A Virtual Prototype Semihosting Approach for Early Simulation of Cyber-Physical Systems

Bruno Prado, Daniel Dantas, Kalil Bispo, Thiago Fontes, Gabriel Santana and Rafael Silva
Department of Computing, Federal University of Sergipe
São Cristóvão, Brazil
{bruno, ddantas, kalil, thiago.fontes, gabrields, rafael.silva}@dcomp.ufs.br

***Abstract*—An early design space exploration of Cyber-Physical Systems (CPS) is a challenging task due to multi-domain areas and tight interaction of computing systems (cyber) and environment actuation, communication and sensing (physical). The CPS designer must be able to rapidly evaluate various solutions and to verify if project constraints were met. In this paper, we propose a CPS development framework based on Virtual Prototype (VP) to provide a high level of abstraction models to accurately simulate CPS behavior, using target independent semihosting interfaces and assessing performance and timing constraints. The experiments show a high timing accuracy in CoreMark and Dhrystone compute-intensive benchmarks and a low overhead in I/O intensive tasks for semihosting access of host resources, such as files and memory. An ECG heart rate detection case study was implemented using heterogeneous interfaces to demonstrate how hardware, human and software components can be seamlessly integrated to exchange real world data.**

***Index Terms*—Cyber-Physical Systems, Virtual Prototype, Semihosting, Modeling and Simulation, Embedded Systems**

I. Introduction

The Cyber-Physical Systems (CPS) are a complex and multi-disciplinary research area which aims to integrate computing systems (cyber) with environment actuation, communication and sensing (physical) [1]. These systems can be described from a high abstraction functional level [2], decoupled from design decisions and architecture, to detailed temporal models where timing constraints are essential for correct system behavior, specially in safety-critical CPS [3], [4]. The heterogeneity of hardware and software components and complex usage scenarios hinder CPS development, specially in situations where the hardware is not defined or even available yet [5]. To support early CPS behavior analysis and architectural exploration, Virtual Prototypes (VP) [6] provide a fast and accurate virtual environment to integrate hardware/-software components.

Despite simulation accuracy or performance, major issues arise from modeling realistic behavior of components from different disciplines and how they integrate and interact [7]. To overcome these challenges, the Hardware-in-the-Loop (HWiL) [8], Human-in-the-Loop (HUiL) [9] and Software-in-the-Loop (SWiL) [10] approaches can been used to host I/O operations, data acquisition and third-party software interaction, respectively. To accomplish this physical integration, low cost open-source platforms, such as Arduino [11], ESP devices [12] and Raspberry Pi [13], are widely available and support various types of actuators and sensors, including software libraries to handle I/O requests. However, these widespread components lack simulation support to assess performance and timing constraints, commonly performed ad-hoc [3].

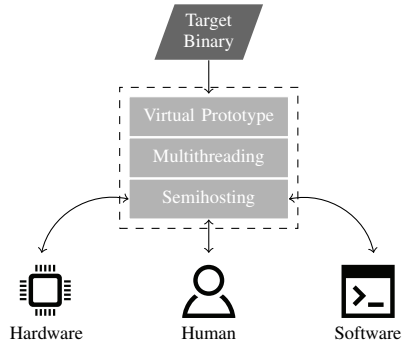


Fig. 1: ArchMERA overview

To enable early system level design, high abstraction simulation models are mandatory to cope with CPS complexity and to evaluate which solution or combination of solutions is more suitable to achieve system goals. In this work we propose a C++ multithreading simulation framework for early virtual prototyping: Architecture Modeling for Energy and Resource Assessment (ArchMERA) to fulfill this CPS requirement. Fig. 1 shows a block diagram of the proposed framework inside of dashed region. The framework receives as input the binary file $Target\ Binary$ which contains the compiled software to be executed by $Virtual\ Prototype$. In VP environment, functional models of buses, memories, modules, processors and registers can be described for a $Multithreading$ execution to take advantage of host parallelism. By the use of $Semihosting$ techniques, the interaction with third-party $Software$ components, such as Python SciPy [14] scientific library, can improve productivity and behavior validation. Besides that, the support for Newlib (C library for embedded systems) [15] provide access of embedded software to host I/O operations for $Hardware$ and $Human$ interaction, for example, file access and keyboard input, respectively. Due to its high level of abstraction, the physical interfaces can be seamlessly bound to host-attached (local) or to network-attached (remote) components, since, for both situations, the cyber software control handles the communication through library or system calls.

