Proceedings of the 2015 Winter Simulation Conference
L. Yilmaz, W. K. V. Chan, I. Moon, T. M. K. Roeder, C. Macal, and M. D. Rossetti, eds.

DEVS MODELLING AND SIMULATION FOR DEVELOPMENT OF EMBEDDED SYSTEMS

Gabriel Wainer

Dept. of Systems and Computer Engineering
Carleton University
1125 Colonel By Dr. Ottawa
ON K1S5B6, CANADA

ABSTRACT

Embedded systems development has interesting challenges due to the complexity of the tasks they execute. Most of the methods used for developing embedded applications are either hard to scale up for large systems, or require a difficult testing effort with no guarantee for bug-free software products. Instead, construction of system models and their analysis through simulation reduces both end costs and risks, while enhancing system capabilities and improving the quality of the final products. M&S let users experiment with "virtual" systems, allowing them to explore changes, and test dynamic conditions in a risk-free environment. We present a Model-driven framework to develop cyber-physical systems based on the DEVS (Discrete Event systems Specification) formalism. This approach combines the advantages of a simulation-based approach with the rigor of a formal methodology. We will discuss how to use this framework to incrementally develop embedded applications, and to seamlessly integrate simulation models with hardware components. Our approach does not impose any order in the deployment of the actual hardware components, providing flexibility to the overall process.

1 INTRODUCTION

Cyber-physical and embedded systems development poses some interesting challenges due to the complexity of the tasks they execute. Formal methods have showed promising results in this area; nevertheless, they are difficult to apply when the complexity of the system under development scales up. . Most existing methods for developing these systems are either hard to scale up for large systems, or require a difficult testing effort with no guarantee for bug-free products. Instead, in recent years, systems engineers have relied on the use of modeling and simulation (M&S) techniques in order to build better applications. Construction of system models and their analysis through simulation reduces both end costs and risks, while enhancing system capabilities and improving the quality of the final products. M&S let users experiment with "virtual" systems, allowing them to explore changes, and test dynamic conditions in a risk-free environment. This is a useful approach, moreover considering that testing under actual operating conditions may be impractical and in some cases impossible. Here we present a Model-driven framework to develop embedded systems based on the DEVS (Discrete Event systems Specification) formalism [1].

DEVS provides a formal foundation to M&S that proved to be successful in different complex systems. This approach combines the advantages of a simulation-based approach with the rigor of a formal methodology. Another advantage of using DEVS is that different existing techniques (Bond Graphs, Cellular Automata, Partial Differential Equations, etc.) have been transformed into DEVS models. We discuss how to use this framework to incrementally develop embedded applications, and to seamlessly integrate simulation models with hardware components. Our approach does not impose any order in the deployment of the actual hardware components, providing flexibility to the overall process. The use of DEVS improves reliability (in terms of logical correctness and timing), enables model reuse, and permits reducing development and testing times for the overall process. Consequently, the development cycle is shortened, its cost reduced, and quality and reliability of the final product is improved.

2 BACKGROUND

Because of the growing complexity of embedded systems and their need for high reliability, their software development is still time consuming, error prone, and expensive. Many techniques have been proposed and used in practice to check correctness of embedded software. Current embedded Engineering methodologies use modeling as a method to study and evaluate different system designs before building the real application. In this way, real systems would have a very high predictability and reliability. In order to apply this methodology, a designer must abstract the physical system at hand and build a model for it, then combine this with a model of the proposed controller design. Then, different techniques can be used to reason about these models and gain confidence in its correctness. Informal methods usually rely on extensive testing of the systems based on system specification [2]. These methods have limitations because, in order to guarantee software reliability, we need to apply exhaustive testing to the software component, using all possible input combinations, which is a costly process. Many techniques have been proposed to enable a practical alternative to this exhaustive software testing [3]. However, we cannot guarantee a full coverage of all possible execution paths in a software component, thus leaving us with limited confidence in our software correctness.

Formal software analysis use is growing as an alternative, as this technique allows full verification of software components to be free of errors. In last decades, these techniques have matured to be used in some industrial capacity for software and hardware correctness verification [4]. Formal techniques can be used to prove the correctness of systems specifications. Nevertheless, they are usually constrained in their application, as they do not scale up well. Likewise, the designers need a high level of expertise in applying these techniques.

Formal verification techniques are of two main types, *deductive* or *algorithmic*. Deductive techniques rely on representing the system and its specification with logic rules, and then try to deduct a proof of system correctness. Algorithmic techniques rely on molding the system in a graphical form, and coding specifications in logical queries. Then an algorithm for *reachability analysis* searches the graph space for nodes reachable from initial system configuration that satisfy specification queries. This method is also called *model checking*. New theoretical advances in model checking allow guaranteeing certain properties about models of such systems using a formal approach. Model checking techniques can be automated to improve the work of the software engineer. Timed automata (TA) theory [5], in particular, has provided many practical results in this area. However, there is still a gap between a system model that is checked as an abstract entity, and the actual system implementation code to be run on a target platform.

A different approach to deal with these issues considers using Modeling and Simulation (M&S) to gain confidence in the model correctness. The use of M&S is not new, and systems engineers have often relied on these methods in order to improve the study of experimental conditions during model definition. The construction of system models and their analysis through simulation reduces both end costs and risks, while enhancing system capabilities and improving the quality of the final products. M&S let users experiment with "virtual" systems, allowing them to explore changes, and test dynamic conditions in a risk-free environment. This is a useful approach, moreover considering that testing under actual operating conditions may be impractical and in some cases impossible. Nevertheless, no practical, automated approach exists to perform the transition that exists between the modeling and the development phases, and this often results in initial models being abandoned, resulting in increased initial costs that project managers usually try to avoid. Simultaneously, M&S frameworks are not as robust as their formal counterparts are.

