, \ell)_1$, respectively. The second sub-frame, $k = 2$ ($\ell' = 1$), contains the lower-criticality tasks, depicted in green. Its length is bounded by $barriers(f, \ell)_2$, where $\ell = 1$, since there is no level-2 execution profile defined for low-criticality tasks. Note that tasks 'HiFrBCP' and 'LoFrBCP' are not executed in parallel due to FMS-specific dependencies discussed later in Sec. 9.1.

Runtime behavior. Given a feasible TTS schedule and the *barriers* function, the scheduler manages task execution on each core within a frame $f \in \mathcal{F}$ as follows:

- For the k-th sub-frame, the scheduler triggers sequentially the corresponding jobs following the predefined order. Upon completion of all jobs on the core, it signals an event and waits until the remaining cores reach the barrier (all jobs of the sub-frame are completed).
- Let the elapsed time from the beginning of the frame until the barrier synchronisation of the k-th sub-frame be t. Below, ℓ_{max} defines the maximum-level execution profile in the frame:

$$\ell_{max} = \underset{\ell \in \{1,\ldots,L\}}{\operatorname{argmin}} \left\{ t \leq \sum_{j=1}^{k} barriers(f, \ell)_j \right\}, \tag{1}$$

The scheduler will trigger jobs in the next sub-frame such that tasks with criticality level lower than ℓ_{max} run in degraded mode.
- The two previous steps are repeated for each sub-frame, until the last sub-frame is reached.

Note that the decision on whether a task will run in degraded mode affects only the current frame. The interval for observing the execution scenario is reset at frame boundaries.

Feasibility. A given TTS schedule is feasible if and only if the following condition holds for all scenarios $\ell \in \{1, \cdots, L\}$:

$$\sum_{k=1}^{L} barriers(f, \ell)_k \leq \mathcal{L}_f, \forall f \in \mathcal{F}\,, \tag{2}$$

where $\mathcal{L}_f$ denotes the length of frame f. If the condition holds for all frames $f \in \mathcal{F}$, it follows that all scheduled jobs can meet their deadlines when running according to their level-ℓ profiles.

Temporal isolation & incremental design. The TTS scheduling policy preserves temporal isolation, since only tasks of the same criticality level can run simultaneously on the platform. The isolation is non-symmetric because of the criticality-monotonic dynamic scheduling of the sub-frames within each frame: The jobs of a sub-frame cannot be delayed in any way by lower-criticality jobs, however higher-criticality jobs can implicitly delay the execution of lower-criticality by shifting the barrier synchronisation point. The TTS policy enables incremental design, since adding new tasks in sub-frames has no impact on previous sub-frames. In addition, the cross-core utilisation of frames is bounded at design time and the remaining *slack intervals*, where all cores are idle, can be even filled by new frames of other applications. Note that for incremental design, an attractive optimisation goal for a scheduler is to 'pack' the sub-frames as evenly across the core as possible, in order to minimize function $barriers$ and maximize the slack intervals.

Example 5 In the schedule of Fig. 3, the feasibility requirement translates into non-negative slack intervals at the end of each frame. Temporal isolation is apparent from the fact that only tasks of the same criticality level are executed in parallel. Finally, the incremental design could be illustrated if we *e.g.,* replicated task 'Filter' on other cores, which would have no impact on the already scheduled high-criticality tasks.

4.2 Mapping and Scheduling Optimization

In DOL-Critical, for a given application and target architecture, we seek an optimal TTS schedule. We define a schedule as *optimal* if (i) it is *feasible*, and (ii) the worst-case total sub-frame lengths are *minimal*. The latter condition implies maximal aggregate slack intervals, which can be used for incremental design.

The problem of optimal task mapping on multiple cores is known to be NP-hard in most cases, resembling the combinatorial bin-packing problem [43]. To tackle this challenge, we propose and implement in our tool-chain the *Mixed-Criticality Mapping and Scheduling Optimization* (MCMSO) tool. MCMSO takes as input a mixed-criticality task set τ and a set of cores $\mathcal{P}$, and returns the mapping function $\mathcal{M}_\tau$ of tasks to cores and a feasible TTS schedule if at least one such schedule exists.

MCMSO performs design space exploration with two main objectives. The primary objective is to find feasible solutions. The second objective is to improve the quality of a

feasible solution by maximizing the total size of slack intervals available for incremental design. To perform the exploration, MCMSO implements a heuristic approach based on simulated annealing [38]. In summary, the MCMSO approach is described by the following steps:

1. Dimension the TTS scheduling cycle and frame lengths based on the periods of tasks in τ.
2. Generate a random schedule of the jobs of τ within hyper-period H on the cores of $\mathcal{P}$ and the frames $\mathcal{F}$ of the TTS cycle, such that all dependencies are respected.
3. Apply a simulated annealing approach to generate and explore neighboring mappings (assignments of tasks to cores) and schedules (assignment of jobs to sub-frames), until an optimized solution is found or a given computational budget is exhausted.

To express the optimality criteria, we define the cost function of the optimization problem as:

$$Cost(S) = \begin{cases} c_1 = \max_{f \in \mathcal{F}} \left\{ \max_{\ell \in \{1,\ldots,L\}} late(f,\ell) \right\} & \text{if } c_1 > 0 \\ c_2 = \|barriers\|_3 & \text{if } c_1 \leq 0 \end{cases} \quad (3)$$

where $late(f,\ell)$ expresses the difference between the worst-case completion time of the last sub-frame of f and the length of f:

$$late(f,\ell) = \sum_{k=1}^{L} barriers(f,\ell)_k - \mathcal{L}_f. \quad (4)$$

Component c_1 of the cost function provides a measure of "infeasibility". If $late(f,\ell) > 0$, the tasks in f cannot complete execution by the end of the frame for their ℓ-level execution profiles. Therefore, with this cost function, we initially guide the design space exploration to find a feasible solution (by penalising infeasible solutions). When such a solution is found, cost c_1 becomes negative or 0. Thereafter, c_2, i.e., the 3-norm of all sub-frame lengths, $\forall f \in \mathcal{F}, \forall \ell \in \{1,\ldots,L\}$, is used to minimize the worst-case lengths of all sub-frames. The 3-norm of a vector x with n elements (here, positive real numbers) is defined as $||x||_3 := \left(\sum_{i=1}^{n} |x_i|^3\right)^{1/3}$. We selected this value to map the flattened vector with the $barriers$ values, for all sub-frames of the frames $f \in \mathcal{F}$ and for all $\ell \in \{1,\ldots,L\}$, over other norms, such as the average or the Euclidean norm, because empirically it provides a good trade-off between reducing the worst-case sub-frame lengths (to ensure schedulability) and enabling progress in the optimization.

The simulated annealing approach for optimizing a TTS schedule is detailed and evaluated extensively in [25].

Timing Analysis. MCMSO is tightly coupled with a timing analyzer in our design flow (Fig. 1). During design space exploration, for every visited TTS schedule this tool performs worst-case response time analysis for all tasks in each sub-frame and each execution scenario, in order to compute the worst-case sub-frame lengths, i.e., the function $barriers$. Real-time analysis of concurrently executing tasks under resource contention is a highly complex problem. We have addressed this by applying the theory of timed automata [4] and real-time calculus [61] in [24], and by an analytic arbitration-dependent approach in [25]. The latter approach is implemented in DOL-Critical. For brevity, we omit the timing analysis here and refer the interested readers to the aforementioned publications.

5 Description Language DOL-Critical

In our design flow, the DOL-Critical language is used for specifying a mixed-criticality application (Sec. 3.1) and a target architecture (Sec. 3.2). The same language, specifically the integrated MCMSO tool and the timing analyzer (Sec. 4.2), are used for design space exploration and determination of a TTS schedule with maximal aggregate slack time. This section provides details about the user-defined specifications of mixed-criticality applications and multi-core architectures, as well as the auto-generated specification of the mapping and scheduling solution in DOL-Critical.

5.1 Specification of a Mixed-Criticality Application

To specify an application that complies with the MC model of computation of Sec. 3.1, in DOL-Critical, we distinguish between two layers: a *functional* layer which consists of tasks and data channels, and a *control* layer which consists of task controllers and task dependencies. The specification of each task contains source code and its execution profiles, while the task controllers (one per task) specify the tasks' activation patterns and deadlines. For the specification, DOL-Critical uses two distinct languages: C/C++ to program the task functionality and complex activation patterns, and XML for the task properties, connections through data channels and dependencies. The choice of these languages is based on practical reasons. C/C++ allows to reuse existing legacy code. XML is easy to handle due to the large number of available tools. Alternative choices are ADA, Simulink, and SDL for functional code [48], and UML or AADL for task control and data interfaces.

Inter-task communication. The DOL-Critical model of computation supports two concrete types of the defined in Sec. 3.1 data channels: blackboards (buffers) and mailboxes (queues). Note that unlike most dataflow languages, we use non-blocking communication and do not force the tasks to write/read a fixed number of tokens at each execution. For this reason, every data channel is equipped with a *validity bit*, which indicates that the channel is not empty.

For simplicity, we present *blackboard* as a protected shared variable[4] that can be written via a 'write' port of a single task and read via a 'read' port by one or more tasks. The reading operation does not change the state of the blackboard, which preserves the last written value. If no value was previously written, the reading operation returns with validity bit set to 'false'.

A *mailbox* connects one writing task with one reading task. It is a bounded queue allowing to store several data elements of the same type. The queue length is determined at design time according to the needs of the given application. It is typically desirable that a writing attempt to a full mailbox never occurs in the nominal mode of execution. If this situation still occurs, the writing operation will not block the writer task, but instead it will return an error code. Similarly, reading from an empty mailbox does not cause blocking, but returns with validity bit set to 'false'.