978-1-5386-6950-1/18/$31.00 ©2018 IEEE

This paper is organized as follows: Section II, closely related approaches for CPS development are reviewed; Section III, the early development flow for CPS is described focused on Design Space Exploration, Virtual Prototype and Interface Refinement processes; Section IV, the Semihosting Approach is detailed; Section V, an ECG case study is demonstrated; and Section VI, the paper conclusion, ongoing and future works.

II. Related Work

Schreiner et al. [16] presented a quasi-cycle accurate processor timing model for CPS software simulation, comparing the processor models against real hardware. His work focuses on real-time critical benchmarks whose behavior strongly depends on the timing simulation accuracy. The simulation results shows a mean error of less than 1% and performance of up to 9.6 Million of Instructions Per Second (MIPS). Their work evaluates the accuracy of results by measuring the runtime cycle counting, despite lacking of further integration with physical devices. Feng et al. [9] proposed Hardware-in-the-Loop (HWiL) and Human-in-the-Loop (HUiL) approaches for rapid embedded software development, offering both hardware and physical location transparent access. The development flow consists on the validation of a MATLAB application for domain-specific synthesis to C/C++ embedded deployment. Despite productivity gain, this approach depends on real hardware for deployment and C/C++ platform-dependent backend and proxies to assess system behavior and resource usage. Werner et al. [10] explored the concept of Software-in-the-Loop (SWiL) to capture real world data from simulation host and validate the application. By accessing host resources, the developer is able to evaluate early different methods of interaction between the target embedded software (cyber) and the host-connected devices (physical). In use cases, a simulation tune adjustment can speed up performance from real-time to up to 26.62 times when compared to real hardware. The file-based semihosting methods prevent a higher level of abstraction in target software calls, leading to throughput limitations in data read and write operations.

Zhang et al. [17] proposed a formalization of hardware/software interface to reduce the gap between tightly integrated software and physical plant components. A satellite altitude control system was co-simulated in a virtual execution environment with a C language embedded software running on QEMU [18] integrated to MATLAB/Simulink components. There was no communication between embedded control and real hardware, therefore, system characteristics too complex to be modeled, such as static friction and mechanical issues, were not properly verified. Theodoropoulos et at. [19] provide a integrated hardware/software CPS development framework focused in optimization of performance and energy consumption application supporting OpenCL, CUDA and C/C++ to generate code for GPU and FPGA devices. Regardless of relying on distributed x86 only system simulator COTSon [20], the design space exploration flow was omitted and no information about simulation versus real hardware prototyping was provided.

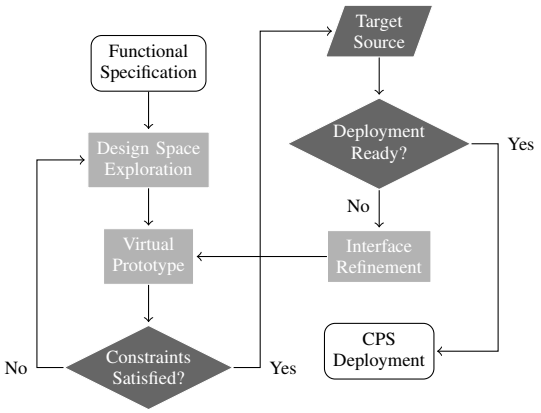


Fig. 2: Early CPS development flow

III. Early CPS Development Flow

Due to its tight interaction with heterogeneous devices from various application domains, it is almost mandatory the use of Virtual Prototype (VP) to enable early design space exploration. Through this simulation environment, the CPS designer is able to precisely verify which processor architecture or communication peripherals are more suitable to match the required integration and interaction constraints.