The DEVS formalism provides a formal M&S framework that can be used to deal with these issues. DEVS was originally defined in the '70s as a discrete-event modelling specification mechanism [1]. It is a systems theoretical approach that allows the definitions of hierarchical modular models that can be easily reused. A real system modeled with DEVS is described as a composite of submodels, each of them being behavioral (atomic) or structural (coupled), as seen in Figure 1. Each basic model consists of a time base, inputs, states, outputs and functions to compute the next states and outputs. As the formalism is closed under closure, coupled models can be integrated into a model hierarchy.

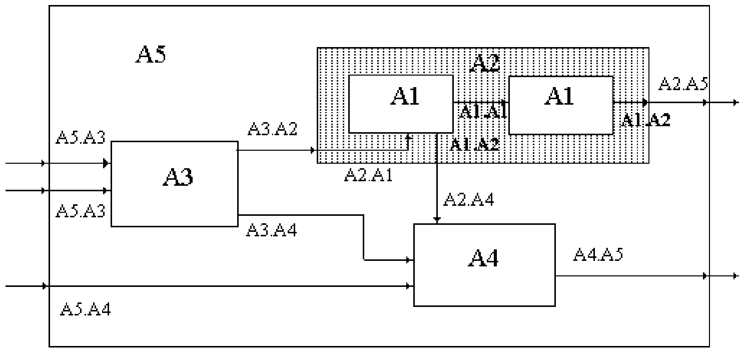

Figure 1. Coupling of DEVS models (A1, A3, A4: atomic models) [7].

A DEVS atomic model is formally described by:

$$M = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D >$$

X: input events set;
S: state set;
Y: output events set;
$\boldsymbol{\delta_{int}}$: S → S, internal transition function;
$\boldsymbol{\delta_{ext}}$: Q x X → S, external transition function; where
Q = { (s, e) / s ∈ S, and e ∈ [0, D(s)]};
λ: S→Y, output function; and
D: S → $\boldsymbol{R}_0^+ \cup \infty$, elapsed time function.

Each model is seen as having input and output ports to communicate with other models. The input and output events will determine the values to appear in those ports. The input external events are received in an input port, and the model specification should define the behavior under such inputs. The internal events produce state changes, whose results are spread through the output ports. The ports influences will determine if these values should be sent to other models.

A basic model can be integrated with other DEVS basic models to build a structural model. These models are called coupled, and are formally defined as:

$$CM = < X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\}, select >$$

X is the set of input events;
Y is the set of output events;
D is an index for the components of the coupled model, and
∀i ∈ D, $\mathbf{M_i}$ is a basic DEVS model, where
$M_i = < X_i, S_i, Y_i, \delta_{inti}, \delta_{exti}, ta_i >$
$\mathbf{I_i}$ is the set of influencees of model i, and ∀ j ∈ I_i, and
$\mathbf{Z_{ij}}$: $Y_i \rightarrow X_j$ is the i to j translation function.
Finally, **select** is the tie-break selector.

A main advantage of the DEVS paradigm is that the models can be specified independently of the simulation mechanism. Zeigler also suggested an abstract simulation mechanism that will be briefly introduced in this section, as the tool here presented uses it.

The simulation process begins by initializing all the component models. The state of each basic model is defined and the next internal transition for each of them is computed. The abstract simulator analyzes the external events and the scheduled internal transitions, and chooses the first model to be activated (called the **imminent** model). In the simulated time t, each component M_i has a state s_i, and has elapsed time e_i. The next event in the system will be the one with lower scheduled time. If there is more than one component with that time, the select function will be used to select the imminent. Once chosen, the imminent model is activated. If a basic model receives an external event $e \in X$, the model executes the external transition function δ_{ext}. As a result, the next internal event can be re-scheduled.

When the time for an internal even arrives, the imminent model must execute its internal transition function. The first step is to execute the output function λ and to generate an output event $y \in Y$. Each output is sent to the influencees as a translated input, using the Z_{ij} translation function. After, the internal transition function δ_{int} will execute, resulting in a state change and the scheduling of a new internal transition function. The behavior for each of the internal and external transition functions is dependent of the basic model behavior.

2.1 DEVS MODEL DEFINITION IN CD++

CD++ implements the DEVS theory. It allows defining models according to the specifications introduced in the previous section [6]. The tool is built as a hierarchy of models, each of them related with a simulation entity. Atomic models can be programmed and incorporated onto a basic class hierarchy programmed in C++. A specification language allows defining the model's coupling, including the initial values and external events.

Model definition in C++ allows the user great flexibility to define behavior. Nevertheless, a non-experienced user can have difficulties in defining models using this approach. The provision of graphical notations can provide the modeler with a powerful tool to define models. Graph-based notations have the advantage of allowing the user to think about the problem in a more abstract way. Therefore, we have used an extended graphical notation to allow the user define atomic models behavior. Each graph defines the state changes according to internal and external transition functions, and each is translated into an analytical definition. For instance, the Figure 1 shows a state called "Start", whose duration is 15 time units.

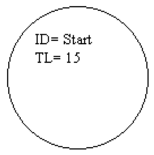

Figure 2: State Graphical Notation: Identifier, Time Length [7]

The syntax for the state analytical specification is:

```
state : stateId …
stateId : lifetime
```

Each model includes an interface with input/output ports. Ports are described by including their name and a type, based on the formal specification for DEVS models. They are specified as:

```
in  : portId:type  portId:type …
out : portId:type  portId:type …
```

Internal transition functions are represented by arrows connecting two states. Each of them can be associated to pairs of ports with values (p,v) corresponding to the output function. The syntax for the output function values is **`p!v.`** For instance, this figure represents that the model will change from state A to state B after 2 time units. First, the output function will send the value 8 through the port q1; 4 through the port q2, and 12 through the port q3.

