CISTER
Research Centre in
Real-Time & Embedded
Computing Systems

Conference Paper

Emulation-in-the-loop for simulation and testing of real-time critical CPS

Renato Oliveira*
Manuel Meireles
Cláudio Maia*
Luis Miguel Pinho*

*CISTER Research Centre

CISTER-TR-180501

2018/05/15

Emulation-in-the-loop for simulation and testing of real-time critical CPS

Renato Oliveira*, Manuel Meireles, Cláudio Maia*, Luis Miguel Pinho*

*CISTER Research Centre

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: prmol@isep.ipp.pt, mjcdm@isep.ipp.pt, clrrm@isep.ipp.pt, lmp@isep.ipp.pt

http://www.cister.isep.ipp.pt

Abstract

Complex cyber-physical systems are more and more a set of components working tightly coupled, with little or no human intervention. Assessing the correctness of these systems by testing components individually, one-by-one, is obviously not sufficient, being required to also test and validate the overall system. KhronoSim is a modular and extensible platform for testing cyber-physical systems in closed-loop, which enables the integration of simulation models and platform emulators to build a closed loop test environment. This paper presents the emulator module of KhronoSim, developed to integrate the well-known QEMU emulator in the closed-loop testing platform.

© CISTER Research Center
www.cister.isep.ipp.pt

Emulation-in-the-loop for simulation and testing of real-time critical CPS

Paulo Renato Oliveira, Manuel Meireles,
Cláudio Maia, Luis Miguel Pinho
CISTER Research Centre
School of Engineering of the Polytechnic Institute of Porto
Porto, Portugal

Gonçalo Gouveia, João Esteves
Critical Software
Coimbra, Portugal

Abstract— **Complex cyber-physical systems are more and more a set of components working tightly coupled, with little or no human intervention. Assessing the correctness of these systems by testing components individually, one-by-one, is obviously not sufficient, being required to also test and validate the overall system. KhronoSim is a modular and extensible platform for testing cyber-physical systems in closed-loop, which enables the integration of simulation models and platform emulators to build a closed loop test environment. This paper presents the emulator module of KhronoSim, developed to integrate the well-known QEMU emulator in the closed-loop testing platform.**

Keywords—CPS; Emulation; Testing

I. Introduction

Complex systems and systems of systems are an integrated set of components and sub-systems, tightly interacting together to achieve a specific goal. While guaranteeing that individual sub-systems behave according to their specifications is a (relatively) "simple" task, the magnitude of the validation largely increases when it comes to providing guarantees on the correct integrated behaviour. As a matter of fact, all the possible interactions between the sub-systems must be properly tested in order to capture all the system properties. In addition, testing a system in its actual environment of operation can be overly expensive and/or too slow, in particular when considering Cyber-Physical Systems (CPS), which are known by their interaction with specific physical environments.

The use of model and platform simulators is spreading around and growing in importance to address the aforementioned issues. Simulators allow for an increase in the productivity of software development, enabling three key features:

1. simultaneous development of software and hardware;
2. testing software before actual hardware exists; and
3. providing an environment for software testing, without requiring actual hardware-in-the-loop.

As it can be inferred from the third feature, an abstracted model of the underlying hardware platform is key for the software development. This is usually done by creating a software version of the target platform that (completely) mimics its behavior, i.e., an emulator.

In this context, KhronoSim [1] is a modular and extensible system for simulation and test of complex systems that enables the integration of simulation models and platform emulators in a closed-loop test environment consisting of physical and virtual systems. The main objectives of KhronoSim are:

- to make it possible to simulate complex systems in real-time by including either the whole or part of the system under test, including the simulation of the environment and other interacting systems;
- to allow for a distributed testing architecture, supporting real-time stimuli from a scalable number of inputs, distributed over several systems;
- to make it possible to emulate the hardware platform upon which the system will execute, specifically by assuming that it is a multicore embedded platform.