Fig. 2 shows an iterative and incremental early CPS development flow based on $Virtual\ Prototype$. Starting from $Functional\ Specification$ to $CPS\ Deployment$, this development flow supports multiple levels for physical components abstraction and enables the fast evaluation of different platforms to satisfy CPS constraints.

A. Design Space Exploration

The most challenging task in CPS development is the integration of multiple domain components and the achievement of a trade-off between conflicting system metrics, such as performance or power consumption. During the CPS conception, undetected design errors are from 5 to 10 times more expensive and problematic to solve in later development stages [21], thus, system designer must be able to make the best decisions.

Despite a commonly large design space, the system designer can easily create multiple platforms using various combinations of processor architectures and component interconnections in $Virtual\ Prototype$. Since it is focused on software integration and interaction, in this phase, the physical models can be highly abstract or describe an hypothetical device. When all system decisions are evaluated and project constraints are satisfied, the $Target\ Source$ for a specific platform can be refined until deployment in real hardware.

B. Virtual Prototype

Several system simulators, such as QEMU [18] and OVP [22], for example, can execute unmodified target binary code on host development machine. For CPS simulation, it is necessary to reach a trade-off between simulation performance and timing accuracy to provide useful results.

Listing 1: Virtual Prototype example

```
1  #include <am.hpp>
2  #include <mips_i.hpp>
3  using namespace archmera;
4  using namespace archmera::mips;
5
6  int main(int argc, char** argv) {
7      mips_i processor;
8      am_bus bus;
9      am_memory memory;
10     processor.MEM(bus);
11     processor.program_arguments(argc, argv);
12     processor.run();
13     bus.bind(processor);
14     bus.bind(memory);
15     memory.allocate(8, 1024 * 1024);
16     memory.elf32("model/software/mips/dhrystone/
           dhrystone.elf")
17     am_start();
18     return 0;
19 }
```

In the proposed framework, the system designer can instantiate, extend and model lightweight buses, memories, processors and registers components using C++ classes from a header library. In Listing 1, a minimal MIPS architecture [23] platform can be created from built-in components in very few lines: the dependencies are included (lines 1 to 4); the platform components are instantiated (lines 7 to 9); the processor memory port is bound to bus, the program arguments are passed and the processor will run until the software exits (lines 10 to 12); the bus is bound to processor and memory (lines 13 and 14); the 8 MB memory is allocated to load the Dhrystone benchmark [24] ELF binary (lines 15 and 16); and the simulation is started (line 17).

To accomplish physical interaction, the developer can input and output data using the embedded C standard library (Newlib) [15] or import his own libraries which would require system call routines for platform integration.

C. Interface Refinement

As already stated, an earlier detection of CPS design flaws can prevent later costly and time consuming fixes. However, in order to enable this early analysis, a high level of abstraction modeling should be applied to reduce the CPS simulation efforts.

Listing 2: High level C library write

```
1  int write(int file, char* ptr, int len) {
2      AM_PARAMETER(0x00, file);
3      AM_PARAMETER(0x04, ptr);
4      AM_PARAMETER(0x08, len);
5      AM_SYSCALL(0x44);
6      return AM_GET_VALUE(0x0C);
7  }
```

In Listing 2, a high level implementation of $write$ syscall uses C macros, that will be detailed in Section IV, and has the following parts: passing function parameters (lines 2 to 4); requesting VP handling (line 5); and retrieving return value (line 6). If software, for example, outputs a string message in terminal, writes sensor captured data to a file or accesses a descriptor-based device, this $write$ syscall will be invoked. By redirecting these requests to simulation host through VP syscall handling, the cyber software can readily interact with available local and remote physical components.

