Safe Implementation of Mixed-Criticality Applications in Multicore Platforms: A Model-Based Design Approach

Pasquale Antonante[1], Juan Valverde-Alcalá[1], Stylianos Basagiannis[1], and Marco Di Natale[2]

[1] United Technologies Research Center, Cork, Ireland
{AntonaP,ValverJ,BasagiS}@utrc.utc.com
[2] Scuola Superiore SantAnna, Pisa, Italy
marco.dinatale@sssup.it

Abstract. Application complexity in safety-critical systems is currently creating an immediate need to employ new model-based approaches to ensure system's safe operation in high performances. At the same time, hardware evolution through multicore and hybrid architectures, while serving performance requirements, has not been realized as a safe and technology-ready solution to be employed in critical domains. In this paper, we report our experiences on the development of a model-based design workflow for safety assurance in mixed-critical applications executed on multicore platforms. Starting from our application specification, we develop intermediate models and extract configuration parameters that help us define a task optimization problem. Tasks composing the application will be weighted according to their criticality degree, allowing us to solve an optimization problem for safe resource and time partitioning at the available multicore resources. Based on code-generation techniques, we automatically generate an optimal and safe schema to be implemented in a real-time operating system, safeguarding the multicore resources from errors while executing the tasks. Indicative results are being presented by a prototype tool developed for a case study while we reason about the applicability of the approach.

Keywords: Aerospace, model-based design, multicore, safety

1 Introduction

In multiple domains such as automotive and telecommunications, embedded systems tend to be governed by multicore platforms [1]. Meanwhile, safety critical systems are composing highly complex networks of interconnected Cyber-Physical Systems (CPS) that question their performance and safety [2]. Consequently, the need for increased computational power on embedded systems' platforms raise questions on the safe usage of multicore against certification regulations. Multicore CPS are still not fully embraced by industry for safety-critical applications. For example, aerospace systems are subject to costly and time-consuming certification processes, which require a predictable behavior under certain hazardous conditions, hard to be proved in multicore platforms. The main reason is reflected on the multicore non-determinism where commercial-off-the-shelf (COTS) solutions are based on Real-time Operating Systems [3].

At the same time, model-based design processes have been acknowledged nowadays as a vital, irreplaceable process of product development in multiple industrial domains including automotive and aerospace. Despite the significant advances achieved in modeling, simulation and verification, there is still a lack of methods and integrated frameworks enabling multicore tasks safe and optimal scheduling and system-wide validation. At the component level, product development for software is driven by DO-178C [18] and for hardware by DO-254 [19]. Re-defining a multicore model-based design approach towards avionics regulations, we target on the DO-178C supplement (DO-331 [21]), where models have to generate proofs of validity during their code generation.

The challenge in this paper is to employ current model-based design techniques and tools used today in the industry and evaluate their effectiveness in determining the correctness of new prototypes. Those models will include every component relevant to the overall system behavior, such as algorithms, control logic, and device analytics. Recent studies [9] have shown that the application of model-based certification and formal verification can be a practical and cost-effective solution against regulations requirements [12]. Examples of available model-based commercial tools are Simulink® [13], SCADE Suite® [14], LabVIEW® [15] and SystemModeler® [16]. Open source academic tools may also be used, such as Scicos [17]. In this work, we employ Simulink, a de-facto model-based design standard for design, simulation, and code generation. Code generation will be based upon Embedded coder® [13], which provides a mature solution to automatically generate C code for our motor control use case. While appropriate requirements should be specified at different design phases of the product, we argue that multicore model-based development should also define a separate process to validate and verify the system validity against its requirements. Based on the later, we reason about information generated by the current work; this information includes intermediate artifacts, namely: a) models composed by tasks forming the application based on their periodicity, b) weight-scores for tagging the tasks depending on their criticality, c) resource-partitioning based on the multicore platform characteristics and d) time partitions and schedulers for safe and deterministic execution of the tasks in the allocated resources.

The rest of the paper is organized as follows: in Section 2 we review the related work around the area of multicore certification concerning the several domains briefly. Later on, in Section 3 we present the proposed model-based design methodology, depicting the tools and models used, optimization problem defined and the solution followed. As a use-case, in Section 4 we focus on a proof of concept multicore platform for a motor-drive, and apply its concept on a Quad-rotor, depicting the developed safe scheduling executed by the application tasks, which are partitioned to the end platform. Finally, we conclude this paper with remarks and ongoing research activities.

2 Related Work

In recent years, the industry is dominated by an increasing need for higher processing power while having a small energy footprint. It is a fact, that traditional approaches for providing more processing bandwidth such as clock frequency increment or instruction pipelining, are no longer sustainable. For this reason, embedded devices urge to transition to multicore platforms which have the potential to meet performance requirements by offering greater computational capabilities and advantages in size, weight, and power consumption (SWaP).