This latter objective is also of paramount importance, as one of the goals of KhronoSim is to be able to test multicore hardware platforms, in several system configurations and thus providing a better understanding of the design choices. This will accelerate the certification and the development process in the aerospace industry (e.g., for real-time operating systems (RTOS), mixed-criticality systems (MCS), etc.) and it will provide a further insight into the latest developed methods and techniques to solve key challenges, which are inherent to multiprocessor systems and continue to raise concerns to certification authorities [2] [3] (e.g., the application-to-core mapping, temporal and spatial protection mechanisms, etc.).

In order to support hardware emulation in the loop, a specific module was developed to integrate the well-known QEMU emulator in KhronoSim. This paper discusses the different emulator choices considered and describes the main architectural features of the developed module as well as the addition of the capability to control the speed of emulation.

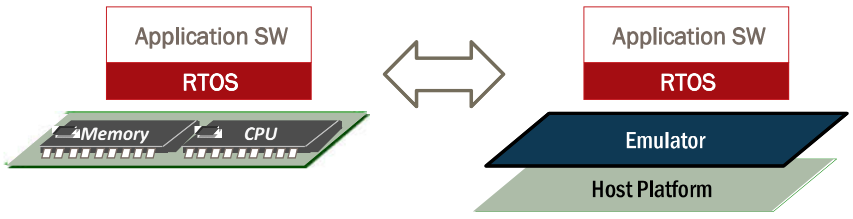

Figure 1 – Hardware Emulation

The paper is structured as follows. Section II briefly discusses the features and advantages of hardware emulation, whilst section III reviews the emulation options considered in KhronoSim. Afterwards, Section IV provides the architecture and features of the emulator module. Finally, Section V concludes the paper.

II. Emulation

Emulation [1] is the process of having both the interface and the functionality of a system (the Guest) implemented on top of another system (the Host) with a different interface and functionality [4]. It thus follows that an emulator is a piece of software that executes an application or a set of applications that were initially built for a different computer architecture than the host platform upon which they are executing [5] (Figure 1).

Broadly speaking, we can classify emulators in two distinct categories, depending on the level of details required during their implementation: (1) instruction set emulators; and (2) functional emulators.

In the first category, the emulator behaves identically as the emulated platform (also known as target platform), on an instruction level. These emulators are predominantly time-driven and cycle-accurate, with the interpretation that each clock cycle of the target platform has to be faithfully simulated. This approach requires that the application source code (and Operating System) are cross-compiled to the target machine code and then interpreted by the emulator.

The advantage of this approach is that the performed execution will match the execution on the target . In addition, the timing properties of the execution will also be maintained at the cycle level, an important issue in the domain of real-time systems, where it is mandatory to provide guarantees for a correct timing behaviour of the system. The disadvantage of this kind of emulators is that they are exposed to a performance penalty: as the hardware becomes more complex, maintaining a reasonable ratio between the simulated time and the actual time becomes increasingly expensive. From these drawbacks, it follows that this type of simulators is inherently expensive and very difficult to develop.

In the second category, the emulator imitates the behaviour of the target platform only at a functional level without trying to exactly match the system behaviour, by exposing only the same interface. These emulators are mostly event-driven, with the interpretation that, at times, they can work with the granularity significantly larger than the target platform cycle. One sub-category of these is called host-compiled emulators, where functions, execution blocks and sets of instructions of the target platform, are converted into the corresponding entities of the host. The advantage is that they are faster than instruction set emulators, and they are easier to deploy. However, an important disadvantage is that the timing properties are not faithfully replicated, as execution is performed on the native platform in its own environment, which may be significantly different from the corresponding environment of the actual target platform. This means that functions can take arbitrarily different execution time.