RETURN VALUE = **function**(PARAMETER 1, ..., PARAMETER n)

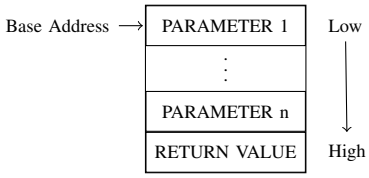


Fig. 3: Semihosting binary interface

Listing 3: Bare metal C library write

```
1  int write(int file, char* ptr, int len) {
2      int i;
3      if(file != STDOUT_FILENO && file != STDERR_FILENO) {
4          len = -1;
5      }
6      for(i = 0; i < len; i++) {
7          uart_send(*ptr++);
8      }
9      return len;
10 }
```

In bare metal systems, that is, systems without operating systems, besides its own functions, the embedded software controls directly the hardware. In Listing 3, the $write$ syscall handles file descriptors to support only standard output and error (lines 3 to 5), sends each byte via serial interface (lines 6 to 8) and returns the number of bytes written (line 9). Although showing an accurate timing behavior, at this abstraction level, the CPS components can be too complex to model.

For this reason, it is highly desirable to explore first the design space from abstract models, integrated to hardware devices (HWiL), human-based interfaces (HUiL) and third-party software tools (SWiL). Once CPS constraints are satisfied, the physical programming interfaces in $Target\ Source$ can be iteratively refined until the achievement of $CPS\ Deployment$.

IV. Semihosting Approach

The capability of embedded software running on a Virtual Prototype (VP) to access host machine facilities, such as C library functions or remote devices, is defined as semihosting. When target software requests a semihosting operation, the VP halts execution, reads input parameters from target to host, performs requested operation in host environment, returns generated values from host to target memory and resumes execution.

A. Binary Interface

Since the VP executes unmodified target binary software, an interface is required to request and to handle semihosting operations. This binary interface describes how parameters are passed, syscall flow is performed and return value from call is retrieved.

Fig. 3 illustrates the semihosting binary interface, that is, how the VP transfers data between host environment and cyber software control. In compiler linker script and VP semihosting map, a reserved memory address space, starting from a $Base\ Address$, must be set to store an arbitrary number of function parameters and a return value.

B. Interaction of Target Software and VP

Each semihosting operation should be assigned to a unique address which is used as key by an associative container. Due to its full customization capability, system design can freely define in which address an operation will be mapped and what abstraction level the data will be transferred between target software and VP.

Listing 4: Target software C macros

```
1  #define AM_ADDRESS(i)          \
2      (AM_BASE_ADDRESS + (i))
3  #define AM_GET_VALUE(i)        \
4      *((int*)(AM_ADDRESS(i)))
5  #define AM_PARAMETER(i, p)     \
6      AM_GET_VALUE(i) = (int)(p)
7  #define AM_SYSCALL(i)          \
8      int saved_return_address;\
9      __asm__("sw $ra, %0" : "=m"(saved_return_address));\
10     __asm__("jal %0" : : "r"(AM_ADDRESS(i)));\
11     __asm__("lw $ra, %0" : : "m"(saved_return_address))
```

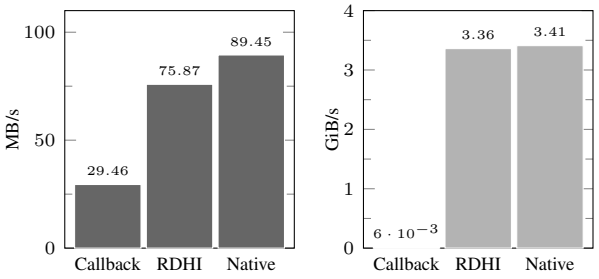
In a example of high level semihosting interface, shown in Listing 2, it is required in target software side the use of binary interface. In Listing 4, the used C macros are described as follows: $AM\_ADDRESS$ calculates the address offset from a defined base address (lines 1 and 2); AM_GET_VALUE reads a memory content from a specific memory address (lines 3 and 4); $AM\_PARAMETER$ writes the parameter in a memory location (lines 5 and 6); and $AM\_SYSCALL$, implemented for MIPS architecture [23], saves the return address of current flow (line 9), jumps to target address which be handled by VP (line 10) and returns to previous execution flow (line 11).