However, commercial domains like automotive and aerospace have special requirements to deal with, due to the high integration of systems of systems with different safety-critical levels, such as flight-critical and mission-critical functions, on a single, shared hardware device. In multicore systems, different cores share hardware resources such as caches and central memory, which were developed with a focus maximizing the overall performance; but when placed in the safety-critical context, they introduce challenges to predictability.

From an industrial point of view, safety-critical systems (e.g. aerospace) are subject to costly and time-consuming certification processes, requiring proofs of device-predictable behavior under fault-free and certain hazardous conditions. Despite these concerns, companies are moving towards a higher exploitation of COTS devices to reduce development costs [6]. For the multicore certification process, it is of paramount importance to ensure the *Execution Integrity* of its software components. This means that the application will be correctly executed in normal conditions while the system state will be predictable in non-nominal situations. Usually, not all functions have the same requirements to deal with, and this difference is expressed by their *Criticality*. A formal definition of criticality can be obtained with reference to the safety standards [4] that define the design and development processes for safety-critical embedded systems (hardware and software). The generic standard IEC-61508 requires that a *sufficient independence* is demonstrated between functions of different criticalities. A practical interpretation of this principle is the concept of *robust partitioning*. It is defined differently by several standards, without an officially agreed or common definition [5]. ARINC-653 [20] contains its interpretation of robust partitioning, quoted as: *"The objective of Robust Partitioning is to provide the same level of functional isolation as a federated implementation."*. Federated architecture is the traditional design for avionic architecture where each application is implemented in self-contained units. Therefore, robust partitioning consists of the following the concepts:

– *Fault Containment.* Functions should be separated in such a way that no failure in one application can cause another function to fail. Low criticality tasks should not affect higher criticality tasks.
– *Space Partitioning.* No function may access the memory space of other functions unless explicitly configured.
– *Temporal Partitioning.* A function's access to a set of hardware resources during a period of time is guaranteed and cannot be affected by other functions.

This space partitioning concept can be implemented on multicore systems with the help of Real-Time Operating Systems or Hypervisors. Simulink provides a way to address the challenge of designing systems for concurrent execution through its *Concurrent Workflow*. It uses the process of partitioning, mapping, and profiling to define the structure of the parallel application. However, this tool is not yet mature (e.g. not ideal for mixed-criticality setting), being also complex to customize. It offers basic model-based partitioning by grouping blocks to form tasks instead of finding a way to isolate functionality towards robust partitioning.

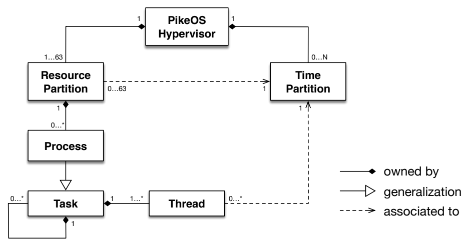
3 Proposed Methodology

As it was already introduced in the previous sections, the main purpose of this work is to offer a design methodology that deals with the application of mixed-critical applications in multicore platforms. The proposed methodology follows a model-based design approach that covers all the development stages from the initial model and till the code generation in a multicore platform using a hypervisor. As traditional embedded system development process follows the standard V-shaped lifecycle, product development process is split into a design and an integration phase. During the design phase, the model is developed and simulated. Once the results are satisfactory, the model is tested directly on the hardware platform and then deployed. The Code Generation step is the crucial point of the process where the application is evaluated without any governing assumptions. Our contributions are positioned from the model evaluation and optimization phase till the automatic code generation. The methodology enables to run different applications with different criticality levels in multicore platforms ensuring adequate safety properties.

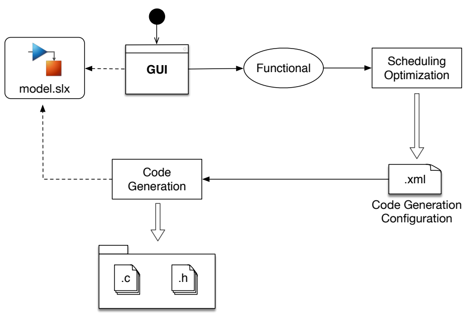
3.1 Hypervisor Selection, Model Development and DAG Generation

Before the actual model design and creation, several information is needed to be capture for the methodology effectiveness. The main concept for the design of a mixed-critical system is, first and foremost, the demonstration of sufficient independence among software components. System virtualization, which is the abstraction and management of system resources, facilitates the integration of mixed-criticality systems [7]. This approach results in independent virtual machines (partitions) that are fully contained in an execution environment that can not affect the remaining system. As there are plenty of commercial and open-source hypervisors, current methodology is based on PikeOS [8] from SysGO AG. It is a microkernel-based hypervisor, suitable for embedded platforms, certified for a series of common regulations (e.g. DO-178C, ARINC-653, ISO-26262, etc.). It has been designed for functional safety and security requirements which make it also a suitable choice for our use case. Moreover, it fully supports multi-core processor with some mechanism to improve performances. PikeOS execution entities are shown in Fig.1a.

