Emsoft'09, ACM, pp 235--244. 2009

Handling Mixed-Criticality in SoC-based Real-Time Embedded Systems

Rodolfo Pellizzoni, Patrick Meredith, Min-Young Nam,
Mu Sun, Marco Caccamo, Lui Sha
University of Illinois at Urbana-Champaign
{rpelliz2, pmeredit, mnam, musun, mcaccamo, lrs}@illinois.edu

ABSTRACT

System-on-Chip (SoC) is a promising paradigm to implement safety-critical embedded systems, but it poses significant challenges from a design and verification point of view. In particular, in a mixed-criticality system, low criticality applications must be prevented from interfering with high criticality ones. In this paper, we introduce a new design methodology for SoC that provides strong isolation guarantees to applications with different criticalities. A set of certificates describing the assumed application behavior is extracted from a functional Architectural Analysis and Design Language (AADL) specification. Our tools then automatically generate hardware wrappers that enforce at run-time the behavior described by the certificates. In particular, we employ run-time monitoring to formally check all data communication in the system, and we enforce timing reservations for both computation and communication resources. Verification is greatly simplified because certificates are much simpler than the components used to implement low-criticality applications. The effectiveness of our methodology is proven on a case study consisting of a medical pacemaker.

1. INTRODUCTION

Systems-on-Chip (SoCs) are increasingly popular in the embedded system domain because they consume less power and cost less money than the multi-chip solutions they replace. We believe that the SoC paradigm is especially promising for safety-critical embedded systems such as those employed in the medical market [35]: it is easier to formally verify the correctness of critical functionalities when they are implemented in hardware rather than in software on top of an OS and library stack [4]. Furthermore, SoC design can offer higher hardware reliability.

However, implementing safety-critical applications on SoC brings additional design challenges. The high degree of integration possible in modern fabrication processes can easily lead to *mixed-criticality systems*: multiple applications with different criticalities run simultaneously on the same chip. As an example, consider a medical pacemaker (a more detailed description can be found in Section 2). Implanted cardiac pacemakers are used to provide pacing assistance for patients with slow or abnormal heart rates. The life-critical pacing component uses leads to send shocks to the heart to force contraction. However, modern pacemakers also implement a variety of other applications such as data logging and remote communication and programming that are used for diagnosis. Since these lower-criticality functionalities must communicate with the life-critical module and share physical resources such as battery energy, memory and communication bandwidth, faults could potentially propagate from a lower to a higher criticality application. This is a daunting problem because formally verifying and/or certifying non safety-critical modules is either extremely expensive or unfeasible: they typically make large use of both software and hardware Intellectual Properties (IP) such as CPU cores and OSes that are very complex.

In this paper, we introduce a new design methodology for SoC that specifically targets mixed-criticality systems. Our methodology provides strong isolation guarantees to applications and imposes limits to fault propagation. In detail, we distinguish between functional isolation, which deals with the correct exchange of data, and physical isolation, which deals with the correct sharing of system resources. Our methodology is similar to the concept of *Platform-Based Design* (PBD) [29]. A platform is a library of (usually parametric) components. A platform instance is a set of library components selected to generate a concrete design. In a PBD design flow, the designer first specifies the system functionality using an implementation-independent description language. The designer then selects a suitable platform and performs *mapping* of functional elements to platform components, thus creating a platform instance. Our platform contains three types of architectural components: processors (which can either be CPUs or *hardware processors*, e.g. logic circuits implementing a specific functionality), communication infrastructures and memories. Functional specification, architectural specification and mapping are all performed using the Architectural Analysis and Design Language (AADL) [14]. AADL is rapidly gaining support in safety-critical markets such as avionic and medical domain and it has been applied to many industrial examples (see Section 2). Mapping is performed by binding functional processes to architectural processors and specifying processes' requirements in term of data transfers and resource usage. This requirement specification creates a *certificate* for each process, e.g. it determines the complete set of acceptable run-time behaviors for the process.

We provide isolation using the following key idea. Each processor is encapsulated in a *monitoring hardware wrapper* that controls all communication and resource usage by the processes executed on that processor, and can therefore enforce their certificates at run-time. System level verification is greatly simplified because it is sufficient to verify that the system is correct based on the certificates of low-criticality applications, and not the complex behavior of the IPs that

EMSOFT '09 Grenoble, France

implement them[1].

To summarize, our main contributions are as follows. 1) We developed a new PBD methodology for SoC based on AADL that is targeted for safety-critical systems. In particular, we developed tools that are able to automatically generate monitoring wrappers based on an AADL specification. 2) We employ run-time monitoring [8] to enforce functional isolation by formally checking the communication behavior of processes at run-time. If a violation is detected, the wrapper is able to take a *preventive* measure by rejecting the faulty communication. 3) At the physical isolation level, we derive a static coschedule of computation and communication and assign timing reservations to all processes in the form of temporal budgets based on their certificates. The wrapper checks usage of both computation and communication resources and prevents a process from exceeding its assigned budget. Finally, in battery-operated devices the platform can monitor overall power consumption and use the wrappers to shut down low-criticality processes and save energy for higher criticality applications.

Our AADL-based system modeling is introduced in Section 2, together with a case study based on a medical pacemaker; we continue to elaborate on the case study in the rest of the paper. Section 3 describes our architectural implementation, focusing on the wrappers. Sections 4 and 5 details functional isolation and timing isolation respectively. Finally, in Section 6 we provides concluding remarks and future work.

Related Work. The concept of PBD is well established and several SoC platforms are commercially available [10, 26]; however, they are not targeted at safety-critical systems. Similarly, a variety of PBD and model-driven design methodologies have been proposed (see [29] for an overview), several of which (for example, see [12, 19]) support formal verification of modeled application behavior. The main difference is that our methodology and platform does not simply allow to verify the correctness of a high-level model: it enforces application isolation at run-time, as long as the designer can correctly specify system isolation requirements in the model. A large body of work is more directly related to either AADL modeling or run-time monitoring; we discuss it in Sections 2, 4.

2. SYSTEM MODEL

The importance of model-based Architecture Description Languages (ADLs) and formal analysis of system models is becoming apparent in the industry of hard real-time systems, such as avionics, aerospace, and medical devices. Model-based ADLs provide the automation capability of generating beneficial abstractions from system models to be verified or examined by computed-aided tools. AADL is a type of ADL initially developed for the avionic market. It is based on 15 years of research, including the MetaH language developed by Honeywell Labs and several DARPA programs [6]. AADL-based tools and analysis methodologies have been developed in the area of safety analysis [32], dependability [28] and schedulability [31, 17, 30]; however, to the best of our knowledge this is the first time that an AADL-based model is used to automatically generate code that enforces safe process behavior. In this section, we first describe how AADL can be used to provide both functional specification and mapping in our platform. Then, we introduce our case study of a medical pacemaker.