Example 6 A partial example of a DOL-Critical application specification can be found in Listing 1 (XML) and Listing 2 (C). Note that in the context of DOL-Critical, we use the terms task and process interchangeably. The application (Fig. 4) features one periodic, implicit-deadline task, `square`. Task `square` reads floating-point values from a mailbox,

[4] In reality, the blackboard is defined and implemented as a more complex object [18], for which the given simplified definition provides a reasonable abstraction.

```
<process name="square" criticality="2">
  <superblock>
    <info level="1" minAccess="5" maxAccess="10"
          minExecution="7" maxExecution="18"/>
    <info level="2" minAccess="5" maxAccess="20"
          minExecution="5" maxExecution="25"/>
  </superblock>
  <port type="in_data" name="pIN"/>
  <port type="out_data" name="pOUT"/>
  <port type="in_event" name="p2">
    <event name="start"/>
  </port>
  <source location="square.c"/>
</process>

<controller name="Ctrl_square" deadline="0.2">
  <activation type="periodic">
    <parameter name="period" value="0.2"/>
  </activation>
  <port type="out_event" name="p1">
    <event name="start"/>
  </port>
</controller>

<data_channel name="dataIN" type="mailbox" size="8" length="252">
  <port name="pdOUT" type="out_data"/>
</data_channel>
<connection name="dataInToSquare">
  <port name="pdOUT"/>
  <port name="pIN"/>
</connection>
```

Listing 1: XML source code for process `square` and data channel `dataIN`

```
struct Square_state {
  int index;
  int length;
};
struct DOLCData {
  bool valid;
  float value;
};

void Square_init(Square_state *ST) {
  ST->index =0;
  ST->length = 200;
}

void Square_fire(Square_state *ST, int mode) {
  DOLCData x,y;

  if (mode == DEGRADED) {
    return;
  }

  if (ST->index < ST->length ) {
    DOLC_read("pIN", &x, sizeof(float));
    if (x.valid) {
      y.value = x.value * x.value;
      y.valid = true;
      DOLC_write("pOUT", &y, sizeof(float));
    }
  }
  ST->index = ST->index + 1;
}
```

Listing 2: C source code for process `square` (`square.c`)

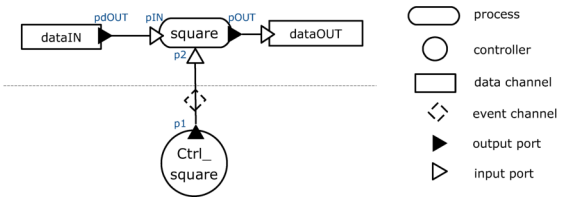


Fig. 4: `Square` Application Example

`dataIN`, computes the square of them, and writes the result to mailbox `dataOUT`, as indicated by the source code in `square.c`. It is characterized by safety criticality level 2 (high in a dual-criticality system) and its execution time (CPU cycles) and number of resource accesses are given for both execution levels. Note that the parameter ranges for level 1 are included into the respective parameter ranges of level 2. The controller `Ctrl_square`, is responsible to activate `square` periodically every 0.2 seconds. Communication between the controller and the task is achieved via an event channel. Specifically, `Ctrl_square` sends a control event `start` to `square` to activate it. The mailbox `dataIN`, from which `square` reads, corresponds to a queue with a capacity of 8 elements, each with a size of 2 bytes.

The C/C++ code that defines the functionality of the tasks is written in a DOL-Critical specific *dialect*. The data channels, control events (for communication between controllers and tasks), and ports of data channels and tasks, which are defined in XML, are re-used in the C/C++ code in a way that establishes a unique connection between the XML and the C/C++ specification (see e.g., port ”pIN” in Listings 1,2). Each task has a state data structure, an initialisation subroutine, and a subroutine defining one execution of a job. In the DOL-Critical application programming interface (API), these are denoted `<Task>_state`, `<Task>_init()`, and `<Task>_fire()`, respectively. Furthermore, the API supports two main functions for the communication between tasks: `DOLC_read()` and `DOLC_write()`

(see Fig. 4 for an example). These functions enable reading/writing from/to a data channel and have different semantics depending on the type of the target data channel. The complete semantics of the DOL-Critical programming interface are omitted here for brevity. However, a detailed presentation of the API as well as XML templates for the specification of mixed-criticality applications in DOL-Critical can be downloaded from [18].

5.2 Specification of a Target Architecture and a TTS Schedule

For the specification of a resource-sharing multicore that complies with the model of Sec. 3.2, the computation and communication components, along with their attributes and connections, are described in XML format. Specifically, one can model processing cores with attributes such as their frequency, and shared resources with their arbitration policy and maximum access latency. The abstraction level defines the accuracy of the timing analysis, which is performed during design space exploration by the MCMSO tool (Sec. 4.2).

After the scheduling optimization, the MCMSO tool exports the optimized TTS schedule (see Fig. 2 for reference) in XML format. This specification includes (i) the mapping of tasks to cores, (ii) the dimensioning of the TTS scheduling cycle (period, number of frames, frame lengths), (iii) the values $barriers(f, \ell)_k$ for all sub-frames k of frame $f \in \mathcal{F}$ and for different execution scenarios $\ell \in \{1 \dots L\}$, (iv) the execution order of the assigned tasks on each core and each TTS frame.

Customized XML schemata are used for describing the format of architecture and mapping specifications. These specifications are used as inputs for timing analysis during design space exploration as well as software synthesis after they are compiled into the concurrency language BIP, which is presented in the following section.

6 Concurrency Language for Mixed-Criticality Systems – BIP

The cornerstone of our rigorous system design approach is the WYVIWYG principle, realized via an automata-based language. We refer to it as '*concurrency language*', as it defines the concurrency and timing semantics of all system software components. After compilation from system specification into a concurrency language, one obtains an executable model that can be simulated for functional validation. This model is also used as the input for system analysis and code generation. In our design flow, the concurrency language is BIP.

Under 'BIP' we refer to the so-called 'RT-BIP' dialect [2], which is designed to express networks of connected timed automata components (Sec. 6.1). In the present work, we extend BIP from timed to task automata, by allowing *self-timed* automata transitions. This extension allows expressing control decisions based on runtime monitoring of task response times in timed automata. This feature is important for runtime resource management mechanisms, such as those employed for mixed criticality. For example, recall that the TTS scheduling policy makes online decisions based on the exhibited sub-frame lengths at runtime. A particular feature of BIP is the ability to specify a *network* of components, so that multiple tasks can be executed in different components concurrently. This makes it particularly suitable for multi-core platforms. Our extensions to the original RT-BIP dialect are presented in Sec. 6.2.

6.1 Introduction to BIP

To familiarise the readers with BIP notation, Fig. 5 shows a BIP example, representing two tasks, A and B. These blocks can be scheduled on one of the two available threads

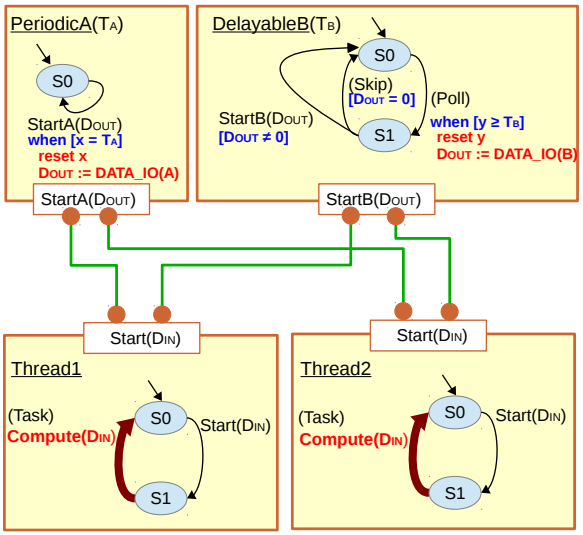


Fig. 5: BIP model example: four single-port components and four dual-port connectors

running on two different cores. The model consists of four components, namely, 'PeriodicA', 'DelayableB', 'Thread1' and 'Thread2'. All the components are defined by an automaton and a set of *ports* (shown in white rectangles), used for connecting to other components via *connectors* (shown as green lines that join the bullets).

A BIP component has multiple *locations*, denoted in Fig. 5 as 'S0', 'S1'. The *execution run* of a component consists of going from location to location by taking a *transition*, denoted by an arc. For example '(Skip)' is a transition from location 'S1' to location 'S0' in component 'DelayableB'. Each component has an *initial transition*, which brings it to initial location at system start. Initial transition is shown as an arc without origin pointing to the initial location, such as location 'S0' in 'DelayableB'. A transition may have an *enabling condition* and may trigger some *action*. In our figures, we show the conditions in blue color and square brackets, *e.g.,* component 'DelayableB' has condition '$[D_{\text{OUT}} \neq 0]$' for transition 'StartB'. The actions are shown in red color.

The transition labels such as 'StartB' signify a port of the component, in which case the transition *participates in interactions* through this port, which means that it is synchronized with transitions in other components whose ports are connected, *e.g.,* 'StartB' may interact with 'Start' in 'Thread1' or 'Thread2'. Note that a port may participate in one interaction at a time. In our example, each port is linked to two connectors, so if both of them have an enabled interaction, a non-deterministic choice has to be made between them. There are also *internal transitions*, not associated to ports, executed by a component independently. We put their labels in parentheses, *e.g.,* '(Skip)' and '(Poll)'.

In BIP, every component is seen as an object in an object-oriented programming sense. Every component encapsulates some data and some subroutines to manipulate the data. The actions of transitions can call subroutines written in an imperative language (C/C++). In the figures, the actions are depicted as blocks of pseudo-code in red color, *e.g.,* in component 'DelayableB', transition '(Poll)' executes action 'D_{OUT} := DATA_IO(B)', where a subroutine is called and its return value is assigned to variable 'D_{OUT}'. The actions have access only to the local variables of their component. Nevertheless, some variables are classified as 'OUT' and 'IN' communication variables, bound to ports, *e.g.,* variables $D_{\text{IN, OUT}}$ are bound to port

'Start'. The components send data from 'OUT' to 'IN' variables at interactions via ports. For example, port 'Start(D_{IN})' receives the new value of D_{IN} from the D_{OUT} of either 'StartA' or 'StartB', depending on the component with which it interacts. Note that the data exchange between ports precedes the transitions, *e.g.,* port 'StartA(D_{OUT})' sends the value of D_{OUT} *before* it is modified by the respective transition.

As for the data variables, in this work we consider four main types: integer, Boolean, reference, and queue. A *reference* is a pointer to a user-type object that is allocated at component initialisation. Our models for critical systems do not dynamically allocate data after system initialisation. A *queue* is a circular buffer of statically-known size. Unless explicitly done otherwise in the initial transition or in natural-language annotations, in the presented figures we assume that the initial transition implicitly sets the data variables to zero in the case of integers, 'False' for Booleans *etc.* Besides data variables, the components can have compile-time parameters, such as period T_A and minimal execution interval T_B in Fig. 5.

The condition to execute a transition in fact consists of two parts: a data condition and a timing constraint, indicated by the keyword 'when'. The *timing constraint* defines an interval of time when a transition may be enabled. By default it is 'always', *i.e.,* the whole time axis.

To define the timing constraints a component uses private *clock variables*. The clocks are real-valued variables that are initialized to zero and whose values are continuously and synchronously increasing with the passage of physical time. In our models, we use letters x, y and t for the clocks, e.g., the model in Fig. 5 uses two clocks. The usage of clocks is restricted to two possible scenarios. Firstly, a clock can be reset to zero inside a transition action (*e.g.,* 'reset x' in 'PeriodicA'). Secondly, it can be used in the timing constraint of a transition, see, (*e.g.,* 'when $x = T_{\text{A}}$' in 'PeriodicA').