Using a host compiled system means that the application source code and Operating System are compiled using the native host compiler, as opposed to a cross-compiler in the former approach. Since it uses a native compiler, the resulting executable does not need to be interpreted and therefore it can reach much higher levels of performance. The downside is the inaccuracy in the number of clock cycles required since the assembly code generated by the host compiler does not match with the code in the actual platform. This situation represents a huge hurdle for system designers when it comes to provide timing guarantees for the system as it renders their work of providing guarantees for a correct timing behaviour during execution very difficult, if not impossible.

A. *Instruction set emulation techniques*

Instruction set emulation is the process of converting the binary data written for a given instruction set to a host processor with a different instruction set. This conversion is usually done by taking the original binary instruction and translating it into one or more equivalent instructions for execution on the host processor. As a 1:1 instruction conversion rate is usually difficult, if not impossible to achieve, the emulator's equivalent to the original program is often much larger. The lower the overheads of the conversion, the faster the emulated software will execute. There are various techniques for the emulation of classical instruction sets and we focus on two of the most popular ones, namely interpretation and binary translation:

- Interpretation. This method is the easiest to implement, but also the slowest in terms of execution time. Here, the binary data is read and each instruction is decoded, translated and executed. It is required from the interpreter to step through each instruction of the source program, to

[1] In this paper we do not discuss the differences between emulation and simulation. Since the hardware platform internals are being abstracted away we consider only the use of emulation.

read it and modify the state of the program accordingly. Then, it is required from the interpretation routine to emulate this instruction on the target instruction set, before the next instruction is fetched.

- Binary translation. This emulation method (also called recompilation), involves a directly binary conversion from the binary data for the emulated instruction set into binary data suitable for execution on the host instruction set. Blocks of source instructions are converted into instructions that are equivalent in terms of functionality. From this viewpoint, this method provides a significant performance boost. Recompilation typically comes in two variants: (1) dynamic recompilation – In this variant, the binary data is translated only on the first pass and is kept in a cache where the translated binary equivalent is stored and referenced whenever that section is executed again; and the (2) static recompilation – In this variant, the binary data is translated once in a single pass over the code. The code to be emulated can be scanned and the translation can be optimized by applying various algorithms. The translated data is then usually saved either in a file or memory, where it is referenced by the program upon execution. While this can sometimes improve performance even over dynamic recompilation, if the code is self-modifying, this method usually fails to properly translate the executable, thus forcing a fall back to a dynamic recompiler or an interpreter to handle the modified code.

In summary, interpretation has a nearly zero start-up overheads but suffers from a slower emulation of the instructions themselves. In the other hand, binary translation suffers instead from a slow start-up due to the need of interpreting and translating the operations. However, once these operations are translated, their repeated execution becomes much faster.

III. EMULATION TOOLS

In this section we provide a non-exhaustive analysis of the emulator tools which were considered for the KhronoSim platform. Since one of the main targets of emulation is the LEON platform (for space applications), a sub-section is dedicated to existent emulators for this processor. Higher details are provided for QEMU, which was selected for the platform due to its support to a multitude of devices, stability, community support, and throttle support.

A. EMUL8

Emul8 [6] is a recent Instruction Set Simulator developed in C#. Currently, the following architectures are supported: ARM Cortex-A and Cortex-M, SPARC, PowerPC and an experimental version of x86. Emul8 also supports several peripherals such as USB, I2C, SPI, etc. including those that connect directly to the system bus. It can be installed under Mac OS and Linux (using Mono 4.6 or newer). Concerning its license, Emul8 is released under the MIT license.

B. GEM5

Gem5 [7] is a simulator that can be used for computer architecture research, both for system level architecture as well as microprocessor architecture. Gem5 provides interpretation based-CPU models, for instance in-order and out-of-order CPUs, and an event-driven memory system that includes caches, crossbars and a DRAM controller model. Currently, it supports the following architectures: Alpha, ARM, SPARC, MIPS, POWER, RISC-V and x86 and it executes on top of Mac OS X, Linux, etc. gem5 is released under a BSD-style open source license.