2.1 AADL-Based Modeling

AADL lets the designer specify the logical functional model separately from the hardware platform. An AADL specification is comprised of different components and their interactions. Components used for the logical design include `process`, `thread`, and `data`. AADL execution platform components include `processor`, `memory` and `bus`; they have a one-to-one correspondence with our architectural components. Each `processor` represents either a custom-built hardware processor or a CPU. `Memory` is used, among others, for all external memories (such as external SRAM or DRAM chips) used to hold shared data. `Bus` represents the unique communication infrastructure used to interconnect all processors and memories. Finally, the `system` component is a special composite component which is used to either encapsulate other components, divide them into groups or represent any new abstract entity that can not be modeled by preexisting components. In our design methodology, the logical components and architectural components belonging to a platform instance are grouped in separate `system` components for modularity and clarity of representation. All components are tagged with properties which add extra information that can not be expressed by structural descriptions (for example, the processor type is specified by a `class` property). The core AADL standard has pre-declared properties that support real-time scheduling as well as other areas of research. AADL also provides the syntax to add new user-defined properties.

Every application is modeled as a collection of `process`es with one or more `thread` subcomponents. Threads are active agents: they receive inputs, process and output data. Period, deadline, and execution time properties are associated with each `process` as a timing reservation for all of its subcomponent threads. The platform guarantees that at each period a process is provided with an amount of execution time at least equal to its request within a time window equal to the specified deadline. Each `process` represents a dedicated memory and design space protected by the system architecture. The platform ensures that all threads inside a `process` are functionally isolated from other threads running on different `process`es.

Two types of interprocess communication are supported. A process can declare any number of input *message queues* to which other processes can send data. Furthermore, shared data objects can be declared and accessed by any process. In AADL, message queues are modeled as `event data port connections` and `data accesse`s are used to connect processes to shared data object. Each communication has an associated `data` component which represents the type and size of data that is to be sent or shared and an associated `deadline` property which represents the maximum allowed amount of time to complete the data transfer. The acceptable communication behavior of a process can be further specified by a `PropertyList` and associated `EventList`s. Each entry in an `EventList` describes a specific communication event, e.g. a send/receive to a message queue or a DMA read/write to a data object. Properties specify behavior as a formal trace of events at run-time, described by either a formula in Past-Time Linear Temporal Logic (PTLTL) or an Extended Regular Expression (ERE) pattern. Property syntax is described in Section 4.

Platform mapping is performed by "binding" logical components to architectural components. Each `process` is bound to a `processor` and each `data` component is bound to a `memory` (either a message queue or an external memory). Each HW processor can execute a single process, while a CPU processor can execute multiple processes. Note that some properties of logical components depend on the binding. For example, required execution time depends on the processor to which a process is bound. The values of these properties

[1] Note that obviously the platform can not prevent a low-criticality process from failing if the unverified processor on which it is executed suffers a critical internal error. Therefore, the certificate must state that such process can stop at any time. However, the platform will prevent the internal error from propagating to other processors.

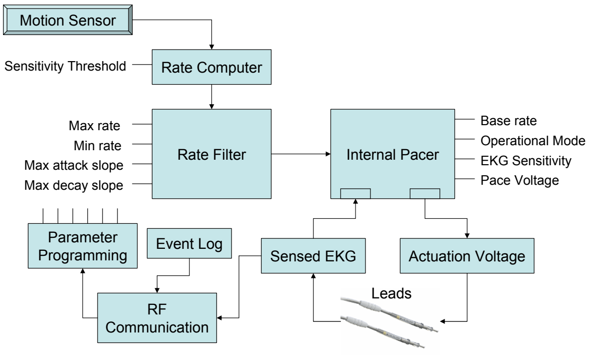


Figure 1: Pacemaker Block Diagram

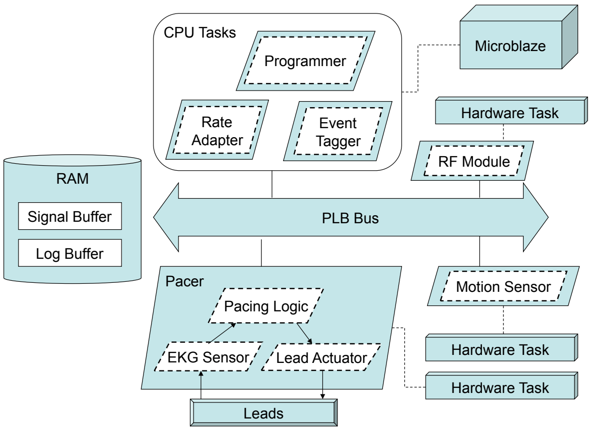


Figure 2: Mapped Pacemaker Platform

need to be re-examined every time a related change in the mapping occurs.

The AADL platform specification would not be very useful if the designer had to manually check it for correctness and convert it into an implementation, since the potential for human error during translation is very high. As such, we built a set of tools to assist the design flow. Our AADL tool makes sure that the model conforms to predefined *analysis specific* rules for correctness; these include all aforementioned binding rules. The tool also automatically derives some property values required for implementation which depend on the binding (in particular, addresses are associated with all memory elements), and generates an XML file based on a customized schema containing a description of the platform instance. In particular, each process is characterized by a certificate specifying timing requirements for the process itself and its communication plus all attached formal properties. Additional tools read the XML file and automatically generate the monitoring wrapper; a detailed description is provided in Sections 3, 4.

2.2 Case Study: Medical Pacemaker

Pacemakers are one of the most critical medical devices [35]. Once they are implanted, they must continue to operate correctly with very little maintenance for at least 5 years. Having methods of self-diagnosis and recovery of software and hardware errors could go a long way for prolonging the lifespan of reliable operation. There have been designs to ensure the safety of pacemaker systems in terms of the control algorithm [4], including model-checking of the pacing controller using UPPAAL [34]. However, such previous work is based on the assumption that the underlying implementation can offer strong isolation guarantees for memory, power and communication. In this paper, we take a more general approach where isolation guarantees are not a by-product of the system implementation, but are instead specified as requirements in the functional model.

A typical functional decomposition of cardiac pacemaker is shown in Figure 1. A pacemaker is connected to the heart via leads (typically two). These leads provide EKG signals which the internal pacer uses to detect whether the heart is contracting properly and to send shocks to the heart to force contraction. The internal pacer has various control parameters which can be tuned to each patient: these include a reference rate of pacing, the operational mode, and the thresholds for sensing and actuation. The operational mode determines how to pace the heart; modes can determine which leads are used for sensing if any, whether rate adaptation is used, and whether certain types of special therapies should be provided.

A simple pacemaker only needs the leads and the internal pacer to be functional. However, for more effective pacing, it is desired to have the pacing rate adapt to the activity levels of the patient. Rate adaptation normally uses some form of motion sensing, usually through accelerometers or pressure sensors. The motion data is then used to compute a rate. A rate computation uses a sensitivity threshold to filter out motion noise (i.e. when riding a car). Furthermore, to match biological characteristics, the computed heart rates values are bounded (min and max rate) as well as their rates of change (attack and decay slope).

Finally, the pacemaker employs a communication component using RF signals. RF communication allows medical personnel to program various parameters of the pacemaker without intrusively removing the pacemaker. Furthermore, logged events and real time sensor data can be sent to the medical personnel during diagnostics and check-up.