In our models we assume that all transitions are marked as '*urgent*' in BIP. The presence of 'urgency' attribute means that the transition should start *as soon as (and no later than)* the given transition and all those that participate in the same interaction (if any) get enabled. For example, consider timing constraint 'when $[y \geq T_B]$' in Fig. 5. Due to this constraint, if component 'DelayableB' is in location 'S0', then it should execute transition '(Poll)' immediately when it sees that clock y has reached a value at least equal to T_B. Note that the 'urgency' property is usually not directly available in timed automata languages, but it is very useful for modeling compute-intensive real-time systems, where typically the system must make progress *immediately* when several conditions become true. For example, in the TTS scheduling policy the barrier synchronization should occur immediately when all tasks scheduled in a given sub-frame finish their execution.

6.2 BIP Extension for Modeling the Tasks

By default, BIP assumed that all data-processing actions cost zero time (at least, conceptually). However, real-time tasks may occupy the processing cores at significant utilisation levels, and to properly model them one should allow executing their data-processing operations in non-zero time. Therefore, in the extended version of BIP, we distinguish between the 'starting' and the 'finishing' times of a transition, and we refer to the time duration in between as *transition response time*. Further, we introduce the '*self-timed*' attribute for the transitions and we assume that all transitions are conceptually instantaneous (*i.e.,* have zero response time) unless they have this attribute. A transition marked as self-timed has a response time equal to the time required to finish the corresponding action on a finite-speed physical resource. This can take any time duration, not known at the moment when the transition starts.

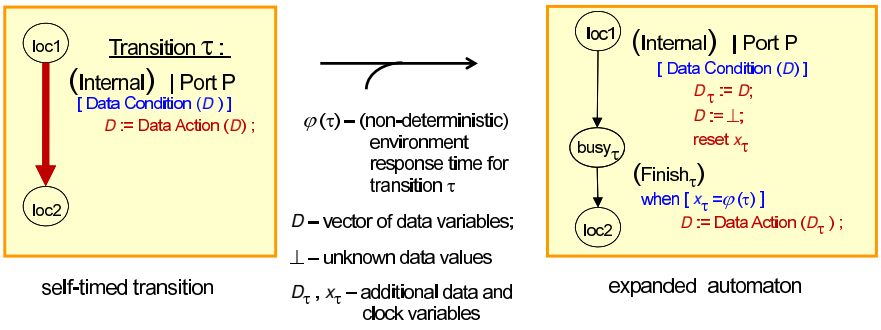


Fig. 6: Modeling tasks in BIP

We use *internal self-timed transitions* to represent task processing steps and *self-timed interactions via ports* to represent inter-task communication. In our figures, we denote self-timed transitions by thick arrows, *e.g.,* '(Task)' transitions in Fig. 5. Note that by putting a self-timed transition in between two instantaneous transitions, one can measure its response time by resetting a clock before and checking the clock value after the self-timed transition. This is a necessary feature to program scheduling policies, especially mixed-criticality ones, such as TTS.

Though the self-timed transitions represent a new concept added into BIP language to model tasks, at the *semantics level* the behavior can be expanded into an equivalent model in the default BIP language, *i.e.,* timed automata with instantaneous transitions. Nevertheless, at the implementation level, the BIP framework needed certain extensions to handle these transitions correctly. Fig. 6 shows a self-timed transition τ of a task automaton in the extended BIP and its expansion into timed automata of the 'default' BIP. In the expanded model, transition τ is represented by two instantaneous transitions, one modeling the start and other one the finish. In between these transitions, there is a location 'busy$_\tau$', which models the state where the system is busy waiting until the platform executes transition τ. Note that the data variables are explicitly set into 'unknown' state, because during the execution they can potentially take arbitrary values. Note also that if the transition interacts with other components via a port, then in the expanded automaton the port is associated to the start transition, which indicates that the interacting components synchronize with each other at the start of their transitions.

An additional clock x_τ measures the elapsed time since the start and the execution of transition τ. The execution finishes when the response time of transition τ, denoted $\varphi(\tau)$, has been reached. Model-wise, it is important to observe that the 'Finish$_\tau$' transition and time $\varphi(\tau)$ are controlled not by the system itself, but rather by the *environment*. Indeed, the software cannot directly influence the time it takes to execute a given, arbitrarily complex piece of the task's code. This is determined by the target platform, which actually acts here as environment. For simulation or modeling purposes, one can make an abstraction of the the environment by letting $\varphi(\tau)$ take non-deterministic values. However, when *implementing* the BIP program on a real platform, the BIP system may not 'decide' by itself, non-deterministically, how long delay $\varphi(\tau)$ should be. Instead it should let the environment 'decide' this. Therefore, it should start the execution of the transition on the platform and wait until the platform eventually signals its completion. This observation makes the difference between executing the BIP model on the left and on the right of Fig. 6.

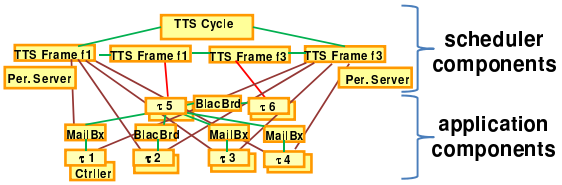


Fig. 7: Overall BIP software model obtained by compilation from DOL-Critical

7 Compilation of DOL-Critical Specification into BIP Models

In this section, we show how to translate the DOL-Critical application (Sec. 5.1) and schedule (Sec. 5.2) specifications into components of the BIP language, and how to connect them with each other. The resulting BIP model is used for functional validation (by simulation) and code synthesis.

Fig. 7 gives a sneak-preview of the final model structure after compilation. The scheduler components are shown on the top and the application components on the bottom. The components are joined by BIP connectors, through which they can perform interactions with each other. The application components include the components dedicated to DOL-Critical tasks, denoted τ_1, τ_2, ..., their controllers, and data channels, denoted 'BlacBrd' and 'MailBx', for blackboard and mailbox, respectively. The scheduler components include one component for TTS Cycle, a set of components for TTS Frames, and Periodic Servers, which present each sporadic task to the scheduler by its periodic over-approximation. The scheduler components are connected to the tasks to coordinate their execution according to the schedule.

Example 7 To illustrate the complexity of the BIP model (number of components), we refer to the FMS application of Example 1. The compilation of the application from the DOL-Critical specification (see Table 1) results in 41 BIP automata components and 130 connectors, including specifically 8 components for tasks, 19 components to implement task controllers, and 14 components to implement data channels. In addition, the compilation of the respective TTS schedule specification (see Fig. 3) results in 20 BIP automata components and 92 connectors. Plugging the two sub-systems together results in a total of 61 components and 222 connectors.

In this section we describe the general procedure of compilation. First, Sec. 7.1 presents the commonly required properties of all BIP components. In Sec. 7.2 we present the scheduling components and in Sec. 7.3 the application components, respectively.

7.1 Required Properties of the Compiled Models

Provided that the DOL-Critical application and scheduling are correctly specified, the generated BIP models should by construction be: (i) *free from local deadlock* and (ii) *action-deterministic*.

Local deadlock is a situation where for a component (in the given global state of the system) no transitions are possible any more. Our BIP components are constructed in such a way that a local deadlock indicates that either the hardware resources cannot handle the activated real-time tasks on time or that the activation does not conform to specification. For example, in Fig. 5, component 'PeriodicA' is ready to execute an interaction at port 'StartA'

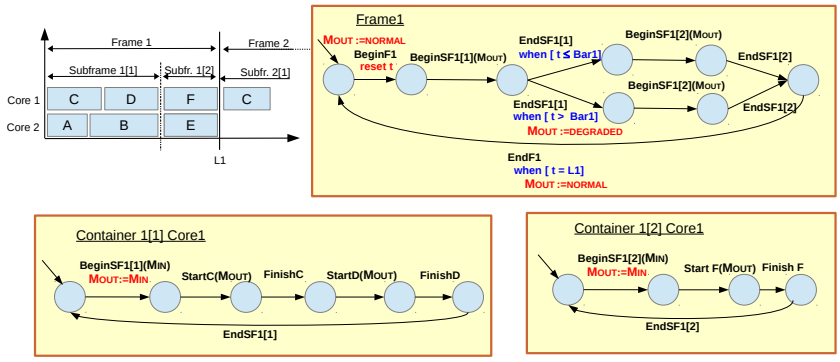


Fig. 8: TTS Scheduling Frames in BIP

only when $x = T_A$. If at this time instant both 'Thread' components are busy executing the previously started '(Task)' transitions, then component 'PeriodicA' will deadlock, as the clock x will continue increasing with time, never returning to the level T_A. To avoid a deadlock in 'PeriodicA', at least one of the 'Thread' components should be ready for interaction at periodic instances in time: $T_A, 2T_A, 3T_A, \ldots$. Certain components obtained by compilation from DOL-Critical have upper-bounded timing constraints, to encode a violation of the required timing properties by a local deadlock. Namely, the task controller components go into deadlock state if the tasks miss their deadlines or violate the required sporadic activation constraints. Most of such components are equipped with additional transitions that raise a runtime error in case of a local deadlock (not shown in the figures for ease of presentation). Note that absence of local deadlocks implies the absence of global system deadlocks.

Action determinism of a BIP model means that the model should never have to make a non-deterministic choice between two mutually-exclusive transitions (actions). The actions that can be taken at each given moment of time fully depend on the current state of the model. If a port is linked to two or more connectors, like in Fig. 5, then our model will enable only one of them at a time. The same holds for two outgoing transitions from the same location.

In the next two sections we present the BIP components generated at compilation and discuss how they satisfy these two properties.

7.2 Compiling the Scheduling Policy into BIP

First we show how the TTS scheduling policy (see Sec. 4.1) is implemented in BIP. For this, we use the example in Fig. 8. The figure shows a partial TTS schedule for an application with tasks denoted 'A', 'B', 'C', *etc.* Note that currently our compiler supports only two levels of criticality, though the models can be extended to more levels in a straightforward way. In dual-criticality systems, as in Fig. 8, every frame consists of two sub-frames.

Recall that '$barriers(f, \ell)_k$' denotes the maximal permitted length of the k-th sub-frame of frame f for the level-ℓ execution scenario. In our models, we use notation '$f[k]$' to denote the k-th sub-frame and 'L$\langle f \rangle$' (*i.e.,* L1, L2, ...) to denote the frame duration $\mathcal{L}_f$. We use 'Bar$\langle f \rangle$' to denote $barriers(f, 1)_1$. Depending on whether the actual runtime length of the first sub-frame respects this barrier or not, the tasks in the second sub-frame will run in normal or degraded mode (see Eq. 1). This is the main mixed-criticality runtime mechanism we aim to reflect in the generated BIP components.

