Real-time Hardware/Software Co-Design Using DEVS-based Transparent M&S Framework

José L. Risco-Martín
Complutense University of Madrid
C/Prof. José García Santesmases, Madrid, Spain
jlrisco@ucm.es

Saurabh Mittal
The MITRE Corporation
McLean, VA, USA
smittal@mitre.org

Juan Carlos Fabero
Complutense University of Madrid
C/Prof. José García Santesmases, Madrid, Spain
jcfabero@ucm.es

Pedro Malagón
Technical University of Madrid
Avda. Complutense, Madrid, Spain
malagon@die.upm.es

José L. Ayala
Complutense University of Madrid
C/Prof. José García Santesmases, Madrid, Spain
jayala@ucm.es

ABSTRACT

Design and development of hard Real-Time (RT) embedded systems present several crucial requirements regarding criticality and timeliness of these systems. Formal methods have been presented as a promising alternative to deal with the design issues of these applications. However, these formal method do not scale well in complex systems. Modeling and Simulation (M&S) provides cost-effective approaches to verify and validate the design and implementation details of complex RT applications. Nevertheless, M&S approaches and artifacts are usually discarded in the later phases of the development. Discrete Event Systems Specification (DEVS) provides an appropriate M&S framework to provide formal specifications to the actual RT system, incrementally moving from software specifications to a full hardware embedded system. In this work, we propose a hardware-in-the-loop model-driven method, based on DEVS for RT/embedded application/systems engineering. Our approach is based on an incremental substitution of DEVS virtual software models with Unix-compliant device files through a formally defined process in the modeling phase. Consequently, any DEVS simulation engine can be used. This paper advances the state-of-the-art in hardware-software co-design methodologies.

Author Keywords

Discrete Event Simulation; DEVS; Model Based Approach; Hardware In the Loop;

ACM Classification Keywords

I.6.1 SIMULATION AND MODELING: Simulation Theory; J.6 COMPUTER-AIDED ENGINEERING: *Computer-aided design (CAD)*; D.4.7 OPERATING SYSTEMS: Organization and Design—*Real-time systems and embedded systems*

INTRODUCTION AND RELATED WORK

The design, development and implementation of real-time embedded systems continues to be a a challenging effort at systems engineers level. The set of constraints related to real time management, task execution deadlines, power consumption, etc., has always represented a serious drawback when designing these kind of devices. The problem is much more accentuated today, as the new era of the Internet of Things (IoT) dawns. This becomes a complex adaptive system when the human element is brought as an component of the system itself. The Modeling and Simulation (M&S) of Cyber Complex Adaptive Systems (CyCAS) are continuously demanding new formal methods to manage the design, development and implementation of such ultra-large systems with a high level of quality, accuracy and fulfilling all the real-time constraints that one can imagine [8].

On one hand, typical approaches already exist that tackle the design of real time embedded systems using set of formal methods, e.g., bond graphs, cellular automata, partial differential equations, queing models, etc. [4]. Most formal methods are either hard to scale up to larger systems, or require a difficult testing effort without guarantees for bug-free final products [12]. On the other hand, systems engineers have often relied on the use of M&S techniques to make system development tasks manageable. Construction of a virtual model along with the corresponding analysis through simulation, reduces both costs and risks, along with enhanced quality and system capabilities. Conclusively, M&S allows users to experiment with a virtual system, explore the design, perform verification and validation mechanisms, and much more. However, many of the M&S techniques do not allow an incremental design, by means of a gradual inclusion of Hardware In the Loop (HIL) components. Instead,

SummerSim-SCSC 2016 July 24-27 Montreal, Quebec, Canada

the virtual model is usually discarded in the later stages of the development, i.e. during the hardware system synthesis, instead of doing an incremental substitution of sub-models with hardware components. Additionally, heterogeneous systems, combining software and hardware models in a whole ecosystem, are the typical scenarios of IoT systems today, e.g., smart cars, smart buildings, smart cities or smart homes in integrated energy environments [11].