Listing 5: VP write syscall handling

```
1  void _write(am_interface_method* data, const am_field&
       base, const am_field& width) {
2      am_data file, ptr, len, return_value;
3      data->read(base + 0x00, file, width);
4      data->read(base + 0x04, ptr, width);
5      data->read(base + 0x08, len, width);
6      return_value = safe_write(file, data->get_pointer(
           ptr), len);
7      data->write(base + 0x0C, return_value, width);
8  }
```

At VP side, a similar procedure is required, however target specific information, such as processor word size and endianness, must be handled to avoid data formatting issues. In Listing 5, VP implements the target software high level C library write function described in Listing 2. While the memory interface is read to retrieve parameters (lines 2 to 5), the target specific word size in bytes (width) is passed and, if required, data endianness adjustments are done. To avoid race conditions from host multithreading execution, a $safe\_write$ wrapper of write syscall is used, receiving as parameters: file descriptor, direct host pointer of VP memory data and total number of bytes (line 6). The return value from syscall is written at specified address, considering word size and endianness formatting (line 7).

The architecture independent semihosting interface implementation in VP models enables a straightforward transition of processor models during design space exploration. This portability is possible due to target software source access to binary interface, which contains all required processor specific operations.



(a) Disk binary file copy (b) Memory data swapping

Fig. 4: Data exchange between host and VP

C. Accuracy and Performance

In order to evaluate simulation accuracy and performance, considering platform described in Listing 1, the CoreMark [25] and Dhrystone [24] benchmarks were executed in an Arch Linux 4.11 Intel Core 2 Quad CPU Q9505 at 2.83 GHz host with 4 GB of RAM and compiled with GCC 7.1.0 and Newlib 2.5.0.

The CoreMark achieved an average score of 1,310.10 and simulation performance of 44.63 MIPS, executing in $3.96 \pm 1.31\%$ minutes. In Dhrystone, a score of 67.13 DMIPS and simulation performance of 34.58 MIPS were achieved, while execution time was $5.12 \pm 0.59\%$ seconds.

In both benchmarks, the execution time deviations were quite close to Schreiner et al. [16], but the simulation performance was nearly up to 6 times faster, mainly due to low overhead between VP and native host calls.

Another important metric for semihosting performance is the data exchange throughput between host and VP. In Fig. 4 it is shown VP semihosting using Newlib calls (Callback), VP runtime direct host interface (RDHI) and host native interface (Native) data throughputs to file copy (4a) and memory data swapping (4b).

These I/O intensive tasks were developed to transfer 1 GB from disk files (block size of 1 MB) and 10 GiB from memory arrays (size of 1 MiB). In Fig. 4a, the proposed semihosting interface (RDHI) achieved an average throughput of 75.87 MB/s which is about 15% slower than native host performance. The memory data swapping using RDHI achieved an average transfer rate of 3.36 GiB/s, while native host was slightly faster (1.5%), as can be seen in Fig. 4b.

Werner et al. [10] showed a callback transfer rate of 5.87 MiB/s, very close to our callback results of 6.07 MiB/s, and its native mapping approach achieved data rate of 481.73 MiB/s, nearly 7 times slower than proposed RDHI throughput. The low overhead RDHI approach reduces the gap between VP and native host environments, enabling an early and fast CPS design evaluation.

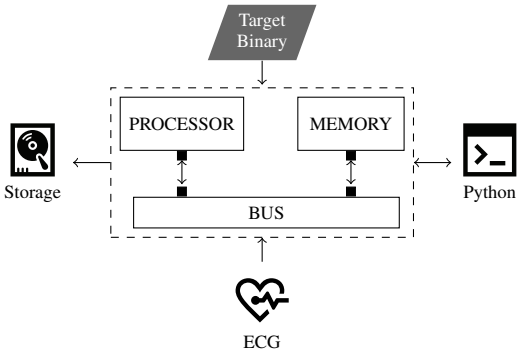


Fig. 5: Heart rate detection CPS

V. Case Study

In this case study, an instantaneous heart rate monitor from acquired ECG signal was implemented to demonstrate semihosting capabilities and early CPS assessments.