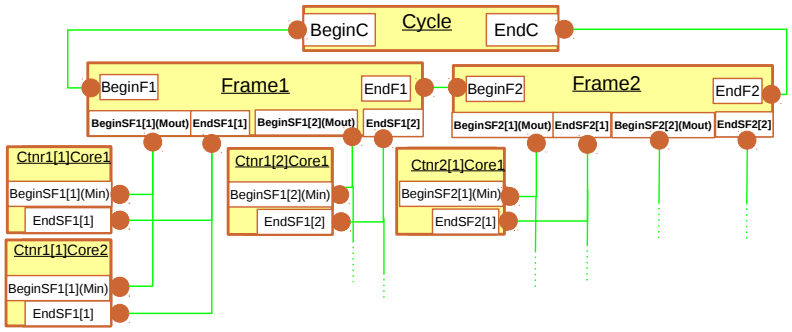


Fig. 9: Composing Cycle, Frames and Containers

To the right of the Gantt chart in Fig. 8, we show a (slightly simplified) general structure of the 'Frame$\langle f \rangle$' component, taking 'Frame1' as example. This component controls the mode 'M_{OUT}' of execution of the two sub-frames contained in the frame. Initially the mode is set to 'normal'. When frame f is about to start, interaction 'BeginF$\langle f \rangle$' ('begin frame f') gets enabled. At this point we reset clock t so that it measures the elapsed time in frame f. Then, we signal the begin of sub-frame $f[1]$ via interaction 'BeginSF$\langle f \rangle$[1]'. At the moment when the sub-frame finishes, the interaction 'EndSF$\langle f \rangle$[1]' gets enabled, and we check the elapsed time t. We keep the normal mode if t does not exceed barrier 'Bar$\langle f \rangle$', otherwise the mode is set to degraded. After executing the second sub-frame, the frame finishes, which is signalled via 'EndF$\langle f \rangle$'.

Examining this component, we conclude that it is characterized by action determinism, as the transition branching has mutually exclusive timing constraints. Also, it is free from local deadlock provided that the schedule is correct and the tasks scheduled in the frame finish their execution by time 'L$\langle f \rangle$'. Otherwise the component will be blocked forever at the origin of transition 'EndF$\langle f \rangle$'.

The two components given at the bottom of Fig. 8 are *Containers*, which are in charge of triggering jobs' execution according to the given TTS schedule. The container components are specific per sub-frame $f[k]$ and core. They trigger jobs according to the corresponding sequential schedule. In the figure, the left component implements the sequential schedule assigned to Frame 1, Sub-frame $[1]$ on Core 1, which executes first a job of task 'C' and then of task 'D'. Therefore, in this component we see a chain of transitions that start and finish these jobs. By convention, we use the notation 'Start_$\langle task_name \rangle$' for the job start interaction, and a similar notation for the job finish interaction. For synchronization with the frame component, the sequence of calls to the jobs is enwrapped in 'BeginSF/EndSF' interactions. At 'BeginSF', the frame component transmits the value of variable 'mode', which is passed through to the task components via the 'Start' interactions.

In Fig. 9 we show how frames and containers are connected to each other. There is a 'Cycle' component, which just executes a cyclic 'Begin/End' sequence. The 'begin' of a cycle triggers the execution of all frames in the cycle in the order of their index f, whereby we join the 'end' of frame f to the 'begin' of frame $f+1$. In the given example we assumed two frames per cycle. For every sub-frame the 'begin' and 'end' connectors join together all the containers for the specific sub-frame on Core 1, Core 2, ... Therefore, the employed 'barrier' mechanism to synchronize the cores at frame and sub-frame boundaries is a multi-party BIP interaction.

7.3 Compiling the Application into BIP

In [56] we give a detailed report on how we compile applications based on the FPPN model of computation (Fixed-priority Process Network [50]) into BIP. FPPN differs from the application model of DOL-Critical by employing a different mechanism for synchronisation among tasks. Also, it does not provide any support for mixed criticality. Nevertheless, we developed the compilation frameworks of DOL-Critical and FPPN together and ensured that several BIP models can be reused in both models of computation. Therefore, for some models we omit the details for brevity and address the readers to [56].

7.3.1 Compiling the Tasks

The BIP model of a DOL-Critical task is automatically extracted from its source code. For example, the code of the `square` task in Fig. 4 (Example 6) is compiled into the BIP automaton shown in Fig. 10(a). The local state variables of a DOL-Critical task become internal data variables of the BIP component. The initial transition implements the '$\langle task \rangle$_init()' subroutine. The rest of the task component implements the source code of the task's job, *i.e.,* the '$\langle task \rangle$_fire()' subroutine (DOL-Critical API). We enwrap the job execution between task start and task finish interactions ('Start/Finish_$\langle task \rangle$'). They are used both to enable the job executions upon their activation by the corresponding DOL-Critical controller and to delay them until the scheduled time by TTS containers (*e.g.,* Fig. 8).

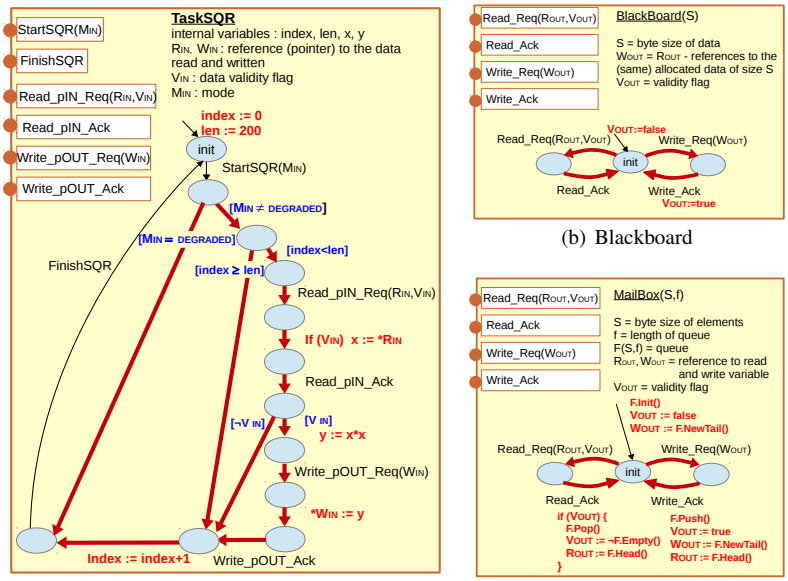
When translating the '$\langle task \rangle$_fire()' subroutine to a BIP model, the source code is parsed, searching for primitives that are relevant for the interactions between the task and the other components of the system. The relevant primitives are calls to 'DOLC_read()' and 'DOLC_write()' for reading/writing from/to the data channels. We see that the behavior of the resulting automaton is consistent with the behavior of the original source code, whereby the interaction primitives are replaced by patterns with interactions via BIP ports. As shown in Fig. 10(a), the pattern for 'DOLC_read()' and 'DOLC_write()' consists of three transitions: (i) request ('Req'), (ii) data-copying, and (iii) acknowledgement ('Ack').

Let us consider reading data for example. First, we have an interaction 'Read_$\langle port \rangle$_Req', which is an interaction requesting access to the channel via the DOL-Critical port '$port$'. In the corresponding interaction, the task receives from the data channel a reference 'R_{IN}' to the memory area from where it can read and a validity flag 'V_{IN}'. The next transition copies the data from the provided reference to the local variable to effectuate the data reading, and the third transition acknowledges the success of the read operation. Writing is performed in a similar way.

When compiled from a reasonable task source code (which, for safety-critical systems, should be confirmed by WCET analysis and software verification tools), the task components cannot introduce local deadlock or non-determistic behavior. By construction, the transitions have no explicit timing constraints and branches have mutually-exclusive data conditions. The transition actions are compiled from pieces of source code that should eventually terminate. All local-state variables should be always initialized to the same value and when a job execution starts from the same local state and reads the same data from the input data channels, it should produce the same data at the output channels.

7.3.2 Compiling the Data Channels

According to the task-to-channel connection topology specified in the XML files, BIP connectors are inserted between 'Read/Write_$\langle port \rangle$_Req/Ack' at the task and the 'Read-/Write_Req/Ack' ports at the data channel components.



(a) 'Square' task example compiled to BIP

(b) Blackboard

(c) Mailbox

Fig. 10: Compiling Tasks and Data Channels to BIP

Recall the DOL-Critical data channels introduced in Sec. 5.1. A basic notion of the supported data channels is the validity flag. The meaning of this flag is availability of data, given the non-blocking nature of read and write operations in DOL-Critical. A blackboard channel represents a shared variable and a mailbox is a queue buffer.

Fig. 10(b) shows the model for a blackboard. At the initial transition, we (implicitly) allocate a user-type variable of given byte size. Read (Write) operations are separated into request and acknowledge transitions, coherently to the task model of Fig. 10(a). During the request the blackboard communicates to the task the memory address, from (to) which it should read (write). In case of a read, the validity flag is communicated as well.

The BIP model of a mailbox is shown in Fig. 10(c). It is similar to blackboard, but instead of allocating a scalar user-type variable, the component initially creates a queue, i.e., a circular buffer, of user-type elements with a given capacity ('length'). Read (write) operations on a mailbox give the address of the tail (head) of the queue.

The branching between 'Read_Req' and 'Write_Req' shows a possibility of non-determinism in the case that the reader and writer tasks try to access the channel at the same time. However, in DOL-Critical we ensure functional determinism by setting dependencies between tasks that share a channel. This obliges the MCMSO optimizer to schedule their jobs in a sequential order in a sub-frame or in separate sub-frames, which excludes the possibility of non-deterministic interleaving of read and write interactions.

7.3.3 Compiling the Controllers

In DOL-Critical, exactly one task controller is instantiated per task, see Fig. 4. The two types of DOL-Critical task controllers – periodic and sporadic – are compiled into two corresponding types of BIP components. The details of these BIP models can be found in [56]. These

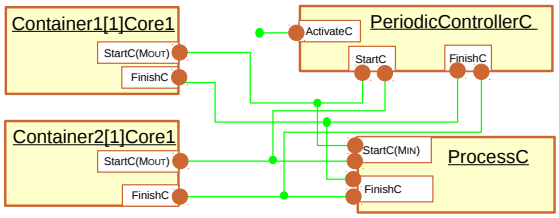


Fig. 11: Connection between a Periodic Task and its Containers

components are responsible to activate the task components according to their periodic or sporadic patterns, and to check their deadlines.