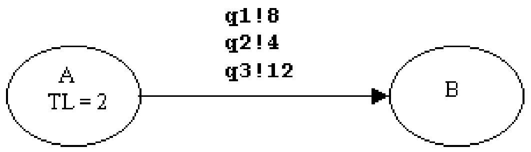

Figure 3: Definition of an Internal Transition Function [6, 7]

The syntax for the internal transition function construction is the following:

```
int : startState endState [outPort!value]+
```

Here we indicate the origin and destination states, and a port list with the corresponding values. For instance, the model in the previous figure can be described as:

```
int : A B q1!8 q2!4 q3!12
```

External transition functions are represented graphically by a dashed arrow connecting two states. The notation used to represent ports and expected values is similar to the one used for external transition, but replacing the exclamation mark by a question mark: `p?v [`t_i`..`t_f`]`. Here, ti...tf represent the initial and final expected simulated times for the external transitions. These values allow validating the timing of the models, raising an error if an external transition comes out of time. The syntax for this construction is the following:

```
ext : startState endState inPort value timeRange
```

It describes the origin and destination states, an input port and a time range counted since the instant arriving to the start state.

All these constructions can be combined to define the behavior of atomic models. For instance, the following Figure 4 represents a simple model using all the constructions:

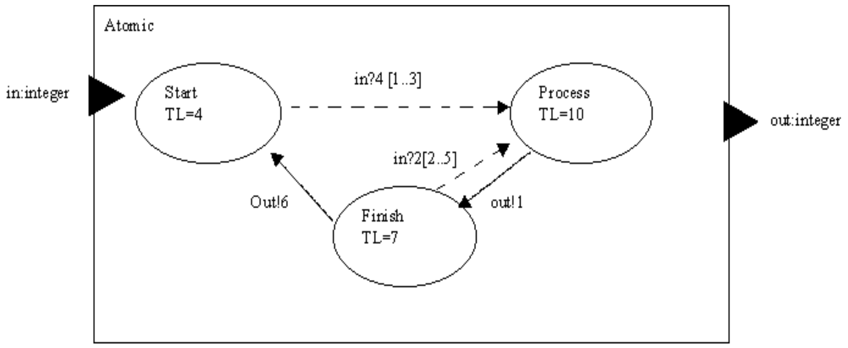

Figure 4: Definition of an Atomic Model [6, 7]

This model can be formally specified as:

$$\text{Simple\_Proc} = < I, X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D >$$

I = <P^X, P^Y> where P^X = { ("in", integer) }; P^Y = { ("out", integer) };
X = Y = $\mathbb{Z}$; **S =** { Start, Process, Finish };

```
δext(s,e,x):
    case port (in) {
        4:  if (e < 1 or e > 3) error();
            phase = Process;
            D = 10;
        2:  if (e < 2 or e > 5) error();
                if (phase != finish)
                    phase = Process;
                    D = 10;           }
$\lambda(s)$:
    case (phase) {
        Finish:  send(out, 6);
        Process: send(out, 1);    }

$\delta_{int}(s)$:
    case (phase)  {
        Process: phase = Finish;  D = 7;
        Finish: phase = Start; D = 4;                    }
```

For instance, in this case we see that being in the *Start* phase, we need to receive an input (and trigger the external transition function) through port 4 between time units 1 and 3; otherwise there will be an error. If the input is received, we change to the phase *Process*, which lasts 10 time units. When this time is consumed, we trigger the output function (which outputs a value of 1 through port *out*), and an internal transition that switches to phase *Finish.* We schedule the next internal transition in 7 time units. At that point, we can receive an input through port 2 (between time units 2-5; otherwise it is an error), or wait until the internal transition is triggered. In the first case, we switch back to the *Process* phase. If we do not receive an input, we generate an output (with value of 6 through port *out*), we go back to the *Start* state.

The previous graphical specification can be used to generate the following text definition, which is equivalent to the previous specifications [7]:

```
[exampleGG]
in: in
out: out
state: Start Process Finish
int: Process Finish out!1
int: Finish Start out!6
ext: Start Process  in 4
ext: Finish Process  in 2
Start: 4
Process: 10
Finish: 7
```

Figure 5: Text Definition of the example in Figure 4 [7]

After each atomic model is defined, they can be combined into a coupled model, which are defined using a specification language specially defined with this purpose. The language was built following the formal definitions for DEVS coupled models. Therefore, each of the components defined in section 2 for coupled models can be included.

As showed in the formal specifications presented earlier, four properties must be configured: components, output ports, input ports and links between models. We use the following syntax:

`Components`: it describes the models integrating a coupled model. The syntax is: `model_name@class_name`, allowing more than one instance of the same model using different names. The class name reference to either atomic or coupled models.

`Out`: it defines the names of output ports.

`In`: it defines the names of input ports.