The *Resource Partitions* implement the space separation whereas the *Time Partitions* implement the temporal separation of a robust partitioned system. In this work, we consider Resource Partition as a group of one or more threads assigned to a Time Partition. This will be convenient to simplify timing analysis and improve predictability of the overall execution. The platform used for our case is a heterogeneous embedded system including a dual ARM core and an Artix-7 FPGA. For this study, the FPGA will be considered as a HW resource for the cores and its implementation is not included in the partitioning and scheduling process. In the same way, the first SW-HW partition of the application is now executed manually depending on the designer. A schematic of the simplified design process is shown in Fig.1b.



(a) PikeOS Execution Entities (SYSGO AG)



(b) Framework Process Overview

Fig. 1. Model-based design framework with Hypervisor entities

The input to the process for our example is a Simulink model. This model is completely agnostic from its final implementation, as it is crucial to allow a clear role separation among designers, such as the domain expert modeling the algorithm, till the software and verification engineer. In this way, an agnostic implementation of the end multicore platform is sought, prohibiting the platform dependent requirements to restrict designers modeling concepts. Initially, the system designer refines the division of the model in subsystems, which are considered as the unit of execution. This refinement is still a manual process since it is highly dependent on the application and the designer preferences. Once the model is divided into subsystems, a configuration step is required which includes: a) assigning criticalities to each subsystem, b) running an initial estimation of the execution time per subsystem and c) load the platform model to see how the HW resources are used by each task.

The platform is a rough representation of the *uniform memory access* architecture as shown in Fig.2. Its simpler form contains the number of CPU and the resources available in the platform, that resemples the platform of our case study. This model, together with the resource usage information, is useful to correctly schedule tasks in order to reduce interferences. Indeed, the scheduler

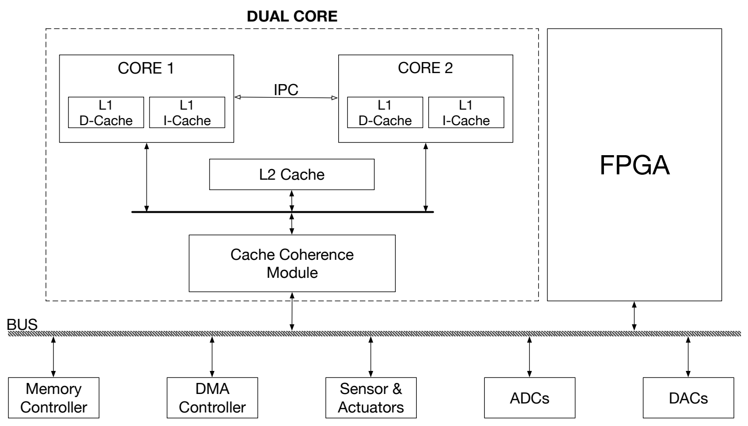


Fig. 2. Unified Memory Acess Architecture

can schedule tasks on all available CPUs, avoiding that two or more tasks access the same peice of a resource at the same time. Note this approach is not limited to hardware resource, for example, shared memory regions can be treated as additional resource hence the scheduler will prevent concurrent access to that memory region.

A current challenge in the development of parallel applications is the achievement of a good scalability with the number of threads and processors. Often the scalability is heavily reduced by the precedence order of execution among threads (usually due to data dependencies). A possible approach to model the problem is through a *Directed Acyclic Graph* (DAG). The DAG is a functional representation of the model, it consists of vertices (threads) and edges (communications/precedences) among nodes. We utilize Simulink modeling API through scripting to parse the model and create a functional representation of it. While the DAG represents a *precedence graph* among threads, it should be mentioned that it also imposes a partial order of execution of the application. To automatically generate an implementation of the application entering the code generation phase, the total order of the DAG must be computed, as it will improve safety and predictability (determinism) through partitioning and scheduling. In order to perform a real-time scheduling for the tasks on multiprocessor platforms there exist two basic approaches: the partitioned approach, in which each task is statically assigned to a single processor and migration is not allowed, and the global approach, in which tasks can freely migrate and execute on any processor. Even though the global scheduling has several advantages, this paper focus on scheduled partitioning because the global scheduling would lead to preemptions and migrations, which produce more overheads and less determinism. In particular the latter might cause certification issues that primarily we would like to avoid.

3.2 Partitioning Based on Criticalities

After extracting the DAG, the framework perform its next step, executing a partitioning algorithm based on criticalities, platform model, and task periodicity. The output of this process, two hierarchical groups, are obtained represented in intermediate DAGs:

1. *P-DAG* the DAG representing all the partitions of the system and their precedence constraints
2. *T-DAG* one for each node of the P-DAG, this DAG represent the tasks assigned to that partition along with precedence constraints

Each partition corresponds to time and resource partitions at the hypervisor domain. The isolation of the time and resource partitions will be guaranteed by the hypervisor.