C. BOCHS

Bochs [8] is an open-source emulator for the x86 architectures, which is written in C++ and licensed under the lesser GNU Public License (GPL). With this emulator, an entire PC platform, including an Intel x86 CPU and several input/output devices such as memory, display, hard disks, network, among others can be emulated. The x86 CPUs that can be emulated notably includes: 386, 486, Pentium and x86-64 (with some Intel and AMD extensions supported as well). Bochs can execute a variety of x86 operating systems, such as: Windows 95/98/NT/2000/XP and Vista, Linux and BSD.

Bochs can be compiled to support SMP (Symmetric multiprocessing) so that it emulates multiprocessors with up to 254 processor threads. However, this feature is still experimental and Bochs does not use threading and consequently executes sequentially even when the host is a multiprocessor hardware. This emulator does not support dynamic translation nor virtualization. Instead, each x86/x86-64 instruction is directly interpreted. During the execution phase, Bochs adopts a lazy approach to update the registers and flags, that is, registers and flags are updated only when it is necessary to do so [9].

D. Emulation Tools for LEON

LEON is a synthesizable VHDL (VHSIC Hardware Description Language) model of 32-bit microprocessors compliant with SPARC V8 RISC (Reduced Instruction Set Computer) architectures, developed by Cobham Gaisler. Currently, LEON is at its fourth version - LEON4. In comparison to the previous versions, this version 4 supports a number of features among the following are noticeable: a 7-stage pipeline with branch prediction and 64-bit internal load/store data paths; optional L2 cache support; AMBA (Advanced Microcontroller Bus Architecture) interface of either 64/128-bits; and SMP support (this feature was already available in LEON3) [10] [11].

1) GRSIM

GRSIM [12] is an event-driven tool developed by Cobham Gaisler, which simulates devices based on the AMBA bus. This tool is suitable for simulating LEON3 and LEON4 systems with multiprocessor configurations and can be used as a standalone application or as a thread-safe library to be integrated into a larger simulating framework. The advantage of this tool is that it fully supports read and write accesses to all LEON registers and memory. Also, it offers a debugger and disassembler. Currently it executes on top of Linux or Windows. Regarding its license policy, GRSIM requires a HASP USB hardware key with proper device drivers installed.