A schematic of an AADL-modeled pacemaker platform instance is shown in Figure 2. The life-critical functionality is the core pacing implemented by the *Pacer* process in a HW processor. The Pacer includes the sensor and actuation interfaces to the leads and also the pacing logic. *EKG Sensor* data arrives at the frequency of 128 16-bit samples per second, which also determines the Pacer period. Also, the Pacer logs *EKG Signal* values and *Shock Event*s in the *signalBuffer* memory area for later retrieval and diagnosis. The values being written to the signalBuffer are timestamped and will eventually be overwritten in a circular manner. The *Motion Sensor* process is also implemented in hardware. It samples data at 80Hz and sends the measured values to the *Rate Adapter* through a message queue.

Three processes requiring complex microprocessing functionality are executed on a Microblaze CPU. The Rate Adapter uses motion data to compute a reasonable rate required for pacing and sends it to the Pacer twice per second; its failure is not life-critical but could still cause significant discomfort to the patient. The other two processes are less critical since they do not immediately affect the condition of the patient. The *Event Tagger*, running at 8Hz, reads in a window of EKG signal values and shock events from the signalBuffer and finds anomalies that are then logged to another section of memory (the *logBuffer*) for more permanent storage and later diagnostics. Finally the *Programmer* is used to send parameter updates. It processes commands sent by medical personal through the *RF Module* (implemented in hardware) and it sends rate adaptation parameters (ie. max/min rate, attack/decay slope) to the Rate Adapter process and pacing parameters (ie. sensor/actuator thresholds, base rate) to the Pacer process through message queues.

3. ARCHITECTURE IMPLEMENTATION

In this section, we detail the architectural components that comprise our platform focusing on how the monitoring wrappers can enforce isolation. Our general platform principles are orthogonal to the specific chip fabrication process being used, albeit certain implementation details and the choice of available IPs necessarily depend on it. Our described prototype implementation is based on a Xilinx Virtex-5 FPGA [36].

Figure 3(a) shows a block diagram for an example platform instance. For simplicity, the example is composed of only one hardware processor, one CPU and one external memory, but note that in our prototype implementation the number of CPUs, HW processors and memories is only limited by the available chip resources. Monitoring wrappers depend on the type of processor they control: *HW wrappers* and *CPU wrappers* are employed for hardware processors and CPU respectively. Each wrapper is comprised of two modules: an *interface module* and a *control module*. The interface module mediates access from the processor to the communication infrastructure and provides standardized communication services. Processes can only exchange information by transmitting data through their interface module[2]. This effectively prevents fault propagation since the interface module is able to reject any data transfer that would lead to a certificate violation. The control module implements the actual monitoring logic: it checks all data transfers performed by the interface module against the property specification included in the certificate, and enforces the process timing reservation. A more detailed description of the interface and control module is provided in Section 3.2. Similarly, each external memory is connected to the communication infrastructure by a memory controller; however, since memories are passive component, no monitoring is required. Finally, the platform includes three additional *global modules*: the *Communication Scheduler*, the *Frame Generator*, and the *Power Manager*, which are used to enforce communication timing reservation and power consumption. They are described in Section 3.1.

We can distinguish among four types of blocks in Figure 3(a): 1) blocks that are generated by the designer. These include the hardware logic of each HW processor implementing the executed process and the code for each software process executed on a CPU. 2) Blocks that are automatically generated by our implementation tools. These include the portion of the control module that implements the certificate and the Communication Scheduler. 3) Blocks that represent available IPs. These include: A) memories and their controllers; B) the communication infrastructure; C) CPU and associated Operating System (OS). 4) Blocks that are manually written by the platform provider, and depend both on the fabrication process and the supported IPs. These include the wrappers, Frame Generator and Power Manager. We implemented these blocks in a parameterized, mixed VHDL/Verilog Register-Transfer-Level (RTL) description.

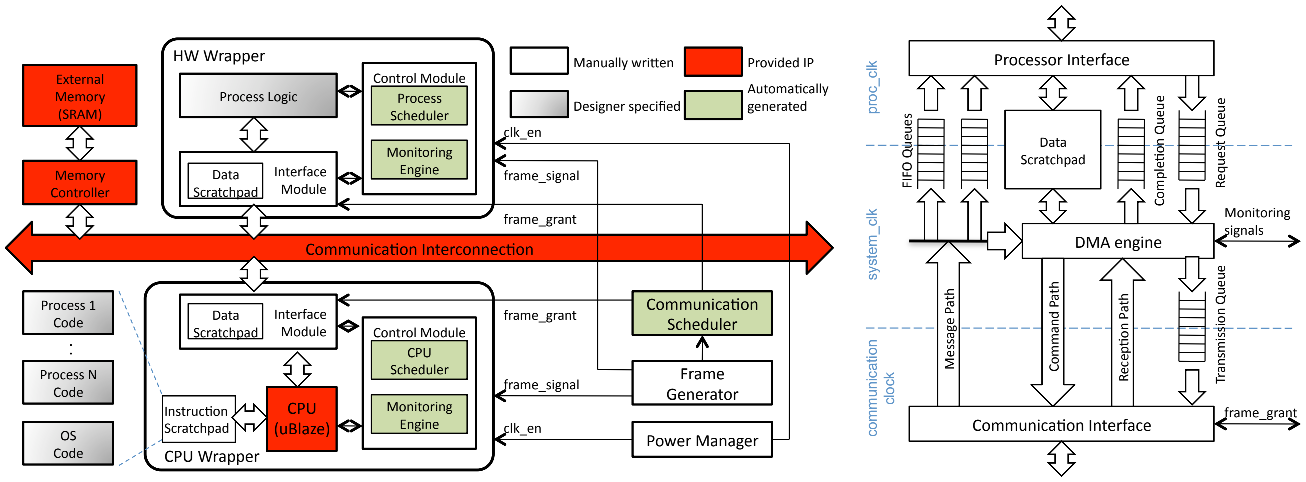
The platform can support a variety of different IPs, albeit due to time constraints we only implemented wrapper support for some of them. However, specific constraints must be imposed on IPs to ensure required isolation and timing predictability. In what follows, we specify the required constraints for A) memories, B) communication infrastructure and C) CPU and OS. A) The access time to external memories should be predictable, or at least upper bounds must be easily computable. Our pacemaker prototype employs an external SRAM, which offers deterministic access time. Predictable DRAM controllers for real-time systems have been described in the literature [2]. B) Shared buses, segmented buses and Networks-on-Chips (NoCs) can all be employed as communication infrastructure, but the specific IP choice impacts the derivation of the communication schedule as described in Section 5. For simplicity, in the rest of the paper we consider a simple shared bus, deferring the analysis of more complex infrastructures to future work; our prototype employs the IBM PLB [18]. C) We do not allow caches on CPU. While caches can greatly improve average computation times for general purpose computing, their inherent unpredictability makes it very hard to obtain tight computation time bounds for processes and to schedule cache miss traffic. Instead, we provide each CPU with local instruction and data memory used as *scratchpads*. Scratchpads can be as fast as caches, but data transfers to/from external memory must be explicitly initiated by the running process instead of being handled by the cache controller. While this imposes additional responsibility to the designer, advantages in term of timing predictability and simplicity of implementation are well documented both in academia [22] and industry [9]. Furthermore, a large number of embedded CPU available either as soft or firm IPs for SoC design have configurable memory paths, and can therefore be easily connected to custom-designed scratchpads. Our platform can implement separate instruction and data scratchpads to support Harvard CPU architectures. In particular, our prototype supports the Xilinx Microblaze soft CPU running the Xilkernel OS. We shall assume that the code of all processes and of the OS executed on the CPU fits in the instruction scratchpad, which is typically possible thanks to the small footprint of embedded processes. The data scratchpad is controlled by the interface module: the processor can request it to transfer data from/to any external memory to/from the data scrachpad. The designer specifies the size of both the instruction and data scratchpad as part of the mapping process. Finally, if multiple processes are executed on the same processor, additional isolation requirements are imposed on the CPU and OS. In particular, the CPU must provide memory protection since scratchpads are shared. Furthermore, the OS must provide strong code isolation, especially regarding shared memory structures and libraries. A full description of OS-level isolation techniques is outside the scope of this paper. As an example, the ARINC653 avionic standard [1] prescribes a set of requirements for the safe integration of mixed-criticality applications on a shared CPU, and ARINC653 certified OSs are available on the market.