In this paper, we propose a method to help with the design of RT/embedded complex systems and advance the state-of-the-art in hardware-software co-design methodologies towards IoT and CyCAS. We define a formal M&S incremental design approach based on the Discrete Event Systems Specification (DEVS) M&S formalism and build on existing work in this area. During the last decade, Moallemi and Wainer, for example, have presented several DEVS-based approaches [17], [12], [13]. In almost all of the approaches, a simulation engine was substituted for RT hardware simulation. In our approach, we do not modify the simulation engine to allow RT DEVS communication with hardware devices. Instead, our methodology suggests a formal straightforward substitution of abstract hardware models (developed as DEVS software models) with equivalent interface models to communicate with actual hardware. The concept of transparent simulation environment was developed for software components/services in our earlier work [9, 10]. We apply the same concept towards transparent hardware simulation environment. We further define a formal procedure to build these interface models, named "star" models.

Of course, there are other interesting techniques to deal with the incremental design of RT embedded systems. As M&S approaches, we may find SysML (Systems Modeling Language) [6], UML-RT (the Unified Modeling Language for Real-Time)) [7], the BIP (Behavior, Interaction, and Priority) methodology [3], and tools like Ptolemy II [5], SystemC [2], MatLab/Simulink [1], etc. Some deal with the modeling domain and some have integrated simulation backend. However, none offers the capability we require. An analysis and comparison of these other techniques and tools is out of the scope of this paper. This paper's objective it to present the new HIL model-driven DEVS-based method to develop RT/embedded applications.

The remainder of this paper is organized as follows. Firstly, a brief background is given. Secondly, the full approach is described. Next, we show a case study where the methodology proposed has been applied. Finally, some conclusions of this work are drawn.

BACKGROUND

The Discrete Event System Specification

DEVS is a general formalism for discrete event system modeling based on mathematical Set Theory [19]. Once a system is described in terms of the DEVS formalism, it can be easily implemented using an existing software/hardware library.

DEVS formally represents a system by three sets and five functions: input set (X), output set (Y), state set (S), time advance function (ta), external transition function (δ_{ext}), internal transition function (δ_{int}), confluent function (δ_{con}), and output function (λ). The DEVS formalism provides the framework for information modeling which gives several advantages to analyze and design complex systems: completeness, verifiability, extensibility, and maintainability.

DEVS models are of two types: atomic and coupled. The atomic model is the irreducible model definition that specifies the behavior for any modeled entity: processes an input event based on its state and condition, and generates an output event and changes its state. The coupled model is the aggregation/composition of two or more atomic and coupled models connected by explicit couplings. The formal definition of parallel DEVS (P-DEVS) is given in [19]. An atomic model is defined by the following equation:

$$A = \langle I, O, X, S, Y, \lambda, \delta_{\text{int}}, \delta_{\text{ext}}, \delta_{\text{con}}, ta \rangle \tag{1}$$

where:

- I is the set of input ports.
- O is the set of output ports.
- X is the set of inputs described in terms of pairs port-value: $\{p, v\}$.
- S is the state space. It includes not only the current state of the atomic model, but also two special parameters called σ and *phase*, which are the time until the next event generation, and a description of the current state (usually in natural language), respectively.
- Y is the set of outputs, also described in terms of pairs port-value: $\{p, v\}$.
- $\lambda : S \rightarrow Y$ is the output function. When the time elapsed since the last output function is equal to σ, then λ is automatically executed.
- $\delta_{\text{int}} : S \rightarrow S$ is the internal transition function. It is executed right after the output (λ) function and is used to change the state S (including *phase* and σ)
- $\delta_{\text{ext}} : Q \cdot X^b \rightarrow S$ is the external transition function. It is automatically executed when an external event arrives to one of the input ports, changing the current state if needed.
  - $Q = (s, e) s \in S, 0 \leq e \leq ta(s)$ is the total state set, where e is the time elapsed since the last transition.
  - X^b is the set of bags over elements in X.
- $\delta_{\text{con}} : Q \cdot X^b \rightarrow S$ is the confluent function, subject to $\delta_{\text{con}}(s, \emptyset) = \delta_{\text{int}}(s)$. This transition is selected if δ_{ext} and δ_{int} must be executed at the same instant.
- $ta(s) : S \rightarrow \Re_0^+ \cup \infty$ is the time advance function.