3.3 Scheduling, Resource Allocation, and Factorization

As a following step, scheduling and allocation of the tasks need to be defined. This step is divided into two phases, a)*inter-partition scheduling*, and b)*intra-partition scheduling*. The intra-partition scheduling takes the T-DAG of each partition and schedule, allocate and assign priorities to each task of the graph. The problem is NP-hard in the strong sense, and a solution will be obtained through a Mixed-Integer-Linear-Programming (MILP) optimization. Inter-core communications are handled by spinlocks in order to avoid overhead from operating system to re-scheduling or context switching the waiting threads. Moreover, spinlocks are proved to be efficient if threads are likely to be blocked for only short periods, which is true to some extent depending on worst-case timing (WCET) analysis reliability. The Linear-Programming (LP) optimization problem will assign priorities such that the FIFO scheduler of PikeOS correctly schedules threads under the optimal schedule solution found including additional constraints imposed by the use of spinlocks. Similarly, inter-partition scheduling part, takes the P-DAG and schedule each partition by directly assigning time-slots and execution order, for the hypervisor. PikeOS will also guaranty error containment inside partitions, in case a resource is malfunctioning (functional alteration of the resource) during the task execution.

As partitions can have different rates, it is assumed that rates are an integer multiple of a base-period. Considering the *Major Time Frame* which is the Least Common Multiplier of all rates multipliers, some partition should execute more than once. For this reason, a *factorization* phase is required. During this step, each partition in the P-DAG is repeatedly replicated in order to execute in the Major time frame. An example of factorization is shown in Fig.3. It should be added that as the P-DAG imposes a partial order of execution among partitions, a correct time slots assignment must take into account this order. As the intra-partition scheduling is considered to be an NP-hard problem, a Branch-and-Bound optimization step is executed, taking the factorized P-DAG in order to identify optimal solution if exists. At the same time, as our focus is a

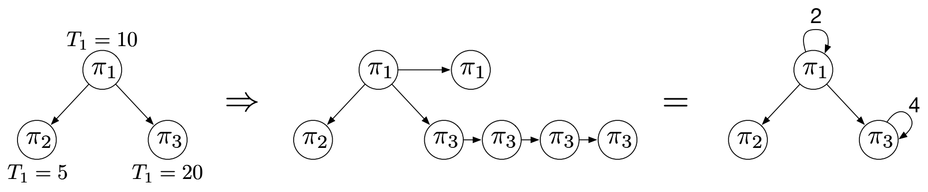


Fig. 3. P-DAG factorization example

model-based design approach towards the safe execution of the tasks led by the generated DAGs, we argue that model checking techniques (such as Simulink Design Verifier or NuSMV) could be used to exhaustively verify temporal isolation of the enumerated resources assigned to execute the tasks. Abstractions on the other hand, are necessary to describe the embedded systems platform, leading us to partial verification of the solution. Thus, from practical point of view, we utilize the certified PikeOS hypervisor solution which proves safe temporal isolation and determinism of the mixed-critical tasks execution on the available cores.

3.4 Code Generation

Once the configuration is loaded, we realize the software architecture described in the configuration file, by scripting. In particular, we have to address the communication issue between the different resources. All the threads inside a process share the same virtual memory address space, so a global variable containing an output (or input) of a thread is accessible from every thread. When the threads are scheduled inside two different partitions, this variable is no longer accessible. For solving this problem, two approaches are possible:

- Communication Primitives: use message oriented communication such as Sampling or Queuing Ports.
- Shared Memory Regions: define shared memory regions between the two partitions.

While shared memory is suitable for a large amount of data, usually a single inter-partition communication channel, which is the implementation of a Simulink Line, transfers a limited amount of bytes. After the aforementioned phases, the information obtained is:

- Time partitions schedule scheme, where each partition has one or more reserved time slots.
- For each partition, which threads it contains tagged with starting times, priorities, core mapping and specification for thread-to-thread communications.

Model Inport and Outports in Simulink that implement an inter-partition communication, need to be substituted with surrogate blocks that, respectively, implement a Read and Write on the Sampling Port. The mechanism that Simulink provide for extending its built-in modeling functionality is the definition of Custom Blocks. Each custom block is composed of two files: an S-Function that describe the behavior of the block during simulation and a TLC file specifying the code that is going to be generated out of it. Each block has a target file that determines what code should be generated for that block. Within each target file, block functions specify the code to be output during the start function, output function and update function. After the Code generation block initialization (i.e. BlockInstanceSetup and BlockTypeSetup) different TLC function generate the resulted executable code. This partitioning configuration is stored in an XML file. It is used to drive the automatic code generation and integration with the hypervisor IDE. Initially, the model is automatically adapted to implement the software architecture described in the XML file, addressing all communication issues. The reason is the following: as all the threads inside a partition will share the same virtual memory address space, a global variable containing an output (or input) of a thread is made accessible from every thread. When the threads are scheduled inside two different partitions, this variable will no longer be available. To overcome this problem, each Simulink line that represents an inter-partition communication channel is automatically substituted with communication through *Sampling ports*. As a result, a fresh value will be present providing a validity flag on the expected refresh rate which is crucial for detecting faults occurred to one of the predecessors of the executing thread; and eventually, implement some handler for a not up-to-date value.