Fig. 5 depicts an overview of proposed case study. $Target\ Binary$ application is executed in VP platform to acquire ECG signal, process data using $Python$ libraries and write instantaneous heart rate in $Storage$. A minimal VP platform, already described in Listing 1, is used to execute embedded software, while the ECG signal is retrieved from a PhysioBank database [26] and processed by Biosppy toolbox [27]. Through proposed high level semihosting approach, the VP platform can seamlessly access these heterogeneous interfaces to incrementaly and iteratively explore design space, even when CPS components are not available yet.

A. ECG Acquisition

The PhysioBank database requires a specialized software WFDB [28] to convert signal information from binary to formatted data. After the ECG signal is processed, the VP control software can read file contents to its internal memory. This data is sent to Biosppy for instantaneous heart rate calculation from ECG signal.

Listing 6: ECG signal acquisition

```
1 int acquire_ecg(int* data, int size) {
2     int i;
3     for(i = 0; i < size && !feof(ECG); i++) {
4         fscanf(ECG, "%i\n", &data[i]);
5     }
6     return i;
7 }
```

The CPS software reads the ECG signal using standard formatted file input, as described in Listing 6. The function prototype $acquire\_ecg$ receives a pointer to integer array ($data$) and its number of requested samples ($size$) and returns the number of samples read (line 1). While the request number of samples and ECG file still has data, a sample is read and stored in $data$ at index i (lines 2 to 5). The index i is returned (line 6) and contains the total number of samples read from file, a value between 0 and $size$.

Fig. 6 shows an example of 7,000 samples read from ECG database. The signal was acquired with 16 bits resolution and sampling rate of 1 kHz.

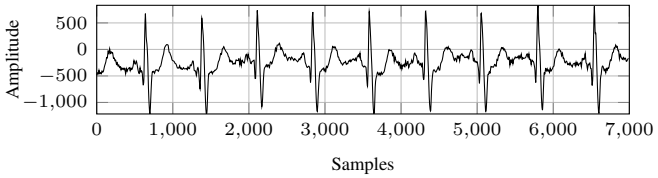


Fig. 6: ECG acquired signal sample

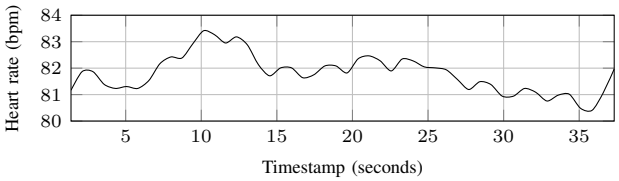


Fig. 7: Instantaneous heart rate

B. Heart Rate Detection

The ECG signal processing requires scientific software libraries and domain knowledge to deal properly with noise interference and breathing influence in R-peak amplitudes. Instead of writing from scratch a costly and time consuming solution, CPS designer can interface with available software toolboxes for early assessment of system requirements.

Listing 7: ECG heart rate detection

```
1 from biosppy.signals import ecg as bs
2
3 def heart_rate_ecg(data):
4     output = bs.ecg(signal = data, sampling_rate = 1000,
          show = False)
5     timestamp = output[5]
6     heart_rate = output[6]
7     return (timestamp, heart_rate)
```

In Listing 7, the heart rate is detected from acquired ECG signal in very few lines of code, described as follows: Biosppy signal processing for ECG is imported (line 1); function $heart\_rate\_ecg$ is defined, receiving ECG signal data (line 3); acquired ECG signal is processed at 1 kHz sample rate without plotting (line 4); and timestamp of detected heartbeats plus instantaneous heart rate tuple is returned (lines 5 to 7).

Once ECG signal is processed, the CPS software receives the timestamps of heartbeats and instantaneous heart rate through semihosting interface and writes them to file for data persistence. The instantaneous heart rate of 38,400 ECG signal samples can be seen in Fig. 7, showing an average of approximately 82 beats per minute.