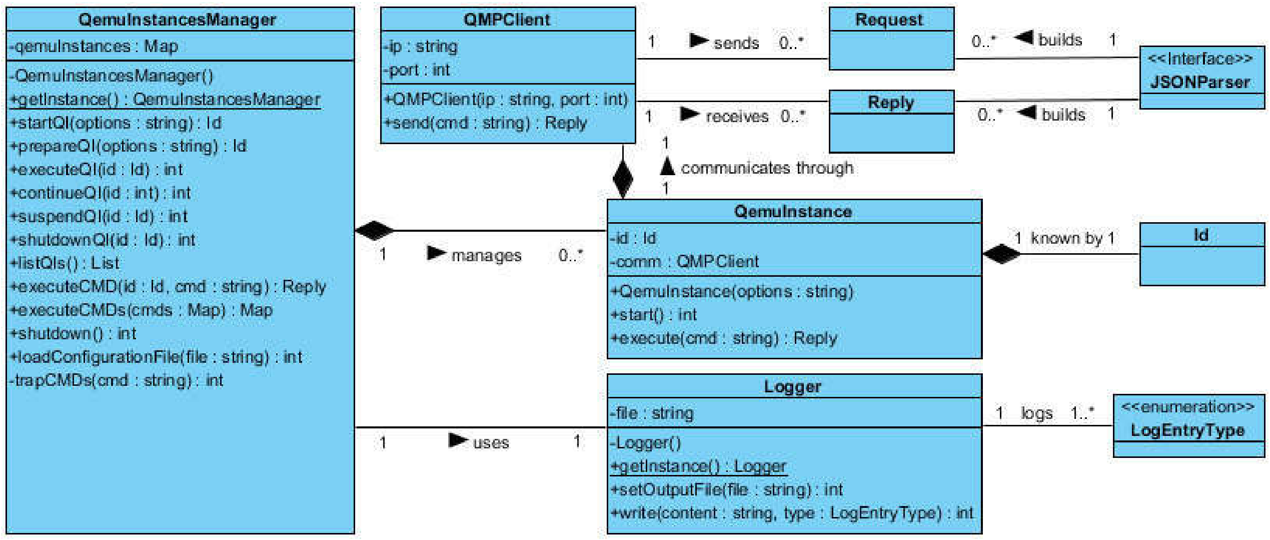
QemuInstancesManager
-qemuInstances : Map
-QemuInstancesManager()
+getInstance() : QemuInstancesManager
+startQI(options : string) : Id
+prepareQI(options : string) : Id
+executeQI(id : Id) : int
+continueQI(id : int) : int
+suspendQI(id : Id) : int
+shutdownQI(id : Id) : int
+listQIs() : List
+executeCMD(id : Id, cmd : string) : Reply
+executeCMDs(cmds : Map) : Map
+shutdown() : int
+loadConfigurationFile(file : string) : int
-trapCMDs(cmd : string) : int
QMPClient
-ip : string
-port : int
+QMPClient(ip : string, port : int)
+send(cmd : string) : Reply
1 sends 0..*
Request
0..* builds 1
<<Interface>>
JSONParser
1 receives 0..*
Reply
0..* builds 1
1
communicates through
1
1 manages 0..*
QemuInstance
-id : Id
-comm : QMPClient
+QemuInstance(options : string)
+start() : int
+execute(cmd : string) : Reply
1 known by 1
Id
Logger
-file : string
-Logger()
+getInstance() : Logger
+setOutputFile(file : string) : int
+write(content : string, type : LogEntryType) : int
1 uses 1
1 logs 1..*
<<enumeration>>
LogEntryType

Figure 2 – KhronoSim QEMU Module

2) T-EMU

The Terma Emulator, in short T-EMU [13], is a tool provided by TERMA for the emulation of microprocessors, including multicores, based on the LLVM framework. Currently, it supports the SPARC V8 processors namely, ERC32, LEON2, LEON3 and LEON4. T-EMU can also emulate a full system by adding memory and peripheral emulation to processor emulation. Devices are written as plugins that can be connected together in order to be possible to emulate a certain system using a specific configuration. Several device models and bus models (e.g., AMBA, CAN, etc.) are supported. T-EMU can only be installed on Linux machines and in order to use it a license key is needed.

3) Spacecraft Multicore Emulator

The Spacecraft Multicore Emulator [14] is based on Leon 3 Sparc V8 RISC architecture for spacecraft operational simulation. It uses direct interpretation to emulate the functionality of the On-board Software of a spacecraft. It is still a work-in-progress.

E. QEMU

QEMU [15] (Quick Emulator) is a processor emulator with two operating modes: namely full system emulation and user mode emulation. It can be installed on Windows, Linux and Mac OS X and can emulate several architectures (e.g. x86/x86 64, PowerPC, Sparc, ARM, etc). QEMU is released open-source under the GNU General Public License, version 2. Under the full system emulation, it is possible to emulate a full guest machine with a set of devices such as configurable CPUs, video card, memory, hard disk, etc. Here, every device is emulated and complete isolation from the host is achieved. In user mode emulation, QEMU can execute processes compiled for a specific CPU on a different CPU, translating system calls during runtime. The limitation of this mode is that not all system calls are supported and some system calls may not even be implemented in the target machine. QEMU uses a dynamic binary translation to native code with support for self-modifying code. The Tiny Code Generator (TCG) is responsible for the transformation of source instructions to an intermediate representation which is then transformed to target instructions.