This operation, which is a model-to-model transformation, is fully transparent to the designer. Therefore, any Inport or Outports (which are blocks) that implement an inter-partition communication need to be substituted with other blocks that, respectively, implement a Read and Write on the Sampling Port. The mechanism that Simulink provides for extending its built-in modeling functionality, and generate the required C code, is the definition of *Custom Blocks* which has been exploited to achieve the model transformation.

Once the model has been adapted, in the second step we scan the model and generate the code for each subsystem. To drive the code generation process, a Simulink *Custom System Target File* has been developed. Using the aforementioned template, C code will be generated that will be based on code mappings populating the final source files, for every new block is encountered. The generation template also creates an additional XML file which outlines the interface of the generated code, including: a) The list of all the source files, b) The name of the structure containing the Inputs/Outputs for the subsystem code and every structure element (ports) with its name and dimension, c) The name of the entry point functions.

Once the code of each subsystem is available, in the third step code generates the code for each partition. The generation is driven by all the XML file that have been generated by the Custom System Target File, enriched with the information produced by the partitioning, scheduling and allocation optimization. Each resource partition will contain a single PikeOS-Hypervisor Process which is composed by the main thread and eventually some optional child threads. The main thread is loaded by the PikeOS kernel, and it is responsible for initializing all the children threads. Therefore, all the subsystems are glued together with the main forming partition by a custom TLC. As a result, the main generated will: 1. Create all the child threads. 2. Enter in an infinite loop where it will resume all child threads and wait for the next period activation.

Finally, through the PikeOS Integration Project we configure the generation of the PikeOS boot image, including all the partitions. The Integration Project uses an XML-based file to store all the configuration information. The generation of the whole file is complex, so, for the seek of simplicity only some XML snippets are generated. This does not lead to a loss of functionality because it is possible to develop a tool that, once an Integration Project is available, can scan it and add programmatically the snippets. With a procedure similar to the generation of the partitions, as a last step of the code generation, a TLC file will generate the XML snippets, in particular, they will cover the following sections of the configuration file: 1. *Partitions and Ports* the list of the partitions with the relative ports, 2. *Channels* the list of channels used by the ports to communicate i.e. the port connection list and 3. *Schedule Scheme* the inter-partition schedule scheme

4 Experimental Results

After the model configuration and code generation of the optimized task allocated to the cores, we describe the experimental results driven by our platform. The hardware used for this proof-of-concept demonstrator. The selected processing unit is based on the Zynq-7000 device, a System on Chip (SoC) that includes two ARM cores and an Artix-7 FPGA fabric (Fig.4). The motor and drive used is the FMCMOTCON2 developed by Analog Devices. The system includes a brushless DC motor (BLY171S-24V-4000) with a dynamometer, and the drive and connection boards compatible with the ZedBoard platform that contains the Zynq-7000 device.

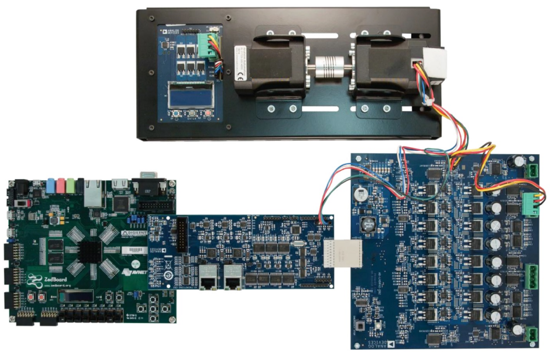


Fig. 4. Zynq-7000 Experimental platform

The implemented system consists of a speed control and Prognostics and Health Management (PHM) algorithm to perform system identification. The implementation is divided into two main parts: the SW side implemented in the multicore, includes the speed control loop, a mode control state machine and the PHM algorithm, while the HW-FPGA implementation includes the current control loop with its Clarke and Park transformations, Space Vector Modulation, and different sensor acquisition algorithms to measure current and position.

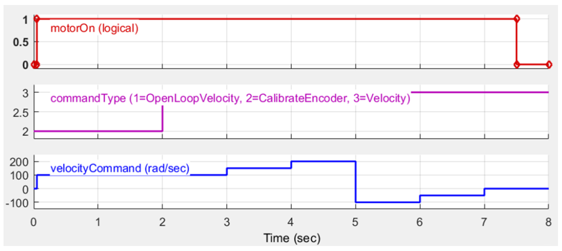


Fig. 5. Reference signal for Hardware-in-the Loop Simulation

The system design for the FPGA is a combination of manually designed and commercial IPs and automatically generated blocks using the HDL Coder from Simulink models. To link both parts automatically, it is necessary to develop a reference design in Vivado (XILINX IDE for the Zynq-7000 device) and extract its TCL script. HDL Coder creates another TCL file with the Simulink blocks and merges them together. In order to provide to the reader a full picture of our application needs, we perform and show simulation results with and without our methodology.