C. Discussion

This case study is a proof of concept to demonstrate that high level of abstraction integration and interaction of heterogeneous CPS components is feasible. Also, a reduced CPS simulation environment footprint, that is, very few lines of code to implement it, and target independent semihosting approach enable a significant productivity gain. Feng et al. [9] estimated a development performance of 15 lines of code per man-hour, therefore, this paper proposes a CPS development

approach which is able to evaluate various candidate solutions in a matter of hours or days, instead of months, requiring less than 40 lines of source code for the ECG case study, as shown in Listings 1, 6 and 7.

As soon as CPS constraints are met, each one of its hardware, human or software interfaces can be iteratively refined or combined at different abstraction levels for evaluation purposes until CPS deployment. In this case study, for example, the ECG database can be replaced by a real-time ECG sensor attached to a human body, while Biosppsy toolbox interface is still used for heart rate detection. When ECG signal acquisition is validated, the Biosspy interface can be refined to a embedded software solution to detect heart rate, assessing software behavior, performance and resource usage.

VI. Conclusion

To cope with CPS design integration and interaction complexity, Hardware-in-the-Loop (HWiL), Human-in-the-Loop (HUiL) and Software-in-the-Loop (SWiL), through semihosting techniques, provide to cyber software a straightforward approach for physical device interfacing. The proposed design space exploration flow and a target independent semihosting approach enable rapid prototyping of various designs, showing small error in timing accuracy running benchmarks ($< 1.18\%$), low execution overhead from 1.5% to 15% in I/O experiments and demanding very low effort from development team.

Ongoing and future works include: more multi-domain case studies using HWiL, HUiL and SWiL components; improvement of VP simulation performance without compromising timing accuracy supporting various processor architectures, such as ARM and RISC-V; and comparison of high level CPS models to deployment solutions in terms of performance, source code productivity and timing behavior.

References

[1] V. Gunes, S. Peter, T. Givargis, and F. Vahid, "A survey on concepts, applications, and challenges in cyber-physical systems," *KSII Transactions on Internet and Information Systems (TIIS)*, vol. 12, no. 12, Dec 2014. [Online]. Available: http://dx.doi.org/10.3837/tiis.2014.12.001

[2] J. Wan, A. Canedo, and M. A. A. Faruque, "Functional model-based design methodology for automotive cyber-physical systems," *IEEE Systems Journal*, vol. 11, no. 4, pp. 2028–2039, Dec 2017.

[3] A. Shrivastava, M. Mehrabian, M. Khayatian, P. Derler, H. Andrade, K. Stanton, Y.-S. Li-Baboud, E. Griffor, M. Weiss, and J. Eidson, "A testbed to verify the timing behavior of cyber-physical systems: Invited," in *Proceedings of the 54th Annual Design Automation Conference 2017*, ser. DAC '17. New York, NY, USA: ACM, 2017, pp. 69:1–69:6. [Online]. Available: http://doi.acm.org/10.1145/3061639.3072955

[4] I. Graja, S. Kallel, N. Guermouche, and A. H. Kacem, "Time patterns for cyber-physical systems," in *2016 IEEE Symposium on Computers and Communication (ISCC)*, June 2016, pp. 1208–1211.

[5] P. J. Mosterman and J. Zander, "Cyber-physical systems challenges: a needs analysis for collaborating embedded software systems," *Software & Systems Modeling*, vol. 15, no. 1, pp. 5–16, Feb 2016. [Online]. Available: https://doi.org/10.1007/s10270-015-0469-x

[6] O. Bringmann, W. Ecker, A. Gerstlauer, A. Goyal, D. Mueller-Gritschneder, P. Sasidharan, and S. Singh, "The next generation of virtual prototyping: Ultra-fast yet accurate simulation of hw/sw systems," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE '15. San Jose, CA, USA: EDA Consortium, 2015, pp. 1698–1707. [Online]. Available: http://dl.acm.org/citation.cfm?id=2757012.2757206

[7] P. Hehenberger, B. Vogel-Heuser, D. Bradley, B. Eynard, T. Tomiyama, and S. Achiche, "Design, modelling, simulation and integration of cyber physical systems: Methods and applications," *Computers in Industry*, vol. 82, no. Supplement C, pp. 273 – 289, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0166361516300902

[8] F. Gao and F. Deng, "Design of a networked embedded software test platform based on software and hardware co-simulation," in *2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, Aug 2016, pp. 375–381.