An important note is relative to the criticality of the various platform blocks. Any fault in the communication infrastructure, external memories and controllers, global modules or in any monitoring wrapper can potentially compromise the whole system. As such, these components must be formally verified and/or certified to the highest degree of safety required by any application executed on the platform. While this can be expensive for the communication infrastructure and memories, we argue that if any form of communication and memory sharing is required in a mixed-criticality system, such requirement can not be avoided. The monitoring wrappers and global modules, in particular the control module and all schedulers, has been designed to be relatively easy to certify. A formal proof of the correctness of generated run-time monitors is available [8], albeit this does not remove the need for certification of the final implemented wrappers, since it is unfeasible to formally verify FPGA implementation tools like the placer and router.

3.1 Global Modules

Three global modules, the Frame Generator, Communi-

[2] Note that since processors can implement external I/O, processes could potentially communicate through the environment. We assume that designers do not intentionally introduce such covert channels, and any environment dependency is solved through application of control theory.



(a) Architecture Block Diagram (b) Interface Module Block Diagram

Figure 3: Pacemaker Platform Instance

cation Scheduler and Power Manager, are included in each platform instance to manage system-wide behavior. Each HW component in our platform can be clocked independently. In particular, each monitoring wrapper employs two distinct clocks: a `system_clk` that is used for the control module and interface module, and has the same frequency for all wrappers; and a `proc_clk` that is used for the processor. In this way, different CPU and HW processors can run at different frequencies. As the integration scale of SoC becomes larger and clock frequency increases, it becomes impossible to synchronize all HW modules based on the same clock because signal propagation delay grows larger compared to the clock period. As such, large ASIC designs are moving towards a Globally-Asynchronous, Locally-Synchronous (GALS) paradigm where individual modules are based on synchronous logic but inter-module communication is asynchronous. Unfortunately, asynchronous communication is ill-suited to safety-critical systems: formally verifying asynchronous systems is order of magnitudes more complex when compared to synchronous design.

To solve this problem, we divide time into fixed-length intervals which we call *frames*. The Frame Generator periodically produces a `frame_signal` that is propagated to all wrappers; the frame period is significantly larger than both the physical clocks (in our prototype, we use a frame period of 1us, while the `system_clk` has a 8ns period) and signal propagation delays. All timing requirements (period, deadline and execution time) expressed in the AADL model are multiples of the frame period, and processors are synchronized based on frames. In particular, processes are periodically activated and communication is initiated on a frame boundary.

The Communication Scheduler uses the frame information to control data transmission on the communication infrastructure. For each process, a `frame_grant` signal is propagated to the corresponding monitoring wrapper: during each frame, the interface module is allowed to transmit on behalf of the process only if the provided `frame_grant` signal is set. Internally, the Communication Scheduler implements a scheduling table based on a fixed length hyperperiod: for each frame in the hyperperiod, the table determines which processes are allowed to transmit. In practice, since a same set of processes could be allowed to transmit for an interval of several frames in a row, the table encodes the length of every such interval. To ensure timing predictability, the table is built to enforce *contentionless communication*: two processes can be allowed to transmit in the same frame only if their data transmissions can be carried out in parallel. In particular, since our prototype employs a shared bus, we allow a single process to transmit in each frame. If for example the communication infrastructure is implemented as a segmented bus connected by bridges, processes using different bus segments could be allowed to transmit simultaneously. Note that apart from timing predictability, the contentionless principle can simplify infrastructure design: for example, wormhole routers with no packet buffers can be used in a NoC [15].

Finally, the Power Manager monitors power consumption. A `clk_en` signal is propagated to each monitoring wrapper. The `proc_clk` is periodically generated only while the `clk_en` is high; therefore, the Power Manager can completely stop a processor by simply lowering the corresponding `clk_en` signal. In our implemented prototype, the Power Manager periodically checks the input voltage to the system using the System Monitor functionality of the Virtex-5 FPGA; in a battery-operated system such as a pacemaker, this can be used to derive an estimate of remaining battery energy. If the energy becomes dangerously low, the Power Manager can shut down the non-life critical systems running on the CPU to save energy for the base pacing module. If additional measurement functionalities were available on the chip, the Power Manager could implement more refined control actions. For example, if power consumption could be measured for each processor, a misbehaving processor consuming an excessive amount of energy could be selectively turned off.

3.2 Interface and Control Module

Figure 3(b) shows a detailed block diagram of the interface module for a HW processor: it is composed of three main submodules, the *Processor Interface*, the *DMA Engine*, and the *Communication Interface*. The Processor Interface is directly connected to the processor and exports the communication services; it uses the same `proc_clk` as the processor. The DMA Engine uses the `system_clk` and implements most of the interface module logic. Finally, the Communication Interface connects to the communication infrastructure and shares its physical clock: it converts data transfer commands issued by the DMA Engine into read/write transactions on the communication infrastructure. This division of concerns simplifies development and certification: a new Communication Interface must be implemented for each communication infrastructure IP, but the DMA Engine is fully reusable. The Communication Interface receives the `frame_grant` as input, and it is allowed to read/write only when the signal is high. In practice, to account for the effect of propagation delay, after `frame_grant` goes high the Communication Interface must wait for a guard time equal to twice the maximum signal propagation delay before it can start to transmit.

Since all three submodules lie in different clock regions, dual-port memory elements are used to connect them. The *data scratchpad* can be simultaneously accessed by both the Processor Interface and DMA Engine. A variable number of *FIFO queues* are used to hold incoming data to message queues; the processor can read from the FIFO queues at any time. Service request by the processor are sent to the *request queue*. There are two types of requests: 1) sending a message to the message queue of another processor; 2) performing either a read (from external memory to data scratchpad) or write (viceversa) DMA operation. A message transfer request contains an ID for the destination queue and the data to be sent, while a DMA request contains the length of the transfer and the base addresses in both external memory and the scratchpad. All transfers are multiple of a 32-bits word. The *completion queue* is used to signal the end of a DMA operation back to the processor. Finally, the *transfer queue* is used to connect the DMA Engine to the Communication Interface.