Note that the sporadic controllers in BIP are parametrized by a C subroutine of DOL-Critical, called *activation protocol*, where the user should implement the polling of system I/O peripherals to evaluate the conditions to activate the task. Next to the response time of task data processing (see Fig. 6), non-deterministic activation is another *environment-dependent* non-deterministic part of overall model behavior. Except for these two circumstances, the compiled BIP model is action-deterministic. We take this observation into account when discussing the system analysis in Sec. 8.3.

7.3.4 Connecting Application and Scheduler

Fig. 11 illustrates the BIP connections between the TTS scheduler and application components for the case of periodic tasks. In general, a task can be scheduled in multiple containers. In the running example, we assume that task 'C' is scheduled in two containers, as in the model of Fig. 8.

According to Fig. 11, in the case of a periodic task, the containers are linked to the 'Start_$\langle task \rangle$'and 'Finish_$\langle task \rangle$' connectors of the task directly, together with the periodic controller. For a sporadic task, such a connection can lead to local deadlock, as sporadic tasks are not regularly activated, whereas the TTS scheduler schedules them regularly. For this reason we insert a 'periodic server' component in between the scheduler and the sporadic task, which acts as a 'bridge' between them. For details on the periodic server, see [56].

Note that linking the task-component ports 'Start' and 'Finish' to multiple connectors indicates a possibility for action non-determinism. However, this is impossible by construction, because the containers connected to a task are active in different frames, and hence never at the same time.

8 Deployment on Target Architecture

In this section, we show how to use the BIP system model for automated code generation on a target platform, specifically the Kalray MPPA®-256. We also describe the feedback loop from the execution to DOL-Critical, which enables refined timing analysis and consideration of the runtime overheads for the optimized TTS schedule.

8.1 From BIP to Executable Code

Fig. 12 illustrates the deployment of the BIP system, using the same notations as in the running example of Fig. 7. We implemented our framework in a single shared-memory

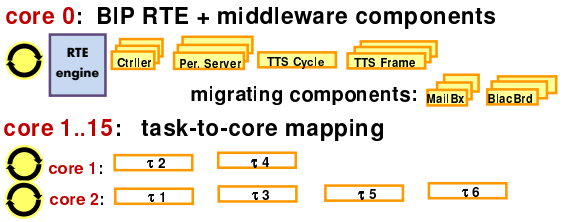


Fig. 12: BIP software model and its deployment on a multi-core system

cluster of the Kalray MPPA®-256 many-core platform. A cluster consists of 16 processing cores and 2MB of shared memory, and it can be programmed using the POSIX threads library, with at maximum one thread per core. Core 0 runs the default thread and Cores 1-15 can execute up to 15 additional threads created at runtime.

The BIP software model is translated into C++ and linked with the multi-threaded BIP runtime environment (RTE), which supports parallel execution of BIP components using POSIX threads, and whose original version was described in [63]. At the heart of this library lies a low-level scheduler that coordinates the interactions between the components, to which we refer as the BIP RTE engine. Our centralized RTE engine architecture simplifies the maintenance of the common notion of global physical time. In this work, substantial extensions to the BIP RTE were necessary for the support of real-time tasks, such as the support for self-timed transitions, the mapping of multiple BIP components to the same thread, as well as a restricted *migration* of components among different threads for enhanced parallelism.

As shown in Fig. 12, on top of the threads that run the tasks, the BIP RTE uses the default thread on Core 0 for the execution of the RTE engine. Our compiler also maps all the 'middleware' components to this thread, *i.e.,* all BIP components except the ones for the tasks. These are the task controllers, the scheduler components, and the data channels. The reason for separating the engine and the middleware from the tasks is the need to execute urgent instantaneous interactions for system control (*e.g.,* task activation, checking the deadline miss, starting a task) as timely as possible. The tasks execute the self-timed transitions for internal computations, and these transitions may take a significant time, up to the worst-case response time of the tasks. The urgent instantaneous interactions cannot wait until self-timed transitions finish, therefore the components that run these interactions are separated into an independent thread. At the same time, multiple tasks can be mapped to the same thread, according to the task-to-core mapping determined by the MCMSO tool. By construction, the tasks mapped to the same core will never try to concurrently obtain permission from the engine to execute on the core, as sequential execution of such tasks is orchestrated by the TTS scheduler components, whereas their timeliness should be ensured by the offline optimizer tool, namely the MCMSO.

An exception from the general rule of static mapping of components to threads is the support of a restricted component migration. Currently, this facility can be applied to the data-channel components, but not yet to tasks. We exploited migration to obtain improved system parallelism by letting the data-channel Read/Write interactions be executed entirely inside the threads of the tasks that perform reading and writing instead of executing them in the engine thread. This permits the tasks to read and write data in parallel, not interfering with each other and the engine.

8.2 BIP RTE Engine and Interaction Scheduling

The role of the BIP RTE engine is to trigger BIP interactions while ensuring their ordering and timing in accordance with the formal semantics of BIP. The components, which can be mapped on different cores (threads), have to notify the engine about the instantaneous interactions that they can potentially execute and wait until they are triggered by the engine [63]. Semantically, the instantaneous interactions should take zero time to execute, but in reality they require some non-zero time. Moreover, often multiple interactions must be triggered at the same time instance, *e.g.,* the 'activate' interactions for all periodic tasks always occur simultaneously at time zero and at the hyperperiod boundary. Since the interactions are triggered sequentially, there is always a certain 'response-time' interval between the time when the interactions should appear semantically and when they are triggered on the physical platform. The *interaction response time* thus includes the execution time of the given interaction and all semantically-simultaneous interactions triggered before it. Formally, the interaction response time represents the difference between the logical and physical values of the clock variables in the BIP model. Therefore it is referred to as 'clock drift' [2]. It corresponds to system timing inaccuracy and therefore should be bounded.

Note that the BIP engine is a simple pragmatic best-effort scheduler, which primarily seeks to ensure *semantically correct* ordering and close-to-correct timing, i.e., with as small clock drift as possible. The responsibility to ensure overall system-level timeliness is delegated to the BIP model itself. In the proposed design approach, it is the scheduler components which are responsible for this, and in our framework those are TTS scheduling components. The BIP engine does not distinguish the scheduler components from the rest. It just responds to the interaction notifications from all components according to their timing constraints.

In our BIP system models, we use instantaneous interactions for simple actions related to basic scheduling steps, *e.g.,* activation, start and finish of a task, beginning and end of a scheduling cycle or (sub-)frame, *etc.* For each instantaneous interaction, the engine determines the exact time instance when it should execute and tries to schedule it as accurately as possible. However, as explained earlier, the non-zero response times of such interactions, i.e., the clock drifts, lead to interaction-schedule inaccuracies that should be provably bounded by some margins. In terms of real-time system design, the clock drift is perceived as *runtime overhead*, which can be accounted for in the system schedulability analysis, by adding the estimated margins to the task execution profiles. This estimation is done via a feedback loop in our design flow, described in Section 8.3. The fact that in our case the executable scheduler model is formal also makes it simpler to express the problem of quantifying the runtime overhead margins in mathematical form.

In contrast to the instantaneous transitions, the self-timed transitions are intended not for carefully-timed 'control' steps, but for 'data processing' operations inside the tasks. Since their exact timing is unimportant, these transitions bypass the engine and get executed by different threads independently. The self-timed transitions are executed in a 'run-until-completion', as soon as possible manner. Unlike instantaneous actions, the execution time of those actions is considered to be *system workload* and not runtime overhead. Note that since in our task models all internal transitions and data-channel interactions are self-timed, there is no need to involve the RTE engine in scheduling any other interactions for a task between its 'Start' and 'Finish'.

The implementation of the RTE engine is based on the standard POSIX (*pthread*) library supported by the MPPA®-256 platform. The *master scheduler* in the thread of Core 0 consults the list of ready components and the *slave executors* in the threads of 'Core 1,2, *etc.*'

keep the lists of automata transitions that were designated for execution. The list of the master is extended by the slaves and the lists of the slaves are extended by the master. The lists are protected by mutex locks, and an empty list may result in a conditional wait. Adding elements to lists causes a notification by sending a signal to wake up possibly waiting threads. The BIP engine algorithm is described in [63].

8.3 Feedback Loop to DOL-Critical

To account for *runtime overheads* during schedulability analysis, we establish a feedback loop from the deployment to the timing analyzer of the MCMSO tool in DOL-Critical. As mentioned previously, the overheads correspond to BIP interactions from the task and scheduler components. In fact, the RTE engine represents a single point of interference among the concurrently executed BIP components, including the task components running on different cores. Namely, tasks contend for access to the RTE at runtime, with their interactions being served in a first-come first-serve, synchronous fashion. This type of interference is captured by our model of shared resources in Sec. 3.2. Therefore, we can model the BIP interactions as accesses to a shared resource, the RTE engine, in a similar way as we model interfering accesses to a shared-memory bus. For this purpose, we include the minimum/maximum issued interactions from the BIP model to the RTE engine in the tasks' execution profiles, and bound the engine access time T_{acc} by applying extensive measurements or static WCET analysis on the source code of the engine. It is worth mentioning that there exists a connection between the two types of shared resources, i.e., the memory bus and the RTE engine, although in the present work we focus on the latter. That is, at runtime each synchronization with the RTE engine triggers a burst of accesses to the shared memory, as inter-thread synchronization is in general accompanied by cache flushing on the MPPA®.

Furthermore, there are RTE engine accesses that cannot be attributed to a particular task, a significant number of which originate from the runtime resource management mechanisms. For instance, take the barrier-synchronisation interaction at the end of each TTS sub-frame or the interactions at the beginning of each scheduling cycle. Such overheads can be modeled as engine accesses issued from additional *synchronization* tasks. These overheads become known only when the complete system executable is generated and linked with the RTE engine. We evaluate and annotate these overheads at the feedback loop of our design flow. Afterwards, the flow is re-iterated, first by evaluating whether the previously obtained scheduling solution is still feasible. To this end, the timing analyzer of the MCMSO tool repeats the analysis for the implemented TTS schedule, by considering the additional timing interference on the shared RTE engine. If the timing analysis shows that the TTS schedule is infeasible, then new optimization, compilation, and code generation rounds are required.