`Link`: it describes the internal and external couplings. The syntax is: `source_port[@model] destination_port[@model]`. The name of the model is optional and, if it is not indicated, the coupled model being defined is used.

```
[top]
components: DeparturesQ@StoppableQueue Track@Track LandingQ@StoppableQueue
             ControlTower@ControlTower Hangar
in : In       out : Out
link : out@DeparturesQ In_d@ControlTower    link : out@LandingQ In_a@ControlTower
link : In in@LandingQ                       link : Out_a@Track In@Hangar
link : Out_d@Track Out                      link : Done_a@ControlTower in@LandingQ
link : Stop_a@ControlTower stop@LandingQ    link : Out@Hangar in@DeparturesQ
link : Departing@ControlTower Departing@Track
link : Landing@ControlTower Landing@Track   link:Done_d@ControlTower in@DeparturesQ
link : Stop_d@ControlTower stop@DeparturesQ

[Hangar]
components : selector@selector deposit1@queue  deposit4@queue deposit3@queue
             deposit2@queue   Chosen@DeMux
in : In   out : Out
link : out1@selector in@deposit1     link : out2@selector in@deposit2
link : out3@selector in@deposit3     link : out4@selector in@deposit4
link : out@deposit1 in1@Chosen       link : out@deposit4 in4@Chosen
link : out@deposit3 in3@Chosen       link : out@deposit2 in2@Chosen
link : In in@selector                link : out@Chosen Out
```

Figure 6: Coupled Model Definition

Let us consider the specification in the previous figure, which represents a small Airport. The **`[top]`** model always defines the coupled model at the top level. The control tower is connected to two queues: one for departures, and the other for arrivals. These queues are used to model the time employed by planes to enter or leave the airport area. The control tower is also connected to a model representing the track. Every time a plane is authorized to depart or land, the track model is activated. Finally, all landed planes go to a Hangar for maintenance. A plane can only leave after service. The hangar can be defined as an atomic model, or as a coupled model with different service stations for the planes. CD++ uses this information to generate instances of previously defined atomic models, or creates new instances of coupled models that can be later reused to form other multicomponent models.

2.2 Simulation Algorithms

As stated, the tool is based on the DEVS formalism, and provides an environment to build discrete events models. The system architecture was built using the abstract simulator concepts described in [1], and it can be summarized as in Figure 7.

There are two basic classes: Models and Processors. The Models classes are devoted to define the conceptual models, and the Processors implement the simulation mechanism. There are different kinds of simulation Processors: Simulators, Coordinators and a Root-Coordinator. These processors are related with the models in pairs: a Simulator is associated with an Atomic model, and a Coordinator with a Coupled model. Each existing model will have a unique associated processor.

Model has instance variables *processor* (to identify its associated processor), *parent* (link to the coupled model containing this model), *inport* and *outport* (to specify model interaction).

The *Atomic* class is used to represent the atomic basic models. The methods *int-transfn*, *ext-transfn*, *outputfn* and *time-advancefn* represent the internal transition, external transition, output and time advancement functions respectively. The functions will be overloaded by the modeller to define the desired behavior depending on the system to be modeled. *Coupled-Model* implements the hierarchical constructions defined by the modelling formalism. A coupled model is defined by specifying its components (*children*), and the coupling relationships. The coupling is specified by the *receivers* and *influences* instance variables, which allow the definition of the Z_{ij} function.

The *Processors* are built to execute the abstract simulation procedures by implementing the DEVS theoretical concepts. Following these concepts, *Simulators* and *Coordinators* are built to manage atomic and coupled models, as stated earlier. The *Root-Coordinator* drives the simulation in its global aspects. It maintains the global time, and it is in charge of the start and finish of the simulation. It also collects the output results. It is related with the highest level coupled model and its corresponding coordinator.

The coupling relationship is recorded in the instance variables *devs-component* and *processor* of the Processor and Model respectively. The *parent* variable is used to indicate the parent processor in the simulators hierarchy. The times of the last event and the next event are recorded to identify the imminent children, as to verify correctness in the message simulated times.