The DMA Engine processes incoming service requests in order based on the request type. First of all, for every message transfer it translates the queue ID into an address. All external memories and message queues are automatically assigned unique system-wide addresses by our tools. The address is used by the communication infrastructure and Communication Interface to determine the destination of a data transfer. Second, for message transfers and DMA write, the DMA Engine simultaneously moves data from the request queue or the data scratchpad to the transfer queue, and sends the data to the *Run-Time Monitoring Engine* in the control module. The engine is responsible for checking each transferred data word against the functional properties included in the certificate, and it issues either a reject or accept command in 4 clock cycles. If the transfer is rejected, the transfer queue is flushed. Otherwise, the DMA Engine commands the Communication Interface to start the data transfer. The DMA Engine then immediately proceeds to service the next request; in particular, to avoid stalling the Communication Interface, it can write to the transfer queue while a previously accepted data transfer is being carried out. Incoming data from DMA read operations is forwarded to the Run-Time Monitoring Engine but never rejected, hence a reception queue is not required; as explained in Section 4, this is allowed because read operations can not cause side-effects in the system. Similarly, incoming messages are monitored but immediately injected into the corresponding message queue.

The control module for a HW processor is composed of two main submodules. The aforementioned run-time monitoring engine is described in more details in Section 4. The process scheduler receives the `frame_signal` as input and periodically activates the HW process through a `process_start` wire. The process is responsible for signaling the end of its periodic execution through a `process_stop` signal. Failure to do so denotes a critical error; as a consequence, the process is stopped by halting its `proc_clk` clock.

The interface and control module for a CPU processor have some added complexity, mainly because multiple processes can be executed on the same CPU. Message queues, request and completion queues, transfer queues and Run-Time Monitoring Engines are duplicated for each process; our wrapper VHDL description is parametric in the number of processes and our Microblaze implementation supports up to ten. Note that we expect few processes to be mapped on each CPU (typically one per executed application), since each process can have multiple threads. One `frame_grant` signal for each executed process is provided to both the DMA Engine and Communication Interface; they each service the process for which the `frame_grant` is high. The Processor Interface is customized based on the CPU IP; in our implementation for the Microblaze all services are exported through memory-addressable registers. Finally, the process scheduler is replaced with a CPU scheduler. It implements a scheduling table for processes which is similar to the table in the Communication Scheduler. Whenever a different process must be executed, the CPU scheduler interrupts the CPU and communicates the ID of the process. In this way we make sure that computation is synchronized at the frame boundary, while requiring only minimal kernel modification. Note that while we do not support it in our Xilkernel implementation, the OS can implement a two-level scheduler (as required for example by ARINC653), where the time assigned to each process by the CPU Scheduler is distributed to its threads according to a low-level, process-specific software scheduler.

3.3 Implementation Results

We have used our developed tools to automatically generate wrappers based on the AADL pacemaker specification described in Section 2.2. The platform was instantiated and synthesized using Xilinx Embedded Development Kit 10.1.3 on a Virtex-5 VLX50T FPGA. We created both correct and faulty versions of low-criticality processes (in particular, the programmer process) to test the capability of the monitoring wrappers to reject faults. The design behaved according to specification, albeit more stringent testing would be required for certification.

Since propagation delay is less of a concern in current FPGA chips, a single 125Mhz `system_clk` is shared among all wrappers. The VLX50T FPGA can support up to 11 additional clocks for processors and memories; in our pacemaker implementation, the CPU and all HW processors use separate `proc_clk` clocks also running at 125Mhz. The communication infrastructure can transfer up to 4 bytes per cycle, for a total bandwidth of 500MB/s; however, due to guard time, a maximum of 480 bytes can be transferred every 1us frame, resulting in an effective bandwidth of 480MB/s. Finally, we measured an OS footprint of 7648 bytes in instruction and 15234 bytes in data scratchpad. The code for the tools, all manually written module and the generated platform is available online at [33], together with detailed descriptions of hardware interfaces and area utilization for several generated modules.

4. FUNCTIONAL ISOLATION

Our approach to provide functional isolation relies on runtime monitoring. As described in Section 2, for each process the designer specifies a set of formal properties that describe its allowed communication behavior. At run-time, the monitoring engine in the control module checks that all properties are satisfied. If a violation is detected, then the control module can take a suitable *recovery action* to keep the system in a safe state; in particular, any faulty data transfer can be rejected before it is propagated on the communication infrastructure.

The validity of the runtime monitoring approach has been proven in the field of software engineering by a large number of developed tools [20, 16, 5, 25, 3]. However, most of these

```
declarations:
{
  signal numFailures : STD_LOGIC_VECTOR(11 downto 0) := X"000";
}

EventList:
logBufferWrite : write in logBuffer
signalBufferWrite : write in signalBuffer
        { numFailures <= numFailures + 1; }

Formula:
ere: (logBufferWrite)*

violation handler:
{
  reject_reg <= '1';
  if(numFailures >= 1000) then
        reset_reg <= '1';
        numFailures <= (others => '0');
  end if;
}
```

Figure 4: Example Property: SafeMemoryAccess

frameworks have been designed to check the correctness of software applications only and furthermore they run on the same CPU of the monitored process, which can result in significant computation overhead (P2 [23] is an exception since generated monitors run in hardware, but it requires significant hardware modifications to the CPU).

Our toolset is based on Monitoring-Oriented Programming (MOP) ([8] and citations there). MOP is a highly extensible and configurable runtime verification framework. The user is allowed to extend MOP with his/her own logics via *logic plugins* which encapsulate the monitor synthesis algorithms. Currently, we support both Past-Time Linear Temporal Logic (PTLTL) and Extended Regular Expressions (ERE). Property specification consists of event definitions and logical formulae or patterns. The formula or pattern designates which "traces" (observed series of events) are valid or invalid. Recovery can be initiated either when a validation or violation of the trace is detected. For EREs, valid traces are those which are strings in the language represented by the ERE, with events treated as the letters in the alphabet of the language. Neutral traces are prefixes of strings in the language, while violations are invalid strings. For PTLTL formulae, valid traces are any traces for which the formula evaluates to true, invalid traces are those for which the formula evaluates to false; there are no neutral traces. For more information on regular languages and temporal logic see [24] and [11], respectively.

Illustrative Example. An example of property specification for the Event Tagger process can be seen in Figure 4. Recall from Section 2 that two data objects are placed in external memory: the signalBuffer and the logBuffer. The property, called *SafeMemoryAccess*, makes sure that the Event Tagger can not erroneously overwrite the data written by the critical Pacer module in the signalBuffer. Each property is comprised of a formula, an `EventList`, a declaration section and validation/violation handlers. In this example, the declaration section simply introduces one monitor-local register that keeps track of the number of erroneously attempted writes. The two events, `signalBufferWrite` and `logBufferWrite`, capture writes to the signalBuffer and logBuffer respectively. The ERE pattern matches zero or more occurrences of the `logBufferWrite` event; hence, any write to signalBuffer will cause a violation. Recovery is specified in the violation handler: the data transfer is rejected. Furthermore, after 1000 erroneous writes the process is reset.