The formal definition of a coupled model is described as:

$$M = \langle I, O, X, Y, C_i, EIC, EOC, IC \rangle \tag{2}$$

where:

- I, O are the set of external (not coupled) input and output ports.
- X is the set of external input events.
- Y is the set of output events.
- C_i is a set of DEVS component models (atomic or coupled). Note that C_i makes this definition recursive.
- EIC is the external input coupling relation.
- EOC is the external output coupling relation.
- IC is the internal coupling relation.

Given the recursive definition of M, a coupled model can itself be a part of a component in a larger coupled model system giving rise to a hierarchical DEVS model construction.

Device files and device drivers

In a GNU/Linux system, there are two main concepts: device file and device driver. A device file is an interface for a device driver.

The device driver, is a piece of software that operates or controls a particular type of device, such as printers or serial ports. However, every device driver can be specialized to interact with only a specific resource on those devices, such as disk partitions. In Unix-like systems, a device driver is typically compiled as a kernel module and can be loaded at runtime.

A device file appears inside a file system almost as an ordinary file. These files allow user's software to interact with a device driver using standard input/output system calls, which simplifies the interface and unifies user-space *Input/Output (I/O)* mechanisms. As one can easily imagine, such device files exist in all operating systems, e.g. MS Windows, MS-DOS, and OS/2. Additionally, device files are useful for accessing system resources that have no connection with any actual devices, like data sinks or random number generators. A device file can represent character devices, which emit a stream data one character at a time, or block devices which allow random access to blocks of data. Device files are usually found under the `/dev` directory and are created with the `mknod` system call. The kernel resource exposed by the device file is identified by a major and minor number. The device file exposes what the device driver shows through the I/O interface. For instance, the character device file representing a mouse, exposes the movement of the mouse as a character stream. Some device files also take inputs, allowing applications to communicate with the device by writing to its device file.

For more information, the reader can refer to [16].

PROPOSED APPROACH

Hardware/Software Interface

The most critical characteristics when designing simulations with HIL are (i) the management of deadlines, more particularly in real-time systems, and (ii) the interfaces used to communicate with hardware. Deadlines are completely handled by the DEVS formalism (DEVS-RT in our case [18], implemented in xDEVS [15]). In this paper, we do not define activity mapping time-constraints. Thus, the HIL interaction can be seen as a DEVS atomic model that receives the input from hardware, processes the input and perform state changes in real-time producing outputs within an acceptable predefined deadline. In other words, the HW/SW interface is considered as a DEVS atomic model that internally manages a device file, which indeed is handled as a regular file. Since the set of model-states are not tied to hardware activities, we do not need to check their duration. At the end, we have a regular RT-DEVS model, which any DEVS simulator engine can execute without much configuration effort. The only constraint is that an additional atomic model attached to a device file must be defined for each DEVS hardware component. This is analogous to creating a DEVS model-wrapper for hardware or software component [11]. Similar work by Mittal and Risco-Martín [10, 9] in the area of Service-oriented Architecture (SOA) enabled DEVS M&S framework uses the transparent simulation platform principle.

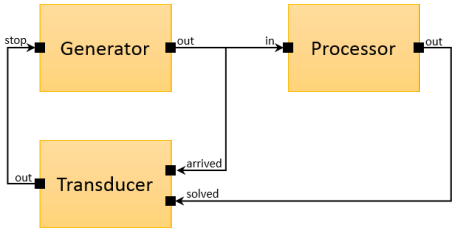

Figure 1. GPT DEVS structure, all implemented in software.

Consider a simple example. Figure 1 shows the DEVS structure of a typical Generator Processor Transducer (GPT) DEVS model [14], consisting of three atomic models. The generator atomic model generates job-messages at fixed time intervals and sends them via the "out" port. The transducer atomic model accepts job-messages from the generator at its "arrived" port and monitors their arrival time instances. It also accepts job-messages at the "solved" port. When a message arrives at the "solved" port, the transducer matches this job with the previous job that had earlier arrived on the "arrived" port (when it was generated) and calculates their time difference. The transducer monitors the response (in this case the turnaround time) of messages that are injected into an observed system. The observed system in this case is the processor atomic model. A processor accepts jobs at its "in" port and sends them via the "out" port again after some finite, but non-zero time period. If the processor is busy when a new job arrives, the processor discards it. Finally, the transducer stops the generation of jobs by sending any event from its "out" port to the "stop" port at the generator.