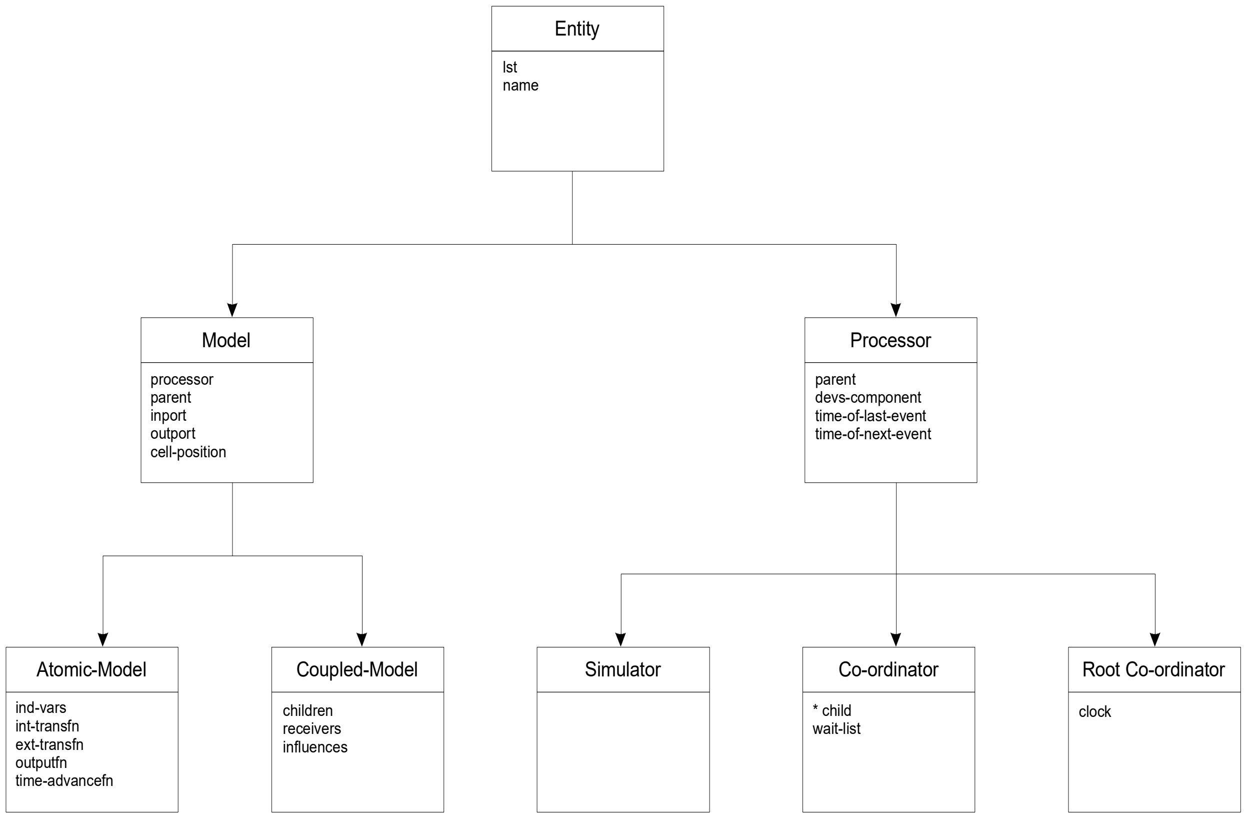

Figure 7. Basic class hierarchy.

The simulation process is carried out by data transfer carried out through message passing. The messages include information related with the message's origin, the time of the related event, and a content consisting of a port and a value.

There are four different messages:

*: used to signal a state change due to an internal event;

X: used when an external event arrives (including its port and value);

Y: model's output; and

done: indicating that a model has finished with its task.

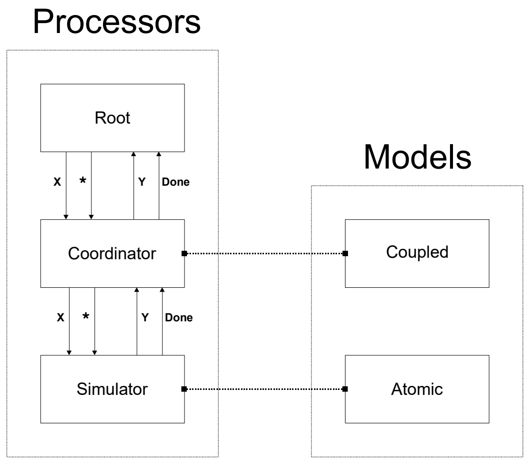

Figure 8. Relationship between Models and Processors.

Simulation advances through message passing between the Processors. When the imminent model is selected, a *-message is sent to its simulator, passing through the middle level coordinators.

When an external message arrives, an X-message is consumed and the external transition function executed. The simulators return done-messages and Y-messages that are converted to new *-messages and X-messages, respectively. The messages are translated using the cell coupling previously defined.

3 MODELLING AND SIMULATION FOR DEVELOPMENT OF EMBEDDED SYSTEMS

In recent years, there have been a number of efforts focusing on defining new formal methods and tools for development of embedded systems, in particular those with real-time constraints. As discussed earlier, most existing methods are still hard to scale up, and they require expensive testing efforts with no guarantees for bug-free products. Instead, systems engineers have often relied on modeling and simulation (M&S) techniques to improve development and obtain higher quality products. M&S-based testing is widely used for the early stages of a project; however, when the development tasks switch towards the target environment, early models are often abandoned. In order to deal with these issues, we introduced a Discrete-Event Modeling methodology based on DEVS, which combines the advantages of a practical approach with the rigor of a formal method; in which one consistently use models throughout the development cycle.

Building system models and their analysis through simulation (M&S) reduces cost and risk, allowing exploring changes and testing of dynamic conditions in a risk-free environment. This is a useful approach, moreover considering that testing under actual operating conditions may be impractical and in some cases impossible. Despite the net gains, most project managers are reluctant to use the techniques because they require extra initial resources in the construction of simulation models that will not be part of the delivered application.

Our idea is to enable the incremental construction of embedded applications using a discrete-event modeling architecture *both for simulation and as the final target architecture for products*. DEVS is a well-defined formalism that is expressive, operates at a high level of specification, and can be used both to represent computing systems and to describe physical systems. DEVS models have a rich structural representation of components, and formal means for explicitly specifying their timing, which is central in the design of cyber-physical systems. The use of DEVS offers the following advantages:

- *Reliability*: logical and timing correctness rely on DEVS's system theoretical roots and sound mathematical theory.
- *Model reuse*: DEVS has well-defined concepts for coupling of components and hierarchical, modular model composition.

- *Hybrid modeling and knowledge reuse*: it has been shown that DEVS is the most general discrete event formalism, and many embedded systems methods have been mapped to DEVS (i.e., VHDL, Petri Nets, Timed Automata, State Charts, etc.). An application can be built using different methods while keeping independence at the level of the executive, using the most adequate technique for each component, enabling the reuse of existing expertise on each part of system architecture.
- *Process flexibility*: these hybrid modeling capabilities are transparent for the executive, which is defined by an abstract mechanism that is independent from the model itself. Existing DEVS tools have showed their ability to execute such variety of models with high performance.
- *Testing*: the construction of experimental frames (that is, sets of conditions under which the system is observed or experimented with) can be automated.

The method uses M&S for the initial stages, and replaces models incrementally with hardware surrogates *without modifying the original models*. The transition can be done in incremental steps, incorporating models in the target environment after thorough testing in the simulated platform, allowing reuse of early models throughout the process. The approach does not impose any order in the deployment of the actual hardware, providing flexibility. Figure 9 shows the architecture of the methodology.