We have first applied MOP to generate hardware monitors in a mixed VHDL/Verilog RTL description in [27]. In particular, we have introduced a customized MOP instance, BusMOP, that can be used to monitor the correctness of COTS peripherals connected to the PCI bus. As described in [27], our tool generates a VHDL module for each property. We have developed new logic that connects all generated property modules together in the Monitoring Engine. Compared to [27], our implementation has two important novelties. First of all, BusMOP could only recover through corrective actions (in particular, overwriting memory locations): this is because data transfers could not be prevented from being propagated on the PCI bus. In our SoC platform, faulty data transfers can be rejected in the interface module. Second, since properties are specified in the AADL model, events can be expressed using the symbolic names of message queues and data objects; our tool then automatically translates from names to corresponding addresses. In BusMOP, the designer had to manually specify all memory addresses, which is error prone. In the remaining of this section, we first detail the event syntax and discuss available recovery actions; we refer the interested reader to [27] for a description of the ERE and PTLTL syntax. We conclude by showing a more complex property example.

Event Specification. A formal description of the event syntax (using Backus Naur Form (BNF) [21] extended with $[p]$ and $\{p\}$, denoting zero or one repetitions of p and zero or more repetitions of p, respectively) can be seen below, where $\langle ID \rangle$ represents a symbolic name, $\langle Number \rangle$ a constant number in VHDL notation, and $\langle ArithmeticExp \rangle$ an arithmetic expression:

$$
\begin{array}{rcl}
\langle \mathit{Event} \rangle & ::= & \langle \mathit{ID} \rangle : \langle \mathit{Expression} \rangle \\
\langle \mathit{Expression} \rangle & ::= & \langle \mathit{ReadOrWrite} \rangle \text{“}\mathit{in}\text{”} \langle \mathit{ID} \rangle [\langle \mathit{Action} \rangle] \\
 & | & \langle \mathit{ReadOrWrite} \rangle \text{“}\mathit{at}\text{”} \langle \mathit{ID} \rangle [\text{“}+\text{”} \langle \mathit{Number} \rangle] \\
 & & [\text{“}\mathit{value}\text{”} [\text{“}\mathit{not}\text{”}] \text{“}\mathit{in}\text{”} \langle \mathit{Range} \rangle][\langle \mathit{Action} \rangle] \\
\langle \mathit{ReadOrWrite} \rangle & ::= & \text{“}\mathit{read}\text{”} \quad | \quad \text{“}\mathit{write}\text{”} \\
\langle \mathit{Action} \rangle & ::= & \text{“”} \langle \mathit{Arbitrary\ VHDL\ code} \rangle \text{“”} \\
\langle \mathit{Range} \rangle & ::= & \langle \mathit{ArithmeticExp} \rangle [\text{“},\text{”} \langle \mathit{ArithmeticExp} \rangle]
\end{array}
$$

All events are read/writes from/to either a message queue or a data element in external memory, identified by its name in the AADL model. Normally, an event is triggered whenever any word of a data element is changed. However, the designer can specify a numerical address inside the data element, in which case the event is triggered only when that specific memory location is read/written. This can useful to access individual components of a complex data structure. In this case, the designer can also specify a desired value range. Ranges can consist of a single arithmetic expression, or a pair of comma separated arithmetic expressions denoting the minimum and maximum values that can trigger the event. Since all transfers in the system are multiple of the word size, values are always assumed to be 32-bits.

Recovery. Recovery actions in the handlers are specified as a list of concurrent VHDL statements. Several recovery actions are possible in the handler: 1) the current data transfer can be rejected in the interface module; 2) the monitored process can be stopped. In a HW processor, this is achieved by stopping the `proc_clk`. In a CPU, the output of the CPU Scheduler is altered so that the processor is never executed; 3) the process can be reset. The reset functionality depends on the design of the processor. HW processors implemented on FPGA use synchronous logic and are provided with a reset signal from the control module. In our Microblaze implementation, the control module can interrupt the CPU and signal the reset action. Xilkernel can then kill the process and restart it. 4) The DMA Engine can be instructed to carry out a send to a message queue or a write to external memory. Monitor requests take precedence over all other data transfer requests, but they can still be carried out only during a frame assigned to the process by the Com-

```
checkPacer   : write at Pacer.parameters       value in X"40000000"
checkRate    : write at RateAdapter.parameters value in X"40000000"
successPacer : read  at Programmer.response    value in X"80000000"
successRate  : read  at Programmer.response    value in X"20000000"
failurePacer : read  at Programmer.response    value in X"40000000"
failureRate  : read  at Programmer.response    value in X"10000000"
commitPacer  : write at Pacer.parameters       value in X"80000000"
commitRate   : write at RateAdapter.parameters value in X"80000000"
```

Figure 5: Programmer Event List

```
Formula:
1. ptltl : successPacer and <*>successRate and
2.   (*)( not (not(successRate) S successPacer) ) and
3.   (*)( not (not(successRate) S failureRate) )

Validation handler:
{
  address_reg <= Pacer.parameters;
  value_reg <= X"8000000";
  execute_reg <= '1';
}
```

Figure 6: SendPacerCommit Property

munication Scheduler. The set of registers described below is used to specify the recovery action in VHDL; symbolic names can be used in place of addresses.

	Data Transfer Interface
address_reg	address
value_reg	value send/written
execute_reg	start transfer
reject_reg	reject transfer
stop_reg	stop process
reset_reg	reset process

Note that only outgoing data transfers can be rejected; incoming data is always accepted. By checking all outgoing transfers, we always make sure that any data propagated on the communication infrastructure conforms to specified certificates. Hence, incoming data must also conform to system specification. Furthermore, since timing isolation is always enforced, DMA read operations can not steal resources or change the state of other processes.

Programmer Process. We now describe a more detailed example to show how monitors can be directly exploited to enforce correct communication between lower and higher criticality applications. As described in Section 2, the Programmer process is used to update execution parameters of both the Pacer and Rate Adapter based on received RF commands. This is potentially dangerous, because both the Pacer and Rate Adapter are more critical than the Programmer and RF process. To solve the problem, we can introduce a commit protocol. The Programmer sends new parameters followed by a `check` command to the Rate Adapter through the *RateAdapter.parameters* message queue. The Rate Adapter validates the received parameters and sends back either a `success` or a `failure` answer to the *Programmer.response* message queue. The Programmer then repeats the same steps for the Pacer using its *Pacer.parameters* queue. If the Programmer receives `success` answers from both processes, it then sends `commit` messages to the Rate Adapter and Pacer process, causing them to load the received parameters. Unfortunately, since the Programmer is a complex, non safety-critical process, it could fail after sending a commit command to just one of the two processes. While this would not compromise the life-critical functionality (the Pacer control algorithm rejects any unsafe control points [4]), it can nevertheless disrupt the Rate Adapter causing significant discomfort to the patient.