Now suppose that the processor component is an actual hardware device that must be included in the simulation. In this case, Figure 2 shows the architecture proposed in this paper. First, a device driver must be developed to implement the communication with the hardware device. Since the pair (device driver, hardware device) will be managed by a device file, a new Processor atomic model must be defined, to emu-

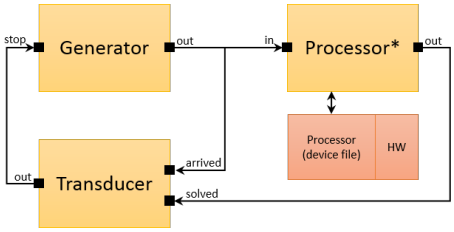

Figure 2. GPT DEVS structure with Processor working as a hardware device.

late the behavior of the Processor atomic model, but by means of Input/Output (IO) operations through the device file. This is the Processor* model defined in Figure 2.

DEVS implementation of the HW/SW interface

Continuing with the GPT example, let us delve in a bit detail for the Processor model. A job j is processed in a wall-clock time equal to j_p seconds. We first show the formal DEVS specification of the original Processor atomic model (Figure 1):

$$
\begin{aligned}
\text{Processor} &= \langle X, S, Y, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, \text{ta} \rangle \\
X &= \{(\text{in}, j \in \mathbf{J})\} \\
&\quad \mathbf{J} \text{ is a set of Jobs} \\
S &= (\sigma \times \text{phase} \times j \in \mathbf{J}) \\
&\quad \text{phase} \in \{\text{busy}, \text{passive}\} \\
Y &= \{(\text{out}, j \in \mathbf{J})\} \\
\text{ta}(\sigma, \text{phase}, j) &= \sigma \\
\lambda(\sigma, \text{phase}, j) &= j \\
\delta_{\text{int}}(\sigma, \text{phase}, j) &= (\infty, \text{passive}, \oslash) \\
\delta_{\text{ext}}\left(\sigma, \text{passive}, j, e, (\text{in}, j')\right) &= \left(j'_p, \text{busy}, j'\right) \\
\delta_{\text{ext}}\left(\sigma, \text{busy}, j, e, (\text{in}, j')\right) &= (\sigma - e, \text{busy}, j)
\end{aligned}
$$

Now, considering that we have a hardware processor and its corresponding device file f, the previous specification must be slightly modified to incorporate the hardware device into the simulation:

$$
\begin{aligned}
\text{Processor}* &= \langle X, S, Y, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, \text{ta} \rangle \\
X &= \{(\text{in}, j \in \mathbf{J})\} \\
&\quad \mathbf{J} \text{ is a set of Jobs} \\
S &= (\sigma \times \text{phase} \times j \in \mathbf{J}) \\
&\quad \text{phase} \in \{\text{busy}, \text{passive}\} \\
Y &= \{(\text{out}, j \in \mathbf{J})\} \\
\text{ta}(\sigma, \text{phase}, j) &= \sigma \\
\lambda(\sigma, \text{phase}, j) &= j \leftarrow f \\
\delta_{\text{int}}(\sigma, \text{phase}, j) &= (\infty, \text{passive}, \oslash) \\
\delta_{\text{ext}}\left(\sigma, \text{passive}, j, e, (\text{in}, j')\right) &= \left(j'_p, \text{busy}, f \leftarrow j'\right) \\
\delta_{\text{ext}}\left(\sigma, \text{busy}, j, e, (\text{in}, j')\right) &= (\sigma - e, \text{busy}, j)
\end{aligned}
$$

As stated in the previous specification, the function δ_{ext} writes the received job in the device file (denoted as $f \leftarrow j'$), whereas λ loads the already processed job from the same file (denoted as $j \leftarrow f$), according to the time constraints specified in j_p. As can be imagined, the greater weight of the needed modifications are handled by the device driver, which must "understand" the atomic model format and transform this into the format of the hardware interface. Figure 3 shows the template used to implement a device driver that will communicate data to the actual hardware. As can be seen, only two functions called `device_write` and `device_read` must be implemented. The first one is automatically executed when data is written in the device file, whereas the second one is automatically executed when the device file is to be read. In the Processor* example, the job must be written in the device file through the `device_write` function (actually the job would be stored in the `buffer` input variable, as text). On the other hand, the job is loaded in the output function, when the data is read from the device file and `device_read` is automatically executed. An exact form of these hardware sections is shown in the case study.