Currently, QEMU supports SMP architectures with up to 255 CPUs depending on the guest/host architecture. In full system emulation mode, only a single thread is used by the host (usually in a round-robin fashion) even though the guest CPUs are emulated in parallel. With the increase of interest in multicore platforms, several efforts are currently being made to take advantage of these platforms and to map what is executing in multiple guest cores into multiple host threads. Several authors have already developed proof of concept implementations for multicore processors. Few examples include: HQEMU [16] implements a dynamic binary translation (DBT) mechanism; PQEMU [17] presents a similar approach to HQEMU where not only a parallelized DBT engine is tried, but advantage is also taken from the parallel code execution; Last but not least, COREMU [18] emulates multicore processors by creating separate instances of DBT per core with a library layer responsible for handling inter-core communication, device communication and synchronization. The multiprocessor support provided by QEMU is still in a work-in-progress state as several issues, such as atomicity and memory consistency still need to be addressed [19].

IV. KHRONOSIM QEMU MODULE

As discussed in the introduction, the goal of this work is to enable the integration of a hardware emulator in the closed-loop testing environment of the KhronoSim platform (the integration of the different components is out of scope of this paper.). For this purpose, a specific module (KqM – KhronoSim QEMU Module) was implemented which allows to create and control QEMU instances from the KhronoSim platform.

QEMU contains a QEMU Machine Protocol (QMP) that allows one to interact with QEMU instances using predefined commands. The KqM monitor natively implements this system and uses TCP sockets to communicate with each instance, by setting a TCP server on each instance and then sending QMP commands to interact, using JSON format. This approach allows for a simpler and suitable way to interact with QEMU. Moreover, it allows one to add new commands to the QEMU system, such as the throttle command which is discussed later in this section.

The KqM module (depicted in Figure 2) allows one to dynamically create and interact with several QEMU instances. The **QemuInstanceManager** is the entry point – each client must create an instance of this class (a singleton) in order to use the functionality of KqM. A private map structure holds all the active QEMU instances started by the manager. The following features are already implemented:

- startQI – This method creates a new QemuInstance object according to the given options passed as a string. On success, the method adds the new instance to the map and returns a Qemu instance identifier (ID), an internal identifier that univocally identifies the new instance. Upon invocation this method calls prepareQI() and executeQI().
- prepareQI – Prepares the new QEMU instance for execution and adds it to the map of available instances. From this state, a QEMU instance can be executed to start the guest machine. As input it accepts a string with the guest boot options.
- executeQI - Starts a previously prepared QEMU instance with the given ID.
- continueQI - Continues execution of a suspended QEMU instance. Takes the instance identifier as parameter.
- suspendQI - Suspends the execution of a QEMU instance. The machine can then be considered paused. Takes the instance identifier as parameter.
- shutdownQI – Shuts down a QEMU instance. Takes the instance identifier as parameter.
- listQIs – Lists all the QEMU instances in the system.
- executeCMD/executeCMDs – Executes one or a set of commands in a instance. Receives as arguments the instance ID and the command(s) to execute.
- shutdown() – Shuts down the QEMU instances manager.
- loadConfigurationFile – Loads the configurations of a given configuration file provided in the argument.
- trapShutdownCMD – Private method to check if a given command breaks the behavior of KqM. E.g. when a shutdown command is passed as input in executeCMD and it is not trapped, then a given QEMU instance would be terminated without the module knowing it.