[9] S. Feng, F. Quivira, and G. Schirner, "Framework for rapid development of embedded human-in-the-loop cyber-physical systems," in *2016 IEEE 16th International Conference on Bioinformatics and Bioengineering (BIBE)*, Oct 2016, pp. 208–215.

[10] S. Werner, L. Masing, F. Lesniak, and J. Becker, "Software-in-the-loop simulation of embedded control applications based on virtual platforms," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2015, pp. 1–8.

[11] Arduino, "Arduino - home," 2018. [Online]. Available: https://www.arduino.cc/

[12] Espressif, "Products — espressif systems," 2018. [Online]. Available: http://espressif.com/en/products

[13] T. R. P. Foundation, "Raspberry pi - teach, learn, and make with raspberry pi," 2018. [Online]. Available: https://www.raspberrypi.org/

[14] S. developers, "Scientific computing tools for python," 2018. [Online]. Available: https://www.scipy.org/

[15] C. Vinschen and J. Johnston, "The newlib homepage," 2018. [Online]. Available: https://sourceware.org/newlib/

[16] S. Schreiner, R. Grgen, K. Grttner, and W. Nebel, "A quasi-cycle accurate timing model for binary translation based instruction set simulators," in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, July 2016, pp. 348–353.

[17] Y. Zhang, Y. Dong, W. Feng, and M. Huang, "A co-simulation interface for cyber-physical systems," in *2016 13th International Conference on Embedded Software and Systems (ICESS)*, Aug 2016, pp. 176–181.

[18] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247360.1247401

[19] D. Theodoropoulos, S. Mazumdar, E. Ayguade, N. Bettin, J. Bueno, S. Ermini, A. Filgueras, D. Jimnez-Gonzlez, C. lvarez Martnez, X. Martorell, F. Montefoschi, D. Oro, D. Pnevmatikatos, A. Rizzo, P. Gai, S. Garzarella, B. Morelli, A. Pomella, and R. Giorgi, "The axiom platform for next-generation cyber physical systems," *Microprocessors and Microsystems*, vol. 52, no. Supplement C, pp. 540 – 555, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141933116304434

[20] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, "Cotson: infrastructure for full system simulation," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 1, pp. 52–61, 2009.

[21] J. Wan, A. Canedo, and M. A. A. Faruque, "Physical codesign at the functional level for multidomain automotive systems," *IEEE Systems Journal*, vol. 11, no. 4, pp. 2949–2959, Dec 2017.

[22] OVP, "Open virtual platforms," 2018. [Online]. Available: http://www.ovpworld.org/

[23] Imagination, "Mips processors - the leading alternative mainstream cpu architecture," 2018. [Online]. Available: https://www.imgtec.com/mips/

[24] R. P. Weicker, "Dhrystone: a synthetic systems programming benchmark," *Commun. ACM*, vol. 27, no. 10, pp. 1013–1030, Oct. 1984. [Online]. Available: http://doi.acm.org/10.1145/358274.358283

[25] CoreMark, "Eembc - coremark - processor benchmark," 2018. [Online]. Available: https://www.eembc.org/coremark/index.php

[26] A. L. Goldberger, L. A. N. Amaral, L. Glass, J. M. Hausdorff, P. C. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, and H. E. Stanley, "Physiobank, physiotoolkit, and physionet," *Circulation*, vol. 101, no. 23, pp. e215–e220, 2000. [Online]. Available: http://circ.ahajournals.org/content/101/23/e215

[27] C. Carreiras, A. P. Alves, A. Lourenço, F. Canento, H. Silva, A. Fred *et al.*, "BioSPPy: Biosignal processing in Python," 2015–. [Online]. Available: https://github.com/PIA-Group/BioSPPy/

[28] PhysioNet, "The wfdb software package," 2018. [Online]. Available: https://www.physionet.org/physiotools/wfdb.shtml