Initial simulation results, show the real-time interaction between the hardware platform and the model, for a single motor control at an isolation phase. This allows a comparison with simulation results and enables the real-time interaction with the user since it is possible to change the speed set point and different operation modes during operation. Controlling one mixed-critical application on a single motor will provide us with the necessary information about the application execution coverage, periodicity, and complexity. This step is essential for the validation of the standalone control algorithms and observes differences between the FPGA and ARM implementations in the real device without changing manually VHDL or C code. This experiment is following the speed pattern shown in Fig.5 and Fig.6.

As the single motor control example is relatively straightforward for showing the tool capabilities on multicore, we aim to elevate the use-case to a multivariant control problem with multiple motors. Therefore, as additional example model, we consider a Quad-rotor flight control with a pan/tilt camera on it shown in Fig.7. The models include the control of six motors, four of them for the quadrotor itself, the last two for a pan-and-tilt camera.

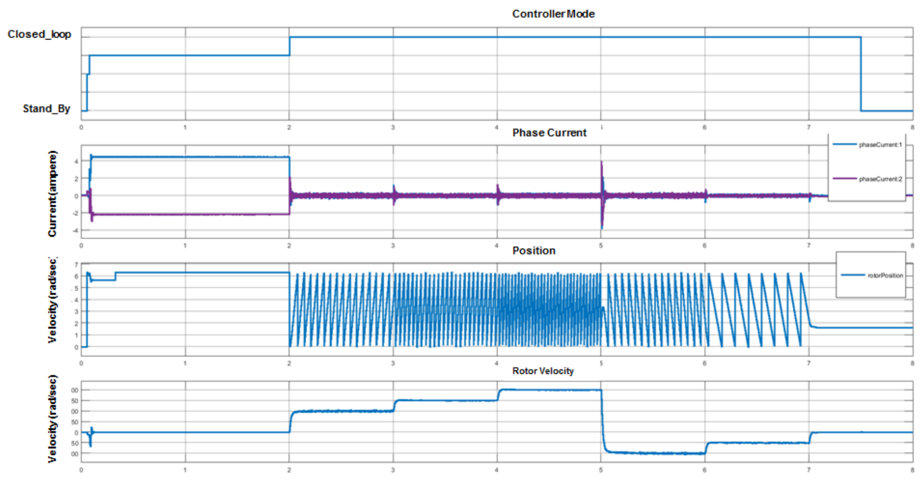


Fig. 6. Test results from Hardware-in-the Loop Simulation

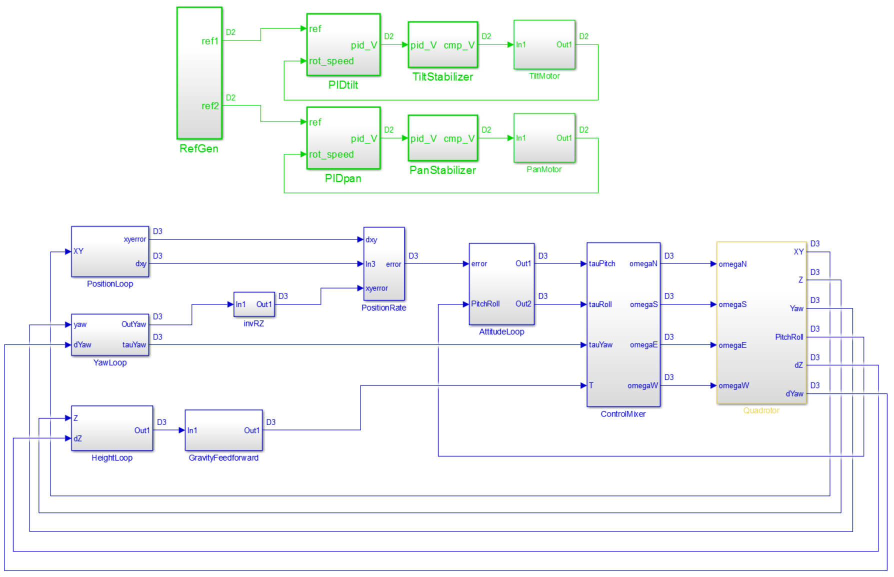


Fig. 7. Compiled Simulink Model Model

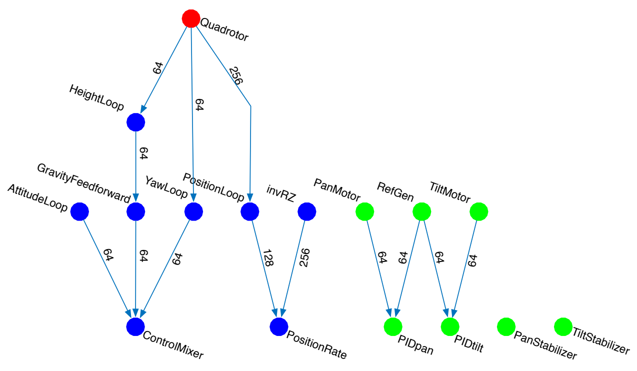


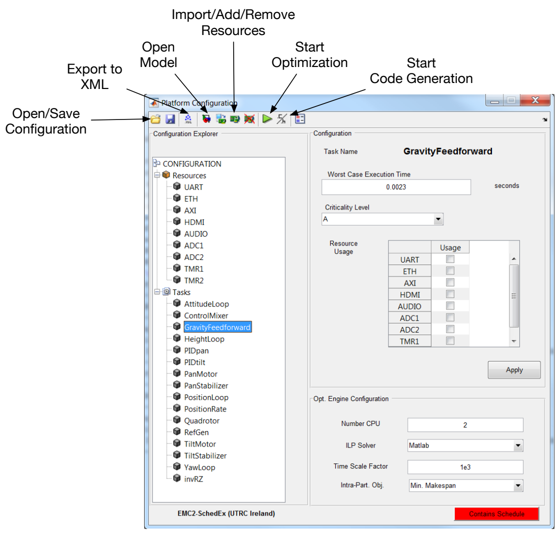
Fig. 8. Functional model Task-set for Quad-rotor example

The adopted drone control scheme is inspired by [10] with minor changes introduced to comply with our design restrictions. The DAG in Fig.8 is automatically extracted as the framework is started, it is then enriched with the information manually configured by the designer through the Framework Graphical User Interface (GUI) in Fig.9. Once the configuration is complete, the designer can start the partitioning, scheduling and allocation optimization. Generated scheduling and optimization diagrams are generated and given to the designer, for him to select if the design produces acceptable results. Indicative diagrams for the quad-rotor example are shown in Fig.10 and in Fig.11.

In order to assess our model executed on multicore platform, *speed-up* and the *efficiency* [11] techniques will be used. The Speed-up technique ($S(p)$) is defined as the ratio of the elapsed time ($E(p)$) when executing a program on a single processor to the execution time when p processors are available ($E(p) = \frac{S(p)}{p}$). Speed-up refers to each partition: in the quad-rotor example, the tasks have been scheduled on two cores obtaining a speed-up of 1.9927 for partition one and two, while partition one, having only one task, has no improvements. In theory, the speed-up can never exceed the number of processors in the best case.