The ***QemuInstance*** class serves as a bridge between the QemuInstanceManager and the QEMU instance itself. It encloses the private communication client (***QMPClient***) that interacts with the QEMU instance via sockets using the QMP protocol. Figure 3 shows a simplified code for this communication.

```
string QMPClient::send(string cmd){

    if(this->isConnected == false) {
        connect_to_server();
    }
    if(this->isConnected == true){
        Json_Parser parser;
        string parsed_cmd = parser.parse(cmd);
        write(this->socket_fd, parsed_cmd.c_str(),
              parsed_cmd.size());
        read_reply();
        return parsed_cmd;
    }
    return NULL_REPLY;
}

string QMPClient::read_reply(){
    string reply;
    reply.resize(BUFFER_SIZE);
    read(this->socket_fd, &reply[0], reply.size()-1);
    return reply;
}

void QMPClient::connect_to_server(){
    int socket_fd = socket(AF_INET,SOCK_STREAM,0);
    if(socket_fd <0)
        { /* Error treatment */ }
    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    const char * addr_const_char =
                     this->ip_address.c_str();

    in_addr_t in_addr = inet_addr(addr_const_char);
    if (INADDR_NONE == in_addr)
        { /* Error treatment */ }
    addr.sin_addr.s_addr = in_addr;
    addr.sin_port = htons(this->port);

    if(connect(socket_fd,(struct sockaddr *) &addr,
               sizeof(addr))<0)
        { /* Error treatment */ }
    else
    {
        this->socket_fd = socket_fd;
        this->isConnected = true;
    }
}
```

Figure 3 – Interfacing with QEMU

A. *Throttle control*

An important feature to support in order to be able to integrate the emulator in the KhronoSim platform is the ability to control and align the time base of the emulator with the overall timebase of the other modules (e.g., the control simulation provided by external components). Although this can be done by suspending/resuming the instance, a better interface is provided by accessing the QEMU throttle control (mostly used during VM migrations during heavy I/O operations).

The throttling of the CPU functionality achieves a reduction in instruction execution speed via non-blocking micro sleeps which result in executing less virtual machine related instructions per host clock cycle. This functionality is accessed by setting the desired throttle percentage, using the *cpu_throttle_set* function of QEMU (Figure 4).

```
void cpu_throttle_set(int new_throttle_pct)
{
    /* Ensure throttle percentage is
       within valid range */
    new_throttle_pct =
        MIN(new_throttle_pct, CPU_THROTTLE_PCT_MAX);
    new_throttle_pct =
        MAX(new_throttle_pct, CPU_THROTTLE_PCT_MIN);
    atomic_set(&throttle_percentage,
               new_throttle_pct);
    timer_mod(throttle_timer,
           qemu_clock_get_ns(QEMU_CLOCK_VIRTUAL_RT)
+
           CPU_THROTTLE_TIMESLICE_NS);
}
```

Figure 4 – Throttle control

For this purpose, QEMU is changed to expose this function to external commands. This is done by adding a new *throttle_cpu* command and the respective *hmp_throttle_cpu* function (Figure 5) in QEMU.

```
 STEXI
@item throttle_cpu
@findex throttle_cpu
ETEXI
{
   .name        = "t_cpu|throttle_cpu",
   .args_type   = "index:i",
   .params      = "index",
   .help        = "throttles the cpu to given pct",
   .cmd         = hmp_throttle_cpu,
},
STEXI

void hmp_throttle_cpu(Monitor *mon,
                      const QDict *qdict)
{
   //get percentage value
   int pct = qdict_get_int(qdict, "index");
   cpu_throttle_set(pct);
   monitor_printf(mon,
       "CPU executing at %d%% speed.\n",100-pct);
}
```

Figure 5 – Exposing the throttle control

V. CONCLUSIONS

This paper discusses the challenge of testing and validation of complex CPS, and the use of hardware emulation in the testing loop. To address this challenge the paper presents an overview of the KhronoSim QEMU Module, which was developed to enable the integration of hardware emulation in a closed loop in the KhronoSim platform, used to test and validate critical real-time CPS. This module enables KhronoSim to use the QEMU emulator to mimic the hardware platform, enabling the start, suspend or resume of QEMU instances as well as controlling the actual speed of emulation (interfacing with QEMU's throttle control).