As can be seen, the greatest advantage of this approach is the idea that no modification is required for the DEVS M&S engine i.e. it is transparent. As a result, an incremental design of model-driven HIL is now possible, starting with pure software models and gradually replacing them with hardware components, until the full hardware implementation is reached. Additionally, hardware verification and validation is also possible which involves comparing the output of the software models against the equivalent hardware models. In the following, we show a case study of the proposed architecture.

CASE STUDY: DESIGN AND IMPLEMENTATION OF AN ELEVATOR CIRCUIT CONTROLLER

System specification

The goal of this case study is to design and build a circuit that will emulate the controller of an elevator in a building with 7 floors. Figure 4 shows a general scheme of the circuit that is to be designed.

The circuit will have the following set of ports:

- A 3-bit output `Q`, which shows the floor in which the elevator is stopped.
- A 3-bit input `X`, which represents the desired target floor.
- A clock input `CLK`.
- An input `INI`, of synchronous initialization to (000), active Low.

Both `X` and `Q` encode the floor in binary, i.e., (000) is the ground floor, (001) is the first floor, and so forth until (111), which is the last (seventh) floor.

The output will be (000) in all those cases where `INI` is 0. In those cycles where `INI` is 1, the behavior of the system will be as follows:

- If Q=X the output will keep its value, i.e., we are in the target level.

```
// ...
// Some global variables:
dev_t devnumber;
struct cdev* currdev = NULL;

int init_module(void) {
  // Major and minor setup
  // ...
  major = MAJOR(devnumber);
  minor = MINOR(devnumber);
}

void cleanup_module(void) {
  if (currdev)
    cdev_del(currdev);
  unregister_chrdev_region(devnumber, 1);
}

static int device_open(struct inode *inode, struct file *file) {
  // ...
  try_module_get(THIS_MODULE);
  return SUCCESS;
}

static int device_release(struct inode *inode, struct file *file) {
  // ...
  module_put(THIS_MODULE);
  return SUCCESS;
}

static ssize_t device_write(struct file* filp, const char* buffer, size_t length, loff_t* off) {
  // buff contains the data written in the external
  // transition function, which is passed to userbuf:
  char userbuf[MAX_COMMAND_LEN+1];
  if (copy_from_user(userbuf,buffer,length))
    return -EFAULT;
  userbuf[length]='\0';
  // Here the data must be sent to the hardware device:
  // HARDWARE SPECIFIC CODE
  // ...
  return length;
}

static ssize_t device_read(struct file* filp, char* buffer, size_t length, loff_t* offset) {
  // At the end, buffer must store the data read by the output function
  // i.e., the output of the hardware device
  char userbuf[MAX_COMMAND_LEN+1];
  // Here the data of the hardware device must be stored into userbuf:
  // HARDWARE SPECIFIC CODE
  // ...
  if (copy_to_user(buffer,userbuf,length))
    return -EFAULT;
  return length;
}
```

Figure 3. Device driver code snippet. The two labeled HARDWARE sections require specific communication with the hardware device.

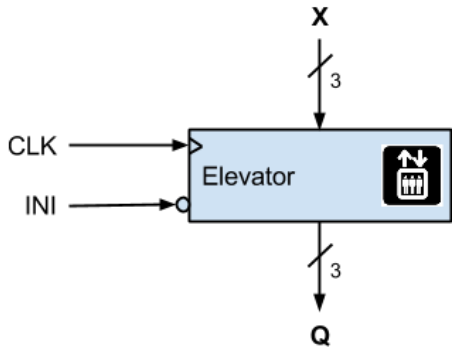

Figure 4. High level specification of the circuit.