The solution that we adopt is to send the `commit` command directly from the monitor; in this way, we isolate the critical functionality of the Programmer module inside the certified wrapper. A set of Programmer events are specified in Figure 5, consisting of `check` and `commit` commands and `success` and `failure` answers for both the Pacer and the Rate Adapter. The *SendPacerCommit* property in Figure 6 is used to send the `commit` command to the Pacer on validation (an equivalent property with different handler can be used to send the `commit` to the Rate Adapter). In the PTLTL formula, `<*>`, `(*)` and `S` are temporal operators denoting *eventually in the past*, *previously* and *since*. Line 1 specifies that a `success` is received from the Pacer in the present and a `success` from the Rate Adapter has been received in the past. Line 2 implies that at least one `successRate` event has been received since the previous `successPacer` (if any), and Line 3 implies that at least one `successRate` has been received since the previous `failureRate` (if any); this makes sure that a valid parameter set has been passed to the Rate Adapter since the last commit operation. Finally, property *CheckProgrammerCommands* in Figure 7 is used to reject any erroneous Programmer command. Line 1 is used to reject `commit` commands from the Programmer, since only the monitor should send them. Lines 2-7 make sure that the Programmer can not send a `check` command if it has not received an answer (either a `success` or a `failure`) to its previous `check` commands to both the Pacer and Rate Adapter: otherwise, the Programmer could send a `check` command immediately before a `commit` is sent by the monitor, causing a wrong set of parameters to be loaded.

```
Formula:
1. ptltl : commitRate or commitPacer or (
2.   (checkRate or checkPacer) and
3.   (*)(
4.     (not(successRate or failureRate) S checkRate)  or
5.     (not(successPacer or failurePacer) S checkPacer)
6.   )
7. )

Validation handler:
{ reject_reg <= '1'; }
```

Figure 7: CheckProgrammerCommands Property

5. TIMING ISOLATION

As described in Section 2, each process certificate specifies a timing reservation for both computation and communication. More in details, each process expresses a *computation request* $\{e, p, d\}$, where e is the execution time, p the period and d the relative deadline specified in the AADL model. Furthermore, if a process defines N communication channels, then it expresses N *communication requests* of the form $\{e_i, p, d_i\}$, where p is the period of the process itself, d_i is the specified communication deadline for the i-th communication channel, and e_i represents its required transmission time in number of frames; e_i is obtained by dividing the data size by the communication infrastructure bandwidth and adding a constant term representing memory access delay and any per-transfer overhead (up to 40ns in our implementation), then rounding up to the frame size. All deadlines must be less or equal to periods.

Timing isolation is provided as follows: each computation request is treated as a periodic task to be scheduled on the assigned processor, and each communication request as a periodic task scheduled on the communication infrastructure. Our tool then generates all scheduling tables by computing

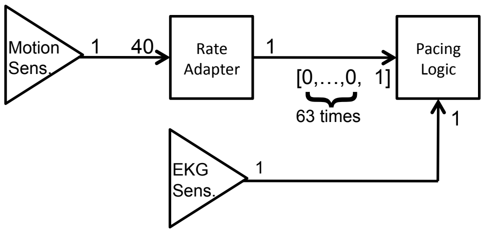


Figure 8: Example Dataflow Communication

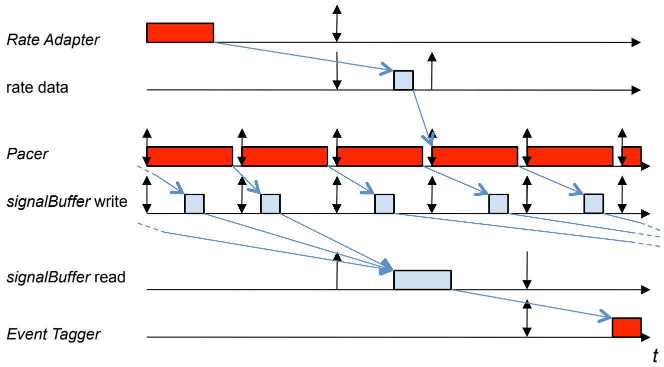


Figure 9: Example Schedule

implicit-EDF schedules [7]: for each CPU and the communication infrastructure, an Earliest-Deadline-First schedule is simulated for an entire hyperperiod (the minimum common multiple of process periods in number of frames), and the process executed during each frame is added to the corresponding table. This static scheduling approach has a key advantage for safety-critical systems: when verifying the correctness of the whole system, it is not necessary to model the scheduling algorithm. Instead, the generated scheduling tables can be used as input to the verification system. In this way as an example, a model checker would only need to check the system behavior for one hyperperiod, since the same exact schedule will be repeated thereafter. As a disadvantage, scheduling tables can potentially grow quite large if process periods are not homogenous; however, we argue that for most system designs, this is not the case (for example, in dataflow models, process periods depend on the ratio between the number of sent and received tokens).

Note that our reservation model does not necessarily restricts the model of computation used by each application. In particular, both time-driven and event-driven activation models can be mapped to our specification, albeit in the second case it can lead to over-reservation. As an example, consider the pacemaker subsystem composed of the Motion Sensor, Rate Adapter, EKG Sensor and Pacing Logic. At a higher-level, this subsystem could be modeled as the cyclo-static dataflow represented in Figure 8, where the Pacing Logic consumes one token from the EKG Sensor every activation and one token from the Rate Adapter every 64 cycles. Based on this representation, periods of 7,810us for the Pacer process (including the EKG Sensor and Pacing Logic), 499,840us for the Rate Adapter and 12,496us for the Motion Sensor can be synthesized, which are within 0.1% of the frequencies (128Hz, 2Hz and 80Hz) specified in Section 2.2 for these processes.

A possible generated schedule for the Rate Adapter, Pacer and Event Tagger processes is shown in Figure 9, where up arrows represent periodic activation times and down arrows represent absolute deadlines. Slanted arrow are used to connect periodic process instances with the data they generate and consume. We represent a single instance of Rate Adapter and Event Tagger because their periods are significantly larger than the Pacer period (64 and 16 times larger respectively); furthermore, note that in reality communication reservations are much smaller than represented, since the size of the exchanged data is very small. Every period, the Event Tagger reads from the signalBuffer the data written by the previous 16 instances of Pacer. Note that the amount of written data is not constant: the Pacer writes EKG data every period, but a Shock Event is written only when the heart is paced. The Pacer communication request must be nevertheless sufficient to send both EKG and Shock data every period. To minimize end-to-end delay, the activation time of each communication task for outgoing data is set to be equal to the deadline of the corresponding process; vice versa, the deadline for a DMA read operation corresponds to the activation time of the next process instance. Furthermore, during mapping the designer can specify an initial activation time (*phase*) for each process; for example, in Figure 9 this is used to make sure that the activation time of Pacer corresponds with the deadline of the rate data sent by the Rate Adapter.

A final consideration is relative to queue size. Both the request queue and all FIFO queues in the interface module must be large enough to avoid overflow; if a queue is full, any incoming data is dropped. In general, required queue size depends on the internal behavior of a process (e.g. when it consumes data in the queue): therefore, the designer must specify minimum queue sizes in the AADL model and validate them for critical processes. As an example, consider the interaction between the Motion Sensor and the Rate Adapter. If the activation time for the Rate Adapter corresponds to the communication deadline for the Motion Sensor and furthermore the Rate Adapter consumes all data from the queue immediately upon activation, the minimum size of 40 samples is sufficient. However, if the time at which the Rate Adapter consumes the data is unknown, a size of up to 80 samples may be required (in general, it is possible to show that given periods p_i, p_j for a producer and consumer process respectively, a worst case size of $\lceil \frac{2p_j}{p_i} \rceil + 1$ samples is required assuming that the consumer reads all available data every period [33]).