ACKNOWLEDGMENT

This work was partially supported by FCT (Portuguese Foundation for Science and Technology) and co-financed by ERDF (European Regional Development Fund) under the Portugal2020 Program, within the CISTER Research Unit (CEC/04234); also by ANI (Portuguese Innovation Agency) under the ERDF (European Regional Development Fund) through "Portugal 2020" program, within project KHRONOSIM, nº 017611, POCI-01-0247-FEDER-017611.

REFERENCES

[1] Critical Software, KhronoSim – System for Simulation and Test of Complex Systems, Portuguese National Project POCI-01-0247-FEDER-017611.

[2] CAST, "CAST-32A, multi-core processors," Position paper, 2016, FAA.

[3] J. Xavier, G. Marc, B. Guy and F. Marc, "MULCORS project - The use of multicore processors in airborne systems," EASA 2011/6, 2012.

[4] S. Jim and N. Ravi, "Virtual Machines: Versatile Platforms for Systems and Processes," The Morgan Kaufmann Series in Computer Architecture and Design, 2005.

[5] G. Phillips, "Simplicity betrayed," Commun. ACM, vol. 53, no. 4, pp. 52-58, 2010.

[6] Antmicro, "Emul8," [Online]. Available: http://www.emul8.se/, last accessed March 2018.

[7] B. Nathan, B. Bradford, B. Gabriel, R. S. K., S. Ali, B. Arkaprava, H. Joel, R. H. Derek, K. Tushar, S. Somayeh, S. Rathijit, S. Korey, S. Muhammad, V. Nilay, D. H. Mark and A. W. David, "The gem5 simulator," SIGARCH Comput. Archit. News, 39(2), pp. 1-7, 2011.

[8] The Bochs Project, "bochs: The open source IA-32 emulation project," [Online]. Available: http://bochs.sourceforge.net/, last accessed March 2018.

[9] M. T. Jones, "Platform emulation with bochs," [Online]. Available: http://www.ibm.com/developerworks/library/l-bochs/, last accessed March 2018.

[10] ESA - European Space Agency, "Onboard computer and data handling - microprocessors," [Online]. Available: http://www.esa.int/Our_Activities/Space_Engineering_Technology/Onboard_Computer_and_Data_Handling/Microprocessors, last accessed March 2018.

[11] C. Gaisler, "LEON4," [Online]. Available: http://www.gaisler.com/index.php/products/processors/leon4, last accessed March 2018.

[12] C. Gaisler, "GRSIM," [Online]. Available: http://www.gaisler.com/index.php/products/simulators/grsim, last accessed March 2018.

[13] TERMA, "T-EMU," [Online]. Available: http://t-emu.terma.com/index.html, last accessed March 2018.

[14] Spacecraft Multicore Emulator, [Online], Available: https://github.com/kmonahopoulos/Spacecraft-Multicore-Emulator, last accessed March 2018.

[15] QEMU [Online]. Available: https://www.qemu.org/, last accessed March 2018.

[16] H. Ding-Yong, H. Chun-Chen, Y. Pen-Chung, W. Jan-Jan, H. Wei-Chung, L. Pangfeng, W. Chien-Min and C. Yeh-Ching, "Hqemu: A multi-threaded and retargetable dynamic binary translator on multicores," in Tenth International Symposium on Code (CGO'12), New York, 2012.

[17] J. H. Ding, P. C. Chang, W. C. Hsu and Y. C. Chung, "Pqemu: A parallel system emulator based on QEMU", in IEEE 17th International Conference on Parallel and Distributed Systems, 2011.

[18] W. Zhaoguo, L. Ran, C. Yufei, W. Xi, C. Haibo, Z. Weihua and Z. Binyu, "Coremu: A scalable and portable parallel full-system emulator," SIGPLAN Not., vol. 46, no. 8, pp. 213-222, 2011.

[19] QEMU, "QEMU - Features/tcg-multithread.," [Online]. Available: http://wiki.qemu-project.org/Features/tcg-multithread, last accessed March 2018.