At this point all information required by the code generation is made available. As described in section 3, first the model is adapted to the communication architecture determined by the partitioning process. This operation is fully automatic since all the communications are encoded in the output of the partitioning problem. Than the code of each block is generated using the developed System Target and finally merged to the partitions.

The final steps only generate the code that glues together sub-systems (tasks). Indeed, inter-partition communication is generated by Simulink based on our custom blockset code generation functionality. As a result, out of the code generation and the task scheduling process, XML snippets generated will help the configuration of the PikeOS Integration Project. Due to space limitations we omit the organization information of the integration project.



Configuration

Task Name **GravityFeedforward**

Worst Case Execution Time 0.0023 seconds

Criticality Level A

Resource Usage

	Usage
UART	☐
ETH	☐
AXI	☐
HDMI	☐
AUDIO	☐
ADC1	☐
ADC2	☐
TMR1	☐

Opt. Engine Configuration

Number CPU	2
ILP Solver	Matlab
Time Scale Factor	1e3
Intra-Part. Obj.	Min. Makespan

Fig. 9. Framework Graphical User Interface

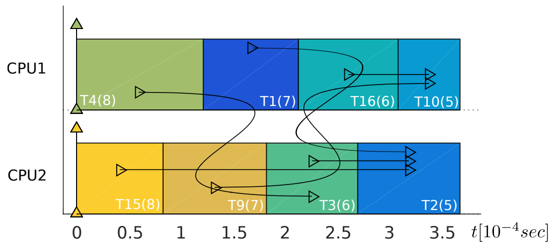


Fig. 10. Optimization results: Partition one Intra-Partition Schedule

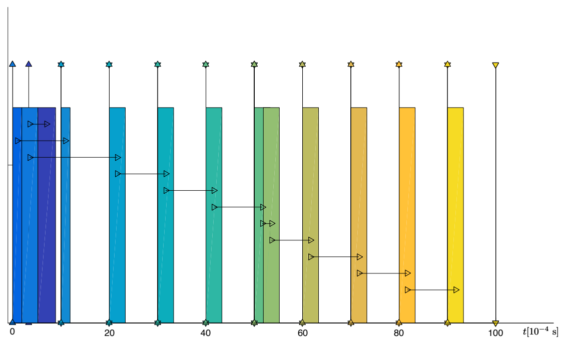


Fig. 11. Optimization results: Inter-Partition Schedule

5 Conclusions and Look Ahead

Automated analysis and fast verification are currently important milestones pursued by major companies for certifying their products. To this end, model-based design approaches nowadays have proven irreplaceable when designing a single aerospace device or an integrated platform. The aforementioned technique presented in this paper, provides an automated tool-assisted solution that steps into the complete development cycle for early and accurate error detection, prior to the final testing of the product. We have presented a model-based approach for the development of multicore prototypes towards safe temporal core isolation of the available resources while performance of our use case is respected. Following resource and time partitioning at a model level of the application tasks, we automatically optimize and allocate tasks to the resources, guarantying their safe execution with a COTS hypervisor solution. As a look ahead, our research activities will focus on evaluating and developing new modeling techniques, combining those with FMI-based co-simulations platforms [23] and implementing new code generation libraries and certification workflows towards the finalization of standardization efforts of manycore solutions for safety-critical domains.