The DOL-Critical application back-annotation with task execution profiles, including the number of RTE engine accesses, and synchronization tasks is currently performed manually in order to capture accurately all identified and measured runtime overheads. To bound the RTE engine access counts, we exploit the property of action-determinism of our BIP model, which implies that different engine access sequences may result either from different task execution times or from different sporadic-task activations. Therefore we (i) *identify all alternative scenarios* in terms of execution times and sporadic protocol and (ii) *simulate them*, while counting the engine accesses. For this, we exploit the observations that these scenarios are orthogonal, that the runtime variability is covered by the level-ℓ execution scenarios of the TTS sub-frames, and that the sporadic task activation can be characterized by maximal activation counts in different TTS frames. In future work, we intend to formalize

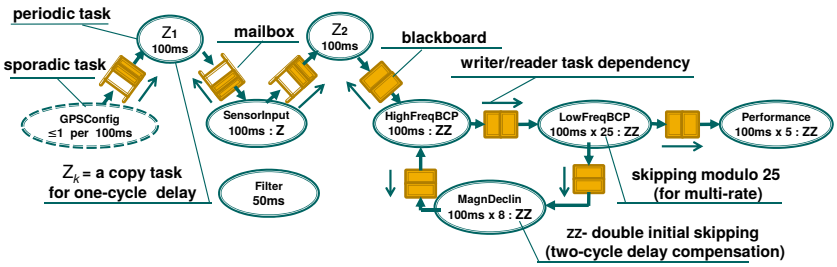


Fig. 13: Flight Management System (FMS) test case

and automate this analytical reasoning and to establish a formal refinement relation between high-level customized timing analysis in DOL-Critical and detailed BIP implementation models, to ensure provably safe estimation of the worst-case runtime overheads. We also intend to study further the connection between interference on multiple shared resources, e.g., the RTE engine and the shared-memory bus.

9 Case-Study

To demonstrate the applicability of the complete DOL-BIP-Critical design flow, we employ an industrial representative implementation of a flight management system (FMS) [19], which was already introduced in Example 1 – Table 1 and Example 4 – Fig. 3. We model the application (Sec. 9.1) and then, step-by-step, we show how our flow finds an optimal TTS schedule on a cluster of the MPPA®-256 platform (Sec. 9.2), how it synthesizes code, executes it, and integrates the runtime overheads (including TTS synchronization overhead) into the final schedule optimization process (Sec. 9.3).

9.1 Flight Management System Specification

The FMS is a safety-critical embedded avionics system, responsible for aircraft localization, flightplan computation for the auto-pilot, detection of the nearest airport, *etc.* In this experiment we look into a sub-system of the FMS. Fig. 13 shows the corresponding DOL-Critical application, which is responsible for calculating the best computed position (BCP) and predicting the performance (*e.g.,* fuel usage) of the airplane, based on periodically collected sensor data and sporadic configuration commands from the pilot, *e.g.,* for configuring the Global Positioning System (GPS). Specifically, after being pre-processed by task 'SensorInput', the sensor data are processed by task 'HighFreqBCP'. Then, they arrive at task 'LowFreqBCP', which post-processes the data at low frequency, and makes them available to other sub-systems of the FMS. 'LowFreqBCP' also provides the results to a feedback loop that takes into account the magnetic declination for computing the airplane position.

All depicted tasks are periodic except for the sporadic task 'GPSConfig', which can execute at most once in any 100-ms interval. All periodic tasks of the FMS are specified with period 100 ms. However, some of them contain in their C code a wrapper to skip the processing at all but every n-th job, to represent tasks with original period $n \cdot 100$ ms. This is done for three reasons: (i) to reduce the effective hyperperiod $\mathcal{H}$, (ii) to ensure deterministic communication, and (iii) to comply with the DOL-Critical specification requirement for equal period among tasks with dependencies. Note that keeping the original $\mathcal{H}$ (in the FMS case, equal to 40 seconds) would result in generating hundreds of TTS frame and

container components in BIP, which would lead to infeasible memory requirements for the implementation on a single MPPA®-256 cluster.

The given task structure originally allowed only a limited two-task parallelism, which consisted in the task-dependency branching from 'LowFreqBCP' to 'MagnDeclin' and 'Performance'. To introduce pipelining parallelism, we inserted two new tasks, denoted as Z_1 and Z_2. These tasks copy input data to the output, thus ensuring double-buffering, which is required for pipelining. Because each inserted Z_k task leads to an additional data-propagation delay of one period, this delay is subtracted from the deadlines of the tasks that follow in the task chain, which, therefore, should be sufficiently large. The wrappers inside these tasks should skip one initial task-code execution to 'compensate' a delay in each Z_k task that precedes in the task chain.

All tasks of the FMS sub-system are used to calculate critical information, i.e., the current position of the airplane. Therefore, they are certified at safety level DAL-B according to the DO-178C standard [17]. We map this safety level to criticality level 2 ('high') in our system model. The execution profiles of the tasks are shown in Table 1 in Sec. 3.1. The tasks are protected from exceptional execution times overruns (due to potential faults and fault correction) by defining a significantly more pessimistic execution profile at level 2 than at level 1. Not having WCET tools for the MPPA®-256 platform at our disposal, we derived level-1 worst-case execution times based on extensive measurements. For the level-2 estimates, we augmented the level-1 bounds by a margin of 10 up to 25 ms, which also makes them at least 10x larger. We introduced a possibility to simulate fault injection, by programming an optional prolongation of the task execution by up to the level-2 execution time through an additional dummy loop in the C code.

Table 1 includes also the bounds on *RTE engine accesses* for each task. We do not distinguish between level-1 and level-2 in this case, as they turned out to be the same. Recall from Sec. 8.3 that RTE accesses correspond to BIP interactions, and their bounds are obtained by manual analysis of the interactions from the respective task automata in the BIP model. Before the optimized scheduling solution is generated, one can analyze only the components for application tasks and their controllers. For the periodic tasks, we observe that their execution causes always exactly three interactions: Start, Finish and deadline check (the latter is done in fact in the controller). Sporadic tasks cause one extra interaction, which is related to the activation protocol. Note that when counting BIP interactions, we neglect self-timed interactions, as they do not lead to RTE engine accesses.

Table 1 includes also three *synchronization tasks*, whose parameters become available only at the second iteration of the design flow, after the scheduler components get synthesized. Note that the synchronization tasks account not only for the TTS components themselves, such as cycle, frames, and containers, but also for other components that cause BIP interactions at the boundaries of the cycle, frame, and sub-frame, respectively. For example, at the beginning of each cycle all eight periodic tasks get activated by task controllers, which explains the high access count of the synchronization task 'Cycle_Begin'.

Through extensive measurements on the MPPA®-256 platform (again, due to non-availability of suitable WCET tools), we derived a (pessimistic) upper bound on the BIP RTE-engine delay per interaction, which amounts to $T_{acc} = 0.42$ ms. We believe that this bound captures the cost not only of accessing the RTE engine, but also of the subsequent accesses to the shared cluster memory, as the measurements included also the impact of data cache flushing at the inter-core synchronization points, where the tasks start and finish their execution. However, for the design of a real-world safety-critical system, such an assumption would need to be further investigated and proven, *e.g.,* through static analysis.

		1st Iteration	2nd Iteration	Empirical
Frame f_1, Sub-frame 1 (DAL-B)	$barriers(f_1,1)_1$	7.46	13.34	8
	$barriers(f_1,2)_1$	29.78	35.66	27
Frame f_1, Sub-frame 2 (DAL-C)	$barriers(f_1,1)_2$	33.26	34.1	34
	$barriers(f_1,2)_2$	3.26	4.1	4
Frame f_2, Sub-frame 1 (DAL-B)	$barriers(f_2,1)_1$	6.04	7.72	6
	$barriers(f_2,2)_1$	31.04	32.72	28
Frame f_2, Sub-frame 2 (DAL-C)	$barriers(f_2,1)_2$	33.26	34.1	34
	$barriers(f_2,2)_2$	3.26	4.1	4

Table 2: Estimated function $barriers$ before vs. after feedback look vs. empirical results

Finally, since the considered sub-system of FMS includes only tasks of criticality level DAL-B (level 2), to obtain a dual-critical application we added an artificial periodic task called 'Filter', with period 50 ms. This task models some digital signal processing functionality, considered as a less critical DAL-C (level 1) task. Since 'Filter' is low-criticality, we model two execution modes: *normal* and *degraded*. Specifically, 'Filter' executes a loop resembling a digital filter, the number of loop iterations being significantly lower in degraded mode, to represent the possibility of providing a reduced level of quality with a smaller number of digital filter coefficients.

9.2 Scheduling and Mapping Optimization

For the FMS sub-system, the maximal degree of parallelism is four (three pipeline stages and one branching). Therefore, we choose to allocate a subset of five MPPA®-256 cores: four for task execution and one for the BIP RTE engine. For the mapping and scheduling optimization, we provide the DOL-Critical specifications of the FMS sub-system and the 5-core subset of the MPPA®-256 cluster to the MCMSO optimizer, which performs design space exploration to optimize the mapping of tasks to cores and the scheduling of the tasks on each core based on the TTS scheduling policy (Sec. 4.1). The optimization goal (Sec. 4.2) is to maximize the slack interval at the end of the frames, while respecting the task dependencies and accounting for the interference of concurrent task accesses to the RTE engine as a shared resource. In this case, the TTS scheduling cycle has a period of 100 ms (equal to the hyper-period of the tasks) and it is divided into two frames, each with a fixed length of 50 ms. MCMSO produced the mapping and scheduling solution which is illustrated in Fig. 3 after 342 ms of exploration. It converged to this solution after having checked 20,548 alternatives. Note that the workload distribution among the cores is fairly balanced, which is due to the cost function that is used to guide the optimization procedure (Eq. 3, Sec. 4.2).

The worst-case sub-frame lengths for the level-1 and level-2 execution scenarios, as computed by the timing analyzer of the MCMSO tool, are presented in Table 2 (Column '1st Iteration'). The analyzer implements the approach of [25] for taking into account the interference on the shared resource. Based on the obtained sub-frame lengths and the condition of Eq. 2, it follows that the TTS schedule of Fig. 3 is *feasible*. Namely, the last sub-frames finish before the end of the containing frames under all execution scenarios, which implies that all tasks receive enough resources to finish before their deadlines according to the respective execution profiles.

9.3 FMS Deployment and Feedback Loop

The optimized TTS schedule for the FMS sub-system, along with the application specification, are compiled into BIP automata, as described in Sec. 7. Functional correctness is validated through simulation, and code is automatically synthesized for the deployment on the MPPA®-256 platform (subset of 5 cores within a cluster). Fig. 15 presents Gantt charts of the FMS execution traces on the MPPA®-256 for three alternative scenarios. Each chart depicts six consecutive TTS scheduling cycles.