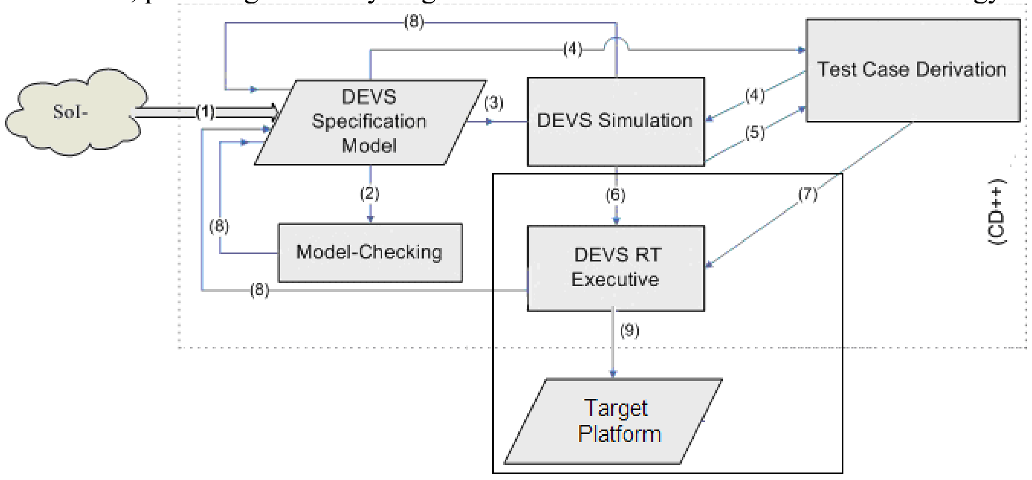

Figure 9. Development cycle [8].

Initially (1), we define a specification model of the System of Interest (SoI) using a formal model (using DEVS or alternative techniques translated to DEVS). Once the specification model is complete, model-checking is used for validation of the model properties (2). *The same models* are then used to define a DEVS simulation of the behavior of the different submodels under specific loads (3). Thus, we study system properties analytically, and complement the proofs using simulation models, which can also be used for hardware/software codesign activities (and with training purposes).

The same DEVS specification model can be used to derive test cases (4), which can be reused for the simulation studies. Deriving test cases from both the model (4) and from the simulation results (5) allows us to check that the models conform to the requirements. Once we are satisfied with both analytical and simulated results, *the same components* are incrementally incorporated into the target platform. A real-time Executive (6) executes the models on a hardware surrogate (9). If the hardware is not readily available, the software components can still be developed incrementally and tested against a model of the hardware to verify viability and take early design decisions. As the design process evolves, both software and hardware models can be refined, progressively setting checkpoints in real prototypes. The executive allows to execute dynamic models and to schedule static and dynamic tasks (activating non-mandatory parts or alternate models under transient overloads).

At this point, for those parts of the model that are still unverified (in the formal and simulated environments), testing will increase the confidence of the engineer into the implemented system (7). Any modifications require going back to *the same model specifications* (8), which ensure that we can provide a consistent set of apparatuses throughout the development. As in any software life cycle, this process is cyclic, allowing refinement following a spiral approach. On each cycle of the spiral, we have a prototype application consisting of software/hardware components interacting with simulated subcomponents.

3.1 E-CD++: and environment for modeling embedded systems

The E-CD++ tool provides a mechanism to implement DEVS models (which can be implemented in C++ or using a built-in language) using DEVS formal specifications and following the approach defined above in Figure 9. For instance, in the following example, the *ButtonInputModule* model shows parts of a component of a cruise control system (CCS) [8, 9].

```
ButtonInputModule::ButtonInputModule ( const string &name ) : Atomic( name ),
    in_BUTTON( addInputPort( "in_BUTTON") ), out_ON( addOutputPort( "out_ON") ),
    out_RESUME( addOutputPort( "out_RESUME") )
{   reactionTime = VTime( 0, 0, 0, 15 );  }

Model &ButtonInputModule::externalFunction ( const ExternalMessage &msg ) {
  if( msg.port() == in_BUTTON )   {
    inType=(int)msg.value();
    holdIn( active, reactionTime );
  } }

Model &ButtonInputModule::outputFunction ( const InternalMessage &msg ) {
  switch(inType) {
    case ON: //take action {
      sendOutput( msg.time(), out_ON, HIGH) ;
      break; }
    case OFF: //take action {
      sendOutput( msg.time(), out_OFF, HIGH) ;
      break; }
  ...} }

Model &ButtonInputModule::internalFunction   ( const InternalMessage & ) {
  passivate();}
```

embedded-CD++ integrates simulation models and hardware components for the methodology. Figure 10 outlines the software hierarchy generated by the executive to execute the CCS model. The *Root Coordinator* manages the interaction with an Experimental Frame (used to test the model). A *Coordinator* synchronizes subcomponents. Timing constraints can be associated to each external input, and when the processing of such an event is completed, the Coordinators checks if the deadlines were met (to obtain performance metrics, or to provide alternate actions if a deadline be missed).