Acknowledgments. This work is supported by the EMC^2 ARTEMIS project: *Embedded Multi-Core Systems for Mixed Criticality applications in dynamic and changeable real-time environments*. UTRC is supported jointly by the European Commission and the Irish Development Agency (IDA), Project Number: 621429.

References

1. Markus L., Conte T.. Embedded Multicore Processors and Systems. In: IEEE Micro 29, pp. 7-9, June 2009.
2. Basagiannis S. Software certification of airborne cyber-physical systems under DO-178C. 2016 IEEE Int. Workshop on Symbolic and Numerical Methods for Reachability Analysis (SNR), Vienna, pp. 1-6, April 2016.
3. Paun V.-A., B. Monsuez, P. Baufreton. On the Determinism of Multi-core Processors. In Proc. of 1st French Singaporean Workshop on Formal Methods and Applications (FSFMA), volume 31 of OpenAccess Series in Informatics (OASIcs), pp. 32-46, 2013.
4. Ernst R. and Di Natale M. . Mixed criticality systems - A history of misconceptions? IEEE Design Test, 33(5):6574, Oct 2016.
5. Jean X. , Faura D. , Gatti M. , Pautet L., and Robert T. . Ensuring robust partitioning in multicore platforms for ima systems. In 2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC), pages 7A417A49, Oct 2012.
6. Fumey M., Jean X. , Gatti M. and Berthon G. The use of multicore processors in airborne systems (mulcors), 2012. https://www.easa.europa.eu/document-library/research-projects/easa20116
7. Trujillo S. , Crespo A. , and Alonso A. Multipartes: Multicore virtualization for mixed-criticality systems. In 2013 Euromicro Conference on Digital System Design, pages 260265, Sept 2013.

8. PikeOS Hypervisor. SYSGO AG, URL: `https://www.sysgo.com/products/pikeos-hypervisor/`.
9. Bhatt D. , Madl G., Oglesby D., and Schloegel K. Towards scalable verification of commercial avionics software. In AIAA Infotech@Aerospace 2010, page 3452. 2010.
10. Corke P.. Robotics, Vision and Control Fundamental Algorithms in MATLAB. 1. ed. 2011, corr. 2. print. Berlin: Springer, 2011.
11. Grama A. Introduction to parallel computing. Pearson Education, 2003.
12. Basagiannis S., Gonzalez-Espin F. Towards Verification of Multicore Motor-Drive Controllers in Aerospace, In Proc. of Int. Conf. on Computer Safety, Reliability, and Security, Springer-LNCS, pp-190-200, vol 9338, Dec. 2015.
13. Simulink. URL: `http://www.mathworks.com/products/simulink/`.
14. SCADE Suite. URL: `http://www.esterel-technologies.com/products/scade-suite/`.
15. NI LabVIEW. URL: `http://www.ni.com/labview/`.
16. Wolfram SystemModeler. URL: `http://wolfram.com/system-modeler/`.
17. Scicos. URL: `http://www.scicos.org/`.
18. DO-178B, 'Software Considerations in Airborne Systems and Equipment Certification. Requirements and Technical Concepts for Aviation', Radio Technical Commission for Aeronautics (RTCA), 2012.
19. RTCA DO-254 / EUROCAE ED-80: Design Assurance Guidance for Airborne Electronic Hardware. Radio Technical Commission for Aeronautics (RTCA) and EURopean Organisation for Civil Aviation Equipment (EUROCAE).
20. ARINC-653 P1 revision 3: Avionics Application Software Standard Interface. Aeronautical Radio Inc, 2010.
21. RTCA DO-331: Model-Based Development and Verification Supplement to DO-178C and DO-278A. Radio Technical Commission for Aeronautics (RTCA), 2011.
22. IEC-61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems. International Electrotechnical Commission.
23. Gorm-Larsen P., Fitzgerald J., Woodcock J., Fritzson P., Brauer J., Kleijn K., Lecomte T., Pfeil M., Green O., Basagiannis S., Sadovykh A.: Integrated tool chain for model-based design of Cyber-Physical Systems: The INTO-CPS project. In proc. of 2nd International Workshop on the CPS Data Workshop, CPS-Week, pp:1-6, 2016.
24. Nowotsch J. and Paulitsch M. , Leveraging multicore computing architectures in avionics. European Dependable Computing Conference, vol. 0, pp. 132143, (2012)