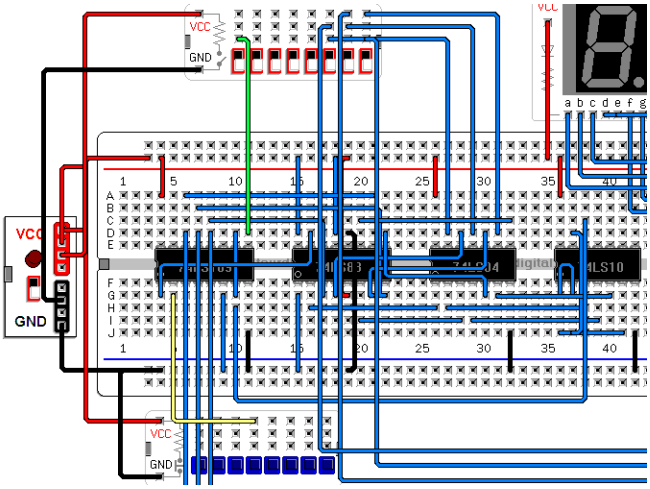

Figure 5. Schematic design of the circuit, drawn in a breadboard.

- If Q>X, the output must be (cycle after cycle) decreasing until Q=X. The elevator must go down because the floor in which it is placed is greater than the commanded floor.
- If Q<X, the output must be increasing until Q=X. In this case, the elevator must go up because the floor in which it is placed is less than the commanded floor.

To implement the controller, the following Integrated Circuits (ICs) are used:

- 74169: A bidirectional modulo 16 synchronous counter (with parallel load signal, active low, which takes precedence over the two signals to enable the count, also active low)
- 74283: A 4-bit full adder
- 7410: 3 3-input NAND gates
- 7404: 6 inverters

System design

To implement a prototype, we first implemented all the four ICs as xDEVS [15] atomic models in a Raspberry Pi, through the software ports per the circuit shown in Figure 5.

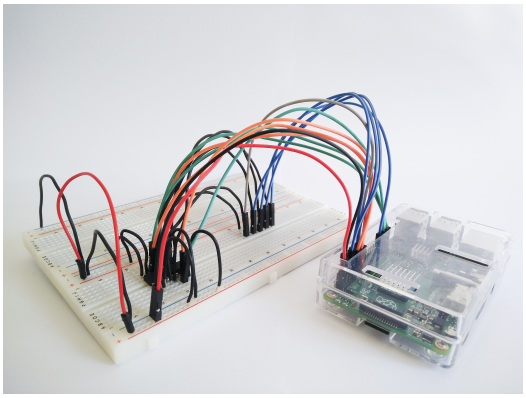

Figure 6. Elevator controller with the adder as a HIL component.

Secondly, we replaced all the ICs incrementally with their equivalent hardware ICs. We successfully connected a breadboard to the Raspberry, developing a device driver and its associated device file. The device drivers were accordingly upgraded as additional software ICs were replaced by actual hardware. Figure 6 shows the circuit implementation with the IC 74283 working in the breadboard, and ICs 74169, 7410 and 7404 implemented in the Raspberry Pi through xDEVS. Once the IC 74283 is moved to the breadboard, we applied our approach as described in the previous section, i.e., we developed the interface as a virtual IC 74283 communicating with the device file abstracting it's state and behavior.

Figure 7 shows a snippet of the IC 74283 driver, commented as HARDWARE SPECIFIC CODE in Figure 3. For the sake of clarity, we have included generic calls to the GPIO Raspberry interface. Basically, the `write` driver function sends A and B (previously read from the device file) through the GPIO pins connected to the IC 74283 inputs in Figure 6. Similarly, the `read` driver function reads the result from the GPIO pins connected to the IC outputs and virtually writes those values to the device file, making them available for the IC 74283* xDEVS hardware interface. As a result, the complete circuit works transparently, without any modification in the simulation engine. As the whole design is operational through DEVS formalism, we can leverage the available model-checking mechanisms for DEVS systems, e.g. Verification and Validation (V&V) tasks.

Finally, Figure 8 depicts all the hardware circuit components on the breadboard as a part of the model. In this case, the xDEVS software model is formed by the equivalent four "star" models plus the simulation clock. We ran several tests on the resulting hardware-software system with no noticeable issues at both the modeling and simulation levels. It is worth mentioning that "star" models are very conducive to perform V&V tests, without including additional DEVS atomic models to the system.