'Level-1' and 'Level-2' scenarios represent corner-cases for timing analysis, where all tasks execute without skipping (which happens on the hyper-period boundaries) and according to their maximal profile at the given level. In this case, the actual sub-frame lengths can potentially approach the worst-case $barriers$ values at the given level. The 'ordinary' scenario represents a possible execution of the system, where periodic tasks skip some periods due to pipelining and original periods, and the sporadic task is activated by some arbitrarily chosen (encoded in DOL-Critical) protocol. In this scenario, we simulated some fault injections in tasks 'Z1', 'Z2', 'HighFreqBCP', and 'SensorIn' in the fifth scheduling cycle (between 400 and 500 ms). Note that the tasks take considerably longer to execute in this cycle, with their execution time being close to their level-2 profile in Table 1. This triggers a level-2 execution scenario, which results in providing degraded service to the lower-criticality 'Filter' task in both frames of this cycle. In degraded mode, 'Filter' runs for approximately 2 ms instead of the usual 32 ms.

The empirical worst-case sub-frame lengths of the TTS schedule, as measured over long execution intervals, are depicted in the last column of Table 2. Note that they actually surpass the respective analytically-derived bounds obtained at the first iteration. This is because several BIP interactions, resp. accesses to the BIP RTE engine, which take place at the beginning of each TTS frame, upon barrier synchronisation, and at hyper-period boundaries, have not been considered in timing analysis. To capture these overheads, we model the additional synchronization tasks 'Frame_Begin', 'Subframe_Bar', and 'Cycle_Begin' with the worst-case RTE access bounds of Table 1. After back-annotating the DOL-Critical application and schedule specifications, the timing analyzer re-evaluates function $barriers$, as depicted in column '2nd Iteration' of Table 2. As expected, the new analytic worst-case sub-frame lengths bound safely the empirical values. Also, according to these bounds, the TTS schedule remains feasible also after accounting for the runtime overheads, therefore the design process has terminated successfully.

Fig. 14 illustrates the worst-case finish time of the last sub-frame in each TTS frame for level-1 and level-2 execution scenarios, as derived by the MCMSO analyzer *before* and *after* the feedback loop, as well as the *empirical* worst-case bound. The last bar is fixed to 50 ms to indicate the end of the respective frame. Note that the empirical worst-case scenario is always bounded by the analytic results of the second MCMSO iteration, unlike the respective results of the first iteration. This clearly confirms the necessity for the feedback loop in our design flow. The analytic worst-case finish times increase up to 20.3% (frame 1, level-2) after the feedback, indicating the non-negligible cost of runtime overheads and the absolute need to consider its effect on schedulability.

In summary, the deployment of the FMS sub-system on the MPPA®-256 validates the applicability of our design flow for the implementation of mixed-criticality systems on commercial multi-core architectures. Temporal isolation is preserved, since tasks of different criticality never overlap and lower-criticality tasks do not interfere with the execution of higher-criticality tasks. Incremental design is enabled, since there is a bounded slack interval at the end of each frame (see the difference between analytic bounds and frame length

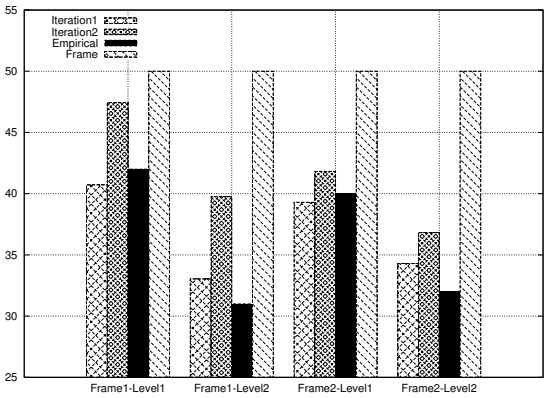


Fig. 14: Worst-case finish time [ms] of last sub-frame in each TTS frame as computed at 'Iteration 1', 'Iteration 2', and empirically

in Fig. 14 and idle intervals in the Gantt charts). This slack can be used to host new lower-criticality tasks if they are added later to the system. Task dependencies are respected, while task execution and communication are performed deterministically, as dictated by the BIP models. Additionally, the MCMSO was able to find a feasible (optimized for incremental design) TTS schedule and bound safely the tasks' worst-case response times even in the presence of non-negligible runtime overheads. Based on this first evidence, we are convinced that the DOL-BIP-Critical design flow can be a viable solution for the rigorous design of mixed-criticality systems, with potential to be applied to complex industrial-scale settings.

10 Conclusion

In this paper, we presented a complete design flow for the efficient and correct-by-construction deployment of mixed-criticality applications on multicores. The design flow enables the specification of complex reactive mixed-criticality applications and determines a mapping and schedule of the application on multicores, such that temporal isolation among different criticality levels is preserved even in the presence of shared resources, and incremental design is enabled. The run-time mechanisms that ensure these mixed-criticality properties are naturally represented in timed-automata models and all software components are compiled from a high-level description language into a network of task automata in BIP language. Code is generated automatically for execution on the target platform. Prototypes of all developed tools are available online and their use has been demonstrated through an industrial-scale avionics application, which is deployed on the cutting-edge Kalray MPPA®-256 platform. As future work, we aim to evaluate our design flow with additional realistic applications, and to improve the design of the BIP RTE in order to reduce its runtime overhead and improve its applicability to high-integrity systems. Moreover, we intend to investigate further the feedback loop of the design flow, by proving formal refinement relations between the automata-based implementation and high-level models, in order to safely account for the runtime overhead in schedulability analysis already at system level.