6. CONCLUSIONS

Implementing safety-critical embedded systems like medical devices as Systems-on-Chip is promising, but unfortunately there is a lack of suitable design methodologies. Integrating mixed-criticality systems is especially challenging because low-criticality IPs are too expensive to be verified and/or certified, but they must be prevented from interfering with high-criticality applications. To solve this problem, in this paper we have introduced a new design methodology and architectural platform. The key idea is that our platform supports behavioral enforcement: low-criticality processes are guaranteed to behave according to a published certificate. This does not remove the need to certify high-criticality components and verify correct system-level behavior, but it enables the design to do so without worrying about unpredictable faults in low-criticality components.

As future work, we plan to extend our platform in several directions. First of all, we would like to integrate our tools with a model checking framework like Maude [13] or UPPAAL [34]; extracted certificates should be directly exported to the model-checking tool, facilitating system-level verification. Second, we believe that some formal properties, like the one in Figure 4, could be automatically extracted from the AADL model without designer intervention, especially if more precise behavioral information is added to each process (for example, AADL has been recently extended with a behavioral annex). Third, we plan to significantly extend our scheduling mechanisms to support parallel communication infrastructures and NoCs, and to be able to specify more general timing reservations to reduce over-reservation.

7. REFERENCES

[1] Aeronautical Radio Inc. *ARINC 653 Specification.* http://www.arinc.com/.

[2] B. Akesson, K. Goossens, and M. Ringhofer. Predator: a predictable sdram memory controller. In *Proc. of the 5th IEEE/ACM inter. conference on Hardware/software codesign and system synthesis*, 2007.

[3] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*, pages 589–608, 2007.

[4] S. Bak, D. Chivukula, O. Adekunle, M. Sun, M. Caccamo, and L. Sha. The system-level simplex architecture for improved real-time embedded system safety. In *Proceedings of the IEEE RTAS*, 2009.

[5] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'04)*, pages 277–306, 2004.

[6] Pam Binns, Matt Englehart, Mike Jackson, and Steve Vestal. Domain Specific Software Architectures for Guidance, Navigation and Control. *International Journal of Software Engineering and Knowledge Engineering*, 6(2):201–227, 1996.

[7] M. Caccamo, L. Y. Zhang, L. Sha, and G. Buttazzo. An implicit prioritized access protocol for wireless sensor networks. In *Proceedings of the IEEE RTSS*, Dec 2002.

[8] F. Chen and G. Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *Proc. of the ACM OOPSLA*, pages 569–588, 2007.

[9] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation: A performance view. Technical report, IBM Research, 2005.

[10] P. Cumming. The TI OMAP platform approach to SoCs. In *Surviving the SoC revolution: A guide to platform-based design.* Kluwer, 1999.

[11] E.A. Emerson. *Handbook of Theoretical Computer Science.* MIT Press, 1990. Chapter 16: Temporal and modal logic.

[12] F. Balarin et al. Metropolis: An integrated electronic system design environment. *IEEE Computers*, 36(4):45–52, 2003.

[13] M. Clavel et al. The maude 2.0 system. In *Proc. Rewriting Techniques and Applications*, 2003.

[14] P.H. Feiler, B.A. Lewis, and S. Vestal. The SAE Architecture Analysis & Design Language (AADL): A Standard for Engineering Performance Critical Systems. In *Proc, of the 2006 IEEE Conference on Computer Aided Control Systems Design*, Oct. 2006.

[15] K. Goossens, J. Dielissen, and A. Radulescu. őthereal network on chip: Concepts, architectures, and implementations. *IEEE Design and Test*, 22(5):414–421, 2005.

[16] K. Havelund and G. Rosu. Monitoring Java programs with Java pathexplorer. In *Proc. First Workshop on Runtime Verification*, 2001.

[17] J. Hugues, B. Zalila, L. Pautet, and F. Kordon. From the prototype to the final embedded system using the Ocarina AADL tool suite. *ACM Trans. on Emb. Computing Sys.*, 7(4):1–25, 2008.

[18] IBM. *Processor Local Bus Specification.* http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/3BBB27E5BCC165BA87256A2B0064FFB4.

[19] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, 2003.

[20] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-mac: A run-time assurance approach for Java programs. *Formal Methods in System Design*, 24(2):129–155, 2004.

[21] D. Knuth. Backus normal form vs. backus naur form. *Communications of the ACM*, 7(12):735–736, 1964.

[22] B. Lickly, I. Liu, S. Kim, H. Patel, S. Edwards, and E. Lee. Predictable programming on a precision timed architecture. In *Proceedings of International Conference on Compilers, Architecture, and Synthesis from Embedded Systems*, Oct 2008.

[23] H. Lu and A. Forin. The design and implementation of p2v, an architecture for zero-overhead online verification of software programs. Technical Report MSR-TR-2007-99, Microsoft Research, 2007.

[24] M. Sipser. *Introduction to the Theory of Computation.* PWS Publishing, 1996. Chapter 1: Regular Languages.

[25] M. Martin, B. Livshits, and M. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 365–383, 2005.

[26] NXP Semiconductors. *Philips Nexperia Digital Video Platform.* http://www.nxp.com.

[27] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu. Hardware runtime monitoring for dependable COTS-based real-time embedded systems. In *Proceedings of the 29th IEEE RTSS*, Dec 2008.

[28] A.-E. Rugina, K. Kanoun, and M. Kaaniche. The ADAPT Tool: From AADL Architectural Models to Stochastic Petri Nets through Model Transformation. In *Dependable Computing Conference, 2008. EDCC 2008. Seventh European*, pages 85–90, May 2008.

[29] A. Sangiovanni-Vincentelli. Quo vadis, SLD? reasoning about the trends and challenges of system level design. *Proceedings of the IEEE*, 95(3):467–506, 2007.

[30] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Scheduling and memory requirements analysis with AADL. In *Proceedings of ACM SIGAda*, volume 25, pages 1–10, Atlanta, Georgia, 2005.

[31] O. Sokolsky, I. Lee, and D. Clarke. Schedulability analysis of AADL models. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, April 2006.

[32] H. Sun, M. Hauptman, and R. Lutz. Integrating product-line fault tree analysis into aadl models. In *High Assurance Systems Engineering Symposium, 2007. HASE '07. 10th IEEE*, pages 15–22, 2007.

[33] University of Illinois. *SoC Mixed-Criticality Platform Website and Repository.* http://netfiles.uiuc.edu/rpelliz2/www/soc.html.

[34] Uppsala University and Aalborg Univeristy. *Uppaal a tool suite for verification of real-time systems.* www.uppaal.com.

[35] John G. Webster. *Cardiac Pacemakers.* IEEE Press, 1993.

[36] Xilinx, Inc. *Virtex-5 User Guide.* Available at www.xilinx.com.