CONCLUSION

We have shown that M&S techniques offer a significant support for the design of complex real-time embedded systems. However, in an IoT new era where real-time systems become

```
// ...
// Some global variables:
// ...

static ssize_t device_write(struct file* filp, const char* buffer, size_t length, loff_t* off) {
  // ...
  // HARDWARE SPECIFIC CODE
  if(sscanf(&userbuf[0],''add \%i \%i\n'', &numA, &numB)) {
    /* Send A and B */
    decimalToArray(numA, A);
    decimalToArray(numB, B);
    sendValues(A, B); // Through GPIO: gpio_set_value(Apin[i], A[i]);
                      // Through GPIO: gpio_set_value(Bpin[i], B[i]);
  }
  return length;
}

static ssize_t device_read(struct file* filp, char* buffer, size_t length, loff_t* offset) {
  // ...
  // Here the data of the hardware device must be stored into userbuf:
  // HARDWARE SPECIFIC CODE
  readSum();            // sum <- gpio_get_value(Cpin[index])
  sprintf(userbuf, "\%i", sum);
  length = strlen(kbuff);
  // ...
  return length;
}
```

Figure 7. Device driver code snippet, implemented for the IC 74283. Only the HARDWARE parts described in the previous section are partially listed here.

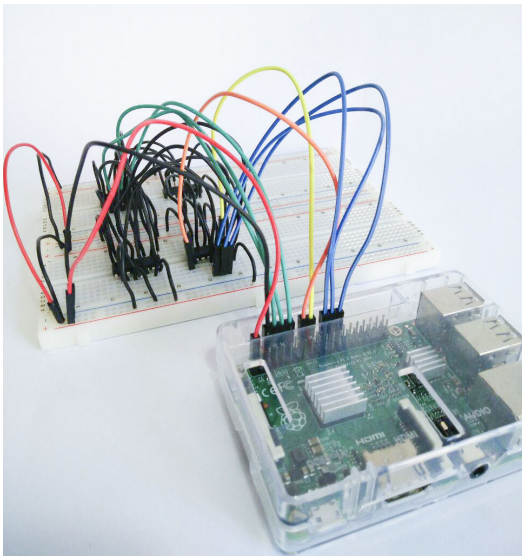

Figure 8. Elevator controller with all the four integrated circuits placed in the breadboard.

much more complex, new mechanisms to allow incremental designs and implementations are urgently needed. In this paper, we have used DEVS as the basis to develop heterogeneous hardware/software systems in an incremental way. The proposed mechanism is straightforward. The transition from pure virtual models to heterogeneous models is performed through well defined steps where each virtual model is substituted with an equivalent model that acts as an interface with the real system. In this regard, the complete system is still a DEVS system and as a consequence, all the DEVS M&S theory is applicable and brought to bear for systems validation and verification. Since the communication with the actual hardware is based on device drivers, the DEVS M&S simulation engine does not have to be tuned each time that a new hardware component must be included. This is the prime contribution as a transparent M&S framework where the model, after validation and verification, is replaced by the actual hardware in an incremental and transparent way.

To demonstrate the transparent hardware-software M&S environment, we presented a case study: the design and implementation of an elevator circuit controller. The whole system was first modeled as a software using xDEVS, an open source DEVS library (though any DEVS library can be used). We showed the methodology to incrementally transition from virtual models to the actual hardware until all the software models are replaced. We also stated that "star" models, that interface between the DEVS virtual model and actual hardware can be also used to perform systems V&V, to be attempted in our future work.

IoT is advancing by leaps and bounds. There is a dearth of system-based M&S methodologies that can guarantee the behavior of a particular hardware when it gets integrated in the

large whole. With this methodology, any hardware, when abstracted through DEVS-Based systems theoretical framework in a transparent manner, has the potential to be experimented upon with larger domain models that can evaluate its behavior in a larger system. Further, the second contribution of this work is the validation of the developed software model of the hardware itself. This capability is very much needed in IoT design and analysis: to evaluate new pieces of hardware technology that are ready to plug into the larger complex adaptive systems, such as Internet or mission-critical systems. With validated software models, large scale simulations with thousands or millions of virtual agents (of abstracted hardware as a DEVS agents) in High Performance Computing environments (HPC) then become possible to study the effects of any new technology that comes to be integrated in IoT.