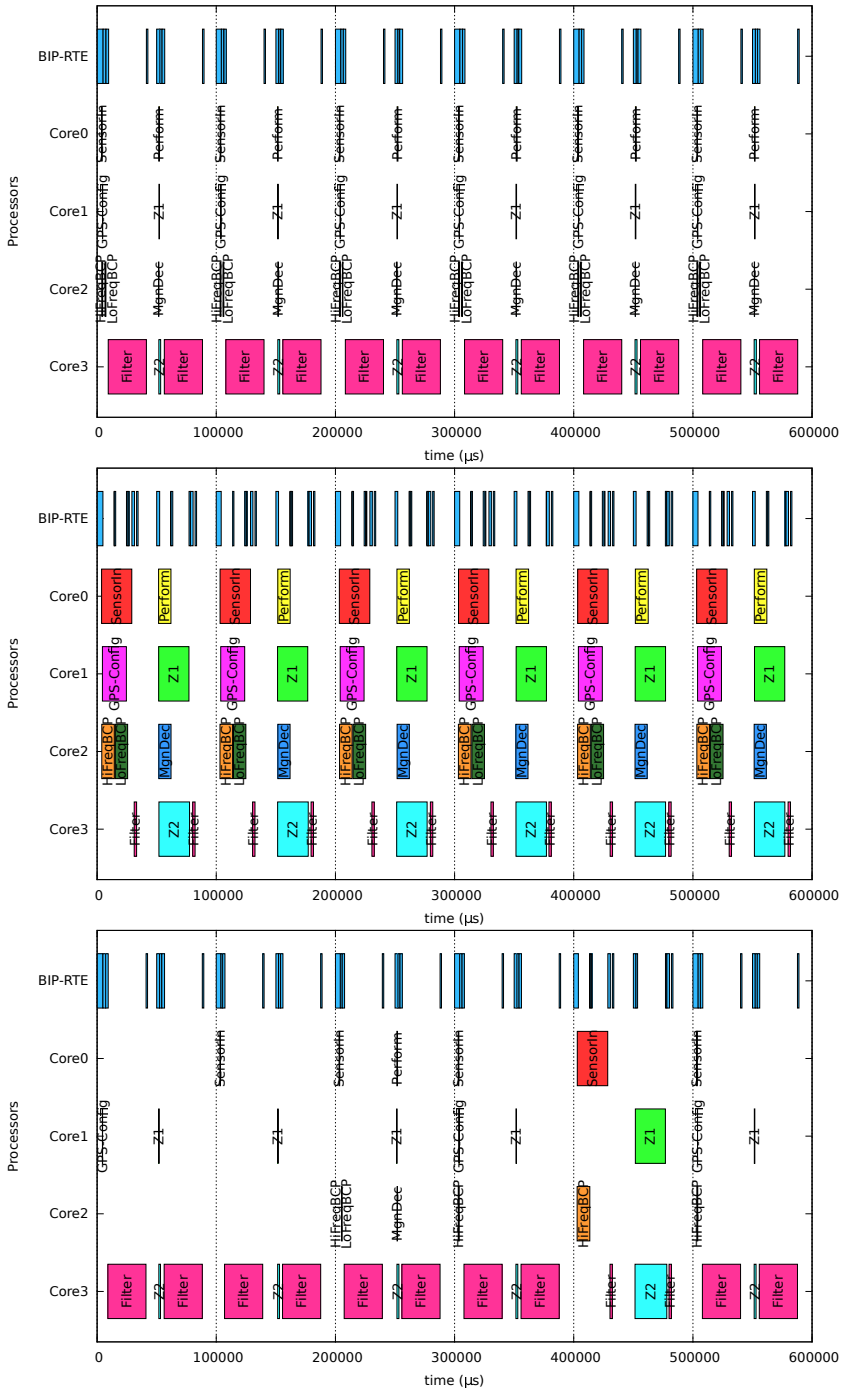


Fig. 15: FMS test case: ‘Level-1’,‘Level-2’, and ‘Ordinary’ traces on MPPA®-256

References

1. ISO 26262, Road Vehicles - Functional Safety, 2011.
2. T. Abdellatif, J. Combaz, and J. Sifakis. Model-based implementation of real-time applications. In *EMSOFT '10*, 2010.
3. AbsInt. aiT Worst-Case Execution Time Analyzers, 2015.
4. R. Alur and D. L. Dill. Automata For Modeling Real-Time Systems. In M. Paterson, editor, *Proc. of the 17th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 443 of *LNCS*, pages 322–335. Springer, 1990.
5. T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES — A Tool for Modelling and Implementation of Embedded Systems. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, pages 460–464. Springer, 2002.
6. J. Anderson, S. Baruah, and B. Brandenburg. Multicore operating-system support for mixed criticality. In *Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, 2009.
7. ARINC. ARINC 653-1 Avionics application software standard interface. Technical report.
8. J. Barhorst, T. Belote, P. Binns, J. Hoffman, J. Paunicka, P. Sarathy, J. Stanfill, D. Stuart, and R. Urzi. White Paper: A Research Agenda for Mixed-Criticality Systems. April 2009.
9. S. Baruah, B. Chattopadhyay, H. Li, and I. Shin. Mixed-criticality scheduling on multiprocessors. *Real-Time Systems*, 50:142–177, 2014.
10. P. Bourgos, A. Basu, M. Bozga, S. Bensalem, J. Sifakis, and K. Huang. Rigorous system level modeling and analysis of mixed HW/SW systems. In *Proc. Int. Conf. Formal Methods and Models for Codesign, MEMOCODE 2011*, pages 11–20, 2011.
11. A. Burns and S. Baruah. Towards a more practical model for mixed criticality systems. *Workshop on Mixed Criticality*, pages 1–6, 2013.
12. A. Burns and R. Davis. Mixed criticality systems: A review. 2015.
13. A. Burns, T. Fleming, and S. Baruah. Cyclic Executives, Multi-core Platforms and Mixed Criticality Applications. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3–12, 2015.
14. J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS^{RT} : A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *RTSS*, pages 111–126, 2006.
15. B. D. de Dinechin, D. van Amstel, M. Poulhiès, and G. Lager. Time-critical Computing on a Single-chip Massively Parallel Processor. In *DATE'14*. EDAA, 2014.
16. D. de Niz and L. T. X. Phan. Partitioned scheduling of multi-modal mixed-criticality real-time systems on multiprocessor platforms. In *RTAS*, pages 111–122, 2014.
17. DO-178C. RTCA/DO-178C, Software Considerations in Airborne Systems and Equipment Certification, 2012.
18. DOL-Critical. Distributed Operation Layer for Mixed-Criticality Applications. `http://www.tik.ee.ethz.ch/~certainty/dolc.html`, 2014.
19. G. Durrieu, M. Faugère, S. Girbal, D. G. Pérez, C. Pagetti, and W. Puffitsch. Predictable Flight Management System Implementation on a Multicore Processor. In *ERTSS'14*, 2014.
20. A. Easwaran. Demand-based scheduling of mixed-criticality sporadic tasks on one processor. In *RTSS'13*, 2013.
21. P. Ekberg and W. Yi. Bounding and Shaping the Demand of Mixed-Criticality Sporadic Tasks. In *ECRTS'12*, 2012.
22. E. Fersman, P. Krcál, P. Pettersson, and W. Y. 0001. Task automata: Schedulability, decidability and undecidability. *Inf. Comput.*, 205(8):1149–1172, 2007.
23. J. Flodin, K. Lampka, and W. Yi. Dynamic budgeting for settling DRAM contention of co-running hard and soft real-time tasks. In *Industrial Embedded Systems (SIES), 2014 9th IEEE International Symposium on*, pages 151–159, June 2014.
24. G. Giannopoulou, K. Lampka, N. Stoimenov, and L. Thiele. Timed model checking with abstractions: towards worst-case response time analysis in resource-sharing manycore systems. In *EMSOFT'12*, 2012.
25. G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele. Scheduling of Mixed-Criticality Applications on Resource-Sharing Multicore Systems. In *EMSOFT'13*, 2013.
26. G. Giannopoulou, N. Stoimenov, P. Huang, L. Thiele, and B. de Dinechin. Mixed-criticality scheduling on cluster-based manycores with shared communication and storage resources. *Real-Time Systems*, May 2015.
27. S. Goossens, B. Akesson, and K. Goossens. Conservative Open-page Policy for Mixed Time-criticality Memory Controllers. In *DATE'13*, 2013.
28. A. Hansson, K. Goossens, M. Bekooij, and J. Huisken. CompSoC: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14(1):2, 2009.

29. M. Hassan, H. Patel, and R. Pellizzoni. A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems. In *RTAS*, pages 307–316, 2015.
30. J. Herman, C. Kenna, M. Mollison, J. Anderson, and D. Johnson. RTOS Support for Multicore Mixed-Criticality Systems. In *RTAS*, pages 197–208, 2012.
31. H.-M. Huang, C. Gill, and C. Lu. Implementation and Evaluation of Mixed-criticality Scheduling Approaches for Sporadic Tasks. *ACM Trans. Embedded Computing Systems*, 13(4s):126:1–126:25, July 2014.
32. K. Huang, W. Haid, I. Bacivarov, M. Keller, and L. Thiele. Embedding formal performance analysis into the design cycle of MPSoCs for real-time streaming applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 11(1):8, 2012.
33. P. Huang, G. Giannopoulou, R. Ahmed, D. B. Bartolini, and L. Thiele. An Isolation Scheduling Model for Multicores. In *RTSS*, San Antonio, TX, USA, Dec 2015.
34. P. Huang, G. Giannopoulou, N. Stoimenov, and L. Thiele. Service Adaptions for Mixed-Criticality Systems. In *ASP-DAC'14*, 2014.
35. G. Kahn. The semantics of a simple language for parallel programming. In *Proc. IFIP Congress on Information Processing*, volume 74, pages 471–475, 1974.
36. B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In *Intl. Coference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 338–349, 1997.
37. N. Kim, B. C. Ward, M. Chisholm, C. Y. Fu, et al. Attacking the One-Out-Of-m Multicore Problem by Combining Hardware Management with Mixed-Criticality Provisioning. In *RTAS*, 2016.
38. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.
39. O. Kotaba, J. Nowotsch, M. Paulitsch, S. M. Petters, and H. Theiling. Multicore in real-time systems–temporal isolation challenges due to shared resources. In *Workshop on Industry-Driven Approaches for Cost-effective Certification of Safety-Critical, Mixed-Criticality Systems*, 2014.
40. J. Lee, K.-M. Phan, X. Gu, J. Lee, A. Easwaran, I. Shin, and I. Lee. MC-Fluid: Fluid Model-Based Mixed-Criticality Scheduling on Multiprocessors. In *RTSS*, pages 41–52, 2014.
41. H. Li and S. Baruah. Load-based Schedulability Analysis of Certifiable Mixed-criticality Systems. In *Intern. Conf. on Embedded Software*, EMSOFT'10, 2010.
42. D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit. Platform 2012, a Many-core Computing Accelerator for Embedded SoCs: Performance Evaluation of Visual Analytics Applications. In *DAC'12*, 2012.
43. R. G. Michael and S. J. David. Computers and intractability: a guide to the theory of NP-completeness. *WH Freeman & Co., San Francisco*, 1979.
44. M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos. Mixed-Criticality Real-Time Scheduling for Multicore Systems. In *Int. Conf. Computer and Information Technology*, CIT'10, pages 1864–1871. IEEE, 2010.
45. M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero. Hardware Support for WCET Analysis of Hard Real-time Multicore Systems. In *ISCA*, pages 57–68, 2009.
46. R. Pathan. Schedulability Analysis of Mixed-Criticality Systems on Multiprocessors. In *ECRTS'12*, 2012.
47. R. Pellizzoni, B. D. Bui, M. Caccamo, and L. Sha. Coscheduling of CPU and I/O Transactions in COTS-Based Embedded Systems. In *RTSS'08*, 2008.
48. M. Perrotin, E. Conquet, P. Dissaux, T. Tsiodras, and J. Hugues. The TASTE Toolset: turning human designed heterogeneous systems into computer built homogeneous software. In *Proc. Embedded Real-time Software and Systems Conference*, 2010.
49. P. Poplavko, P. Bourgos, D. Socci, S. Bensalem, and M. Bozga. Multicore Code Generation for Time-critical Applications (Tool), http://www-verimag.imag.fr/Multicore-Time-Critical-Code,470.html, 2015.
50. P. Poplavko, D. Socci, P. Bourgos, S. Bensalem, and M. Bozga. Models for Deterministic Execution of Real-time Multiprocessor Applications. In *DATE*, 2015.
51. J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 99–108, 2011.
52. F. Santy, L. George, P. Thierry, and J. Goossens. Relaxing mixed-criticality scheduling strictness for task sets scheduled with FP. In *ECRTS*, pages 155–165. IEEE, 2012.
53. L. Sha, M. Caccamo, R. Mancuso, J.-E. Kim, M.-K. Yoon, R. Pellizzoni, H. Yun, et al. Single Core Equivalent Virtual Machines for Hard Real-Time Computing on Multicore Processors. Technical report, University of Illinois at Urbana-Champaign, November 2014.
54. L. Sigrist, G. Giannopoulou, P. Huang, A. Gomez, and L. Thiele. Mixed-Criticality Runtime Mechanisms and Evaluation on Multicores. In *RTAS'15*, 2015.

55. D. Socci, P. Poplavko, S. Bensalem, and M. Bozga. Modeling Mixed-critical Systems in Real-time BIP. In *ReTiMiCs'2013*, 2013.
56. D. Socci, P. Poplavko, P. Bourgos, S. Bensalem, and M. Bozga. A Timed-automata based Middleware for Time-critical Multicore Applications . (Extended version of SEUS'15 workshop paper). Report TR-2015-12, Verimag, 2015.
57. S. Sriram and S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization, Second Edition*. Signal Processing and Communications. Taylor & Francis, 2009.
58. H. Su and D. Zhu. An elastic mixed-criticality task model and its scheduling algorithm. In *DATE*, pages 147–152, 2013.
59. D. Tamas-Selicean and P. Pop. Design Optimization of Mixed-Criticality Real-Time Applications on Cost-Constrained Partitioned Architectures. In *RTSS'11*, 2011.
60. L. Thiele, I. Bacivarov, W. Haid, and K. Huang. Mapping Applications to Tiled Multiprocessor Embedded Systems. In *ACSD'07*, 2007.
61. L. Thiele, S. Chakraborty, and M. Naedele. Real-time Calculus for Scheduling Hard Real-Time Systems. In *ISCAS*, 2000.
62. S. Tobuschat, P. Axer, R. Ernst, and J. Diemer. IDAMC: A NoC for mixed criticality systems. In *RTCSA*, pages 149–156, 2013.
63. A. Triki, J. Combaz, S. Bensalem, and J. Sifakis. Model-based implementation of parallel real-time systems. In *FASE'13*. Springer, 2013.
64. S. Vestal. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. In *RTSS'07*, 2007.
65. M. T. B. Waez, J. Dingel, and K. Rudie. A survey of timed automata for the development of real-time systems. *Computer Science Review*, 9:1–26, 2013.
66. R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966 –978, 2009.
67. Z. P. Wu, Y. Krish, and R. Pellizzoni. Worst Case Analysis of DRAM Latency in Multi-requestor Systems. In *RTSS*, pages 372–383, Dec 2013.
68. G. Yan, X. Zhu, R. Yan, and G. Li. Formal Throughput and Response Time Analysis of MARTE Models. In *Proc. Formal Methods and Software Engineering*, pages 430–445, 2014.
69. H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 155–166, 2014.
70. H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory Access Control in Multiprocessor for Real-Time Systems with Mixed Criticality. In *ECRTS'12*, 2012.