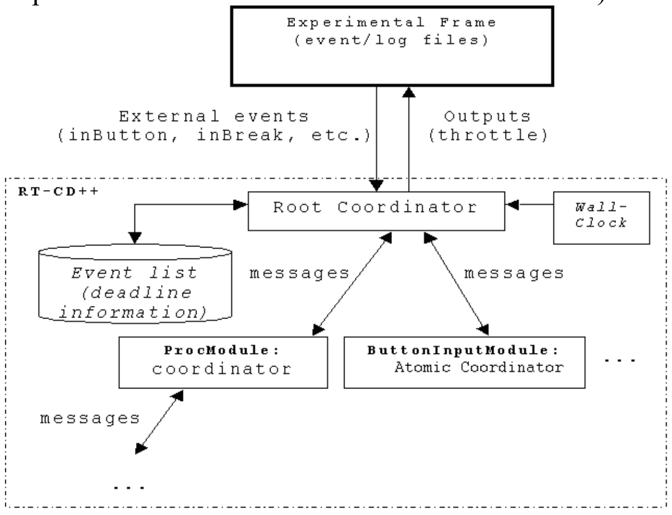

Figure 10. embedded-CD++ simulation scheme [8]

This was the base for Embedded CD++ (E-CD++), designed to execute in embedded environments interacting with hardware surrogates. The time advance function is tied to the real-time clock, and inputs/outputs can interact with external devices, including runtime checks of the timing deadlines. The engine runs on a single board computer (SBC), and an Eclipse-based IDE helps non-expert users (including a graphical environment based on DEVS-graphs) following the methodology.

We will show how to use our methodology to develop embedded applications incrementally, integrating simulation models and hardware components. Initially, we develop models entirely in E-CD++, and we replace them with hardware surrogates at later stages of the process, making the transition in incremental steps, incorporating models in the target environment with hardware-software components after thorough testing in the simulated platform (*using the specification models throughout the process*).

On http://cell-devs.sce.carleton.ca/mediawiki/index.php/Robocart the reader will find a sample application built as an experimentation environment for the construction of robotic controllers.

Once satisfied with the overall behavior of the simulated model, we progressively replace simulated components with their hardware counterparts. The first step was to replace the button controller model (i.e., we use a keypad to receive requests, and we send them to the simulated model). The remaining components are unchanged. Replacing the component is straightforward: we only need to remove the original component from the coupled model definition. Testing the model only requires reusing the experimental frames used for simulation. After conducting extensive tests, we can also remove the remaining components to the microcontroller [8].

Our method was applied on a prototype for an application (embedded in a Network Processor) managing the Quality of Service (QoS) of high-speed data-flows. We want to enforce low-level traffic shaping actions according to both high-level QoS policies (which assign network resources to the competing traffic) and the evolving performance of traffic. At the higher levels we find coarse-grained global policies (with a few changes per day). At lower levels, QoS shaping algorithms modify the assignment of network resources to data-flows (every few seconds). At the lowest levels, specific algorithms take granular decisions at the microsecond time scale, on a per-packet basis. We designed a prototype QoS shaper system that accepts policies from higher levels while knowing the status of the lowest level traffic (the *packet drop-rate*). Depending on the policies set and the drop-rate, information is sent to the lower packet-level algorithms to enforce granular decisions. I/O information is sent through real-time ports by E-CD++. When a QoS policy change comes from the higher levels, the model's parameters get different values, generating an adapted QoS Controller with new behavior.

We used an Intel IXP2400 Network Processor (an OC-48/2.5 Gbps line rate packet chip structured in two levels: the *slow data path* with an Intel XScale Core processor, and the *fast data path*, with 8 multi-threaded micro engines - ME). The IXP2400 allows implementing reconfigurable rule engines that can be adapted on demand with high performance for the packet handling tasks. We embedded E-CD++ on the XScale Core processor as Core Component (either in standalone mode or linked with the MEs). The models executed by E-CD++ and the *microblocks* (software units of code that run multithreaded inside each of the MEs) interact in real-time.

We followed the steps for an incremental co-development prototype of the system. On a first stage, we verified the system behavior in the E-CD++ standalone simulator. The QoS Actuator and Traffic Sensor models commands and sense the drop-rate values respectively. They talk to their counterparts in the Packet Processing system, a QoS Shaper and a Metering System that know how to communicate with the QoS Actuator and Traffic Sensor.

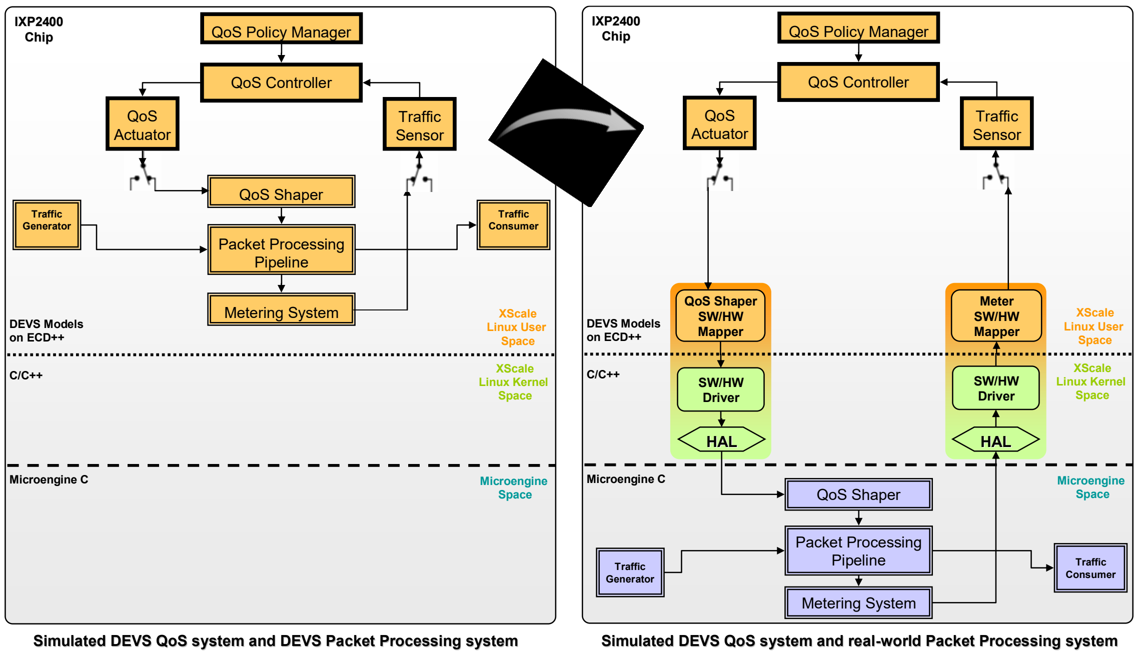

Figure 11. Modeling a QoS processing system [8, 9]