ACKNOWLEDGMENTS

The authors would like to thank Mr. Miguel Higuera Romero, who implemented the case study presented in this paper as part of his senior project. We would also like to thank Dr. Alberto A. del Barrio for providing encouraging feedback and insightful comments that improved the content and the quality of this paper. This work is supported by the Spanish Ministry of Economy and Competitivity under research grants TEC2012-33892, TIN2013-40968-P, and TIN2014-54806-R.

DISCLAIMER

The author's affiliation with The MITRE Corporation is provided for identification purposes only, and is not intended to convey or imply MITRE's concurrence with, or support for, the positions, opinions or viewpoints expressed by the author.

Approved for Public Release. Distribution Unlimited. Case Number: 16-1506.

REFERENCES

1. The Mathworks Inc. https://www.mathworks.com/, 2016.
2. SystemC. http://www.systemc.org/downloads/standards/systemc, 2016.
3. Basu, A., Bozga, M., and Sifakis, J. Modeling heterogeneous real-time components in bip (2006). 3–12.
4. Beydeda, S., Book, M., and Gruhn, V., Eds. *Model-Driven Software Development*. Springer, 1998.
5. Eker, J., Janneck, J. W., Lee, E. A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., and Xiong, Y. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE 91*, 1 (2003), 127–144.
6. Friedenthal, S., Moore, A., and Steiner, R. *A Practical Guide to SysML*. MK/OMG Press, 2012.
7. Huang, D., and Sarjoughian, H. Software and simulation modeling for real-time software-intensive systems. In *Proceedings of 8th IEEE Symp. on Distributed Simulation and Real-time Applications* (2004).
8. Mittal, S. Model engineering for cyber complex adaptive systems. In *EMSS* (2014).
9. Mittal, S., Risco, J. L., and Zeigler, B. P. Devs-based simulation web services for net-centric t&e. In *Proceedings of the 2007 Summer Computer Simulation Conference*, SCSC '07, Society for Computer Simulation International (San Diego, CA, USA, 2007), 357–366.
10. Mittal, S., Risco-Martín, J. L., and Zeigler, B. P. Devsml: Automating devs execution over soa towards transparent simulators. In *Proceedings of the 2007 Spring Simulation Multiconference - Volume 2*, SpringSim '07, Society for Computer Simulation International (San Diego, CA, USA, 2007), 287–295.
11. Mittal, S., Ruth, M., Pratt, A., Lunacek, M., Krishnamurthy, D., and Jones, W. A system-of-systems approach for integrated energy systems modeling and simulation. In *Proceedings of the Conference on Summer Computer Simulation*, SummerSim '15, Society for Computer Simulation International (San Diego, CA, USA, 2015), 1–10.
12. Moallemi, M., and Wainer, G. A. A simplified real-time embedded DEVS approach towards embedded and control design. In *Proceedings of the 2009 Winter Simulation Conference* (2009).
13. Moallemi, M., and Wainer, G. A. Modeling and simulation-driven development of embedded real-time systems. *Simulation Modelling Practice and Theory 38* (2013), 115–131.
14. Risco-Martín, J. L., Cruz, J. M., Mittal, S., and Zeigler, B. P. eUDEVS: Executable UML with DEVS Theory of Modeling and Simulation. *SIMULATION 85*, 11-12 (June 2009), 750–777.
15. Risco-Martín, J. L., and Mittal, S. xDEVS. https://github.com/jlrisco/xdevs, 2016.
16. Tanenbaum, A. S., and Bos, H. *Modern Operating Systems*. Pearson, 2014.
17. Wainer, G. Applying modelling and simulation for development embedded systems. In *2013 2nd Mediterranean Conference on Embedded Computing (MECO)* (2013), 1–2.
18. Wainer, G. A. *Discrete-Event Modeling and Simulation: A Practitioner's Approach*. CRC Press, 2009.
19. Zeigler, B. P., Praehofer, H., and Kim, T. G. *Theory of Modeling and Simulation. Integrating Discrete Event and Continuous Complex Dynamic Systems*, 2 ed. Academic Press, 2000.