Once the functionality of the QoS controller was verified, we moved it into the IXP chip. Then the micro engines take the place of their DEVS counterpart models (implementing traffic generation, traffic consumer, QoS Shaper, Metering System and Packet Processing Pipeline). The QoS Actuator and Traffic Sensor were redirected to special software/hardware *Mapper* models (signal forwarders that know how to communicate with the IXP libraries for doing the mapping). The scenario switch is completely transparent for the QoS Controller system. Finally, by means of a constant-rate packet-dropping generator code (running on the MEs), the whole system was validated. In the meantime, other hardware pieces were reprogrammed by a separate development team to prepare the RED algorithm to effectively react to the new Shaping commands, thus interleaving the software and hardware co-development process, and then starting a new incremental cycle of system verification and validation.

4 CONCLUSION

In recent years, the Software Engineering community has spent a tremendous effort in creating formal methods and tools for developing embedded systems, in particular, those with real-time constraints. Despite these efforts, most existing methods are still hard to scale up, and they require expensive testing efforts with no guarantees for bug-free products. Instead, systems engineers have often relied on modeling and simulation (M&S) techniques to improve the development task and obtain higher quality products. M&S-based testing is widely used for the early stages of a project; however, when the development tasks switch towards the target environment, early models are often abandoned. Using an M&S-based development methodology based on discrete-event systems specifications like the one presented here combines the advantages of a practical approach with the rigor of a formal method, in which one consistently use models throughout the development cycle.

The proposed methodology uses the DEVS formal modeling technique, improving the safety and development times of the simulations. An abstract simulation mechanism enables defining different simulation techniques without needing changing the models developed, as they follow the formal specifications of DEVS formalism. The models are easy to reuse thanks to the hierarchical construction. Consequently, the costs of development are reduced, the quality of the models improves, and non-expert users are able to start developing new applications with a fast learning curve. Several types of models can be integrated in an efficient fashion, allowing multiple points of view to be analyzed using the same model. The formalism allows improving the security and cost in the development of the simulations. Experimental results of application showed improvements for expert developers.

We discussed how to transition from simulation in a model-based environment, and then execute the same models directly in a hardware surrogate. We showed the use of DEVS to build this kind of models, which allowed us to develop incrementally different applications including hardware components and DEVS models. The transition from simulated models to the actual hardware can be incremental, incorporating deployed models into the framework when they are ready. This approach does not impose any order in the deployment of the hardware components, providing flexibility to the overall process. The use of DEVS improves reliability (in terms of logical correctness and timing), enables model reuse, and permits reducing development and testing times. Consequently, the development cycle is shortened, its cost reduced, and quality and reliability of the final product improved. We showed the use of these methods in the E-CD++ toolkit, in which embedded systems can be designed following DEVS-based methodologies, and be implemented on different hardware (FPGA, SBCs, general purpose processors or specialized ones like the IXA platform). The verified models can be deployed to the targets without modifying a single line of code. In this way, we can provide advanced capabilities for rapid prototyping and development.

5 REFERENCES

[1] B. Zeigler, T. Kim, H. Praehofer. "Theory of Modeling and Simulation". Academic Press 2000, ISBN-10: 0127784551.

[2] M. J. Rehman, F. Jabeen, A. Bertolino, and A. Polini, 2007, "Testing Software Components for Integration: a Survey of Issues and Techniques", Software Testing,Verification and Reliability. 2007, volume 17, issue 2, pages 95–133.

[3] Rainer Gerlich, Ralf Gerlich, Thomas Boll. 2007. "Random Testing: From the Classical Approach to a Global View and Full Test Automation". In Proceedings of the 2nd international Workshop on Random Testing: Co-Located with the 22nd IEEE/ACM international Conference on Automated Software Engineering (ASE 2007), Atlanta, Georgia.

[4] M. B. Dwyer, J. Hatcliff, R. Robby, C. S. Pasareanu, and W. Visser. 2007. "Formal Software Analysis Emerging Trends in Software Model Checking". In Proceedings of the 2007 Future of Software Engineering (FOSE '07). IEEE Computer Society, Washington, DC, pages 120-136.

[5] R. Alur, D. Dill. "Theory of Timed Automata". Theoretical Computer Science, volume 126, pg. 183-235, 1994.

[6] G. Wainer. "Discrete-Event Modeling and Simulation: a Practitioner's approach". CRC Press. Taylor and Francis. 2009.

[7] G. Wainer. "CD++: a toolkit to define discrete-event models". *Software, Practice and Experience*. Wiley. Vol. 32, No.3. pp. 1261-1306. November 2002.

[8] G. Wainer, R. Castro. "DEMES: a Discrete-Event methodology for Modeling and simulation of Embedded Systems". *Modeling and Simulation Magazine*. Society for Modeling and Simulation International. San Diego, CA. April 2011.

[9] Yinfeng Henry Yu, Gabriel Wainer, "eCD++: an engine for executing DEVS models in embedded platforms", Proceedings of the 2007 SCS Summer Computer Simulation Conference, San Diego, CA - 2007.