Simulation Modelling Practice and Theory 38 (2013) 115–131

Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

journal homepage: www.elsevier.com/locate/simpat

Modeling and simulation-driven development of embedded real-time systems

Mohammad Moallemi *, Gabriel Wainer

Systems and Computer Engineering Department, Carleton University, 1125 Colonel by Dr., Ottawa K1S 5B6, Canada

ARTICLE INFO

Article history:
Received 5 September 2012
Received in revised form 22 July 2013
Accepted 28 July 2013
Available online 30 August 2013

Keywords:
Discrete-event simulation
Embedded systems
Real-time simulation and control
Model-based approach

ABSTRACT

The design and development of embedded hard real-time (RT) systems is one of the complex development practices, because of the requirements of criticality and timeliness of these systems. One critical aspect of RT systems is the production of output before specified deadline. Formal methods are promising in dealing with the design issues of these applications, although they do not scale well for complex systems. Instead, Modeling and Simulation (M&S) provides a cost-effective approach to verify the design and implementation details of very Complex RT applications. M&S methods provide dynamic and risk-free testing environments to verify different scenarios, and they are used for feasibility analysis and verification of such systems. Nevertheless, the simulation models are usually discarded in the later phases of the development.

We present the application of an M&S-based method referred to as DEVSRT (Discrete EVent System Specifications in Real-Time) to solve the discontinuity between the simulation models and the final embedded application, in this paper. DEVSRT defines explicit deadline notation for DEVS transitions, draws a clear mapping between DEVS transitions and real-time tasks and provides a formal method and tool for integration of simulation models with the associated hardware components.

1. Introduction

Real-time (RT) and embedded systems are employed in various applications ranging from telecommunications, customer electronics, transportation, medical equipment, and automated systems. An RT system is formally defined by Liu as "a system that is required to complete its work and deliver its services on a timely basis" [1]. The system in which all timing constrains must be met are considered "hard real-time systems". Real-time application development has evolved rapidly because of the growing use of these applications. Many of these systems are deployed in embedded controllers working in hardware computing platforms with special configurations and interfaces. An "embedded system" is described by Steve Heath as "a microprocessor-based system that is built to control a function or a range of functions and is not designed to be programmed by the end user in the same way that a PC is" [2]. In the case of embedded systems with hard real-time constraints, the design decisions can lead to catastrophic consequences for infrastructures or lives [3]. The size, variety, and criticality of the computations carried out on these systems have attracted creative and visual design methods that increase their complexity, reliability and performance. The architecture of these systems usually integrates different types of hardware components such as processors, analog and digital components, as well as mechanical (e.g. sensors and actuators) and visual components, which demands increasingly challenging multidisciplinary design and development efforts [4].

* Corresponding author. Tel.: +1 3862267972.
E-mail addresses: moallemi@sce.carleton.ca (M. Moallemi), gwainer@sce.carleton.ca (G. Wainer).

1569-190X/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.simpat.2013.07.004

Nevertheless, because of the heterogeneity of these systems and their constraints (such as cost, time to market, and performance), their development cycle is still time consuming, error prone and expensive.

A solution that provides a correct framework for designing these systems is the adoption of formal methods [5], described as special cases of mathematical-based techniques for designing, development and verification of software and hardware systems [4]. They allow for appropriate mathematical specification and analysis of the designs, which can contribute to the reliability of the final product, yet they add to the complexity of the design by applying mathematical-based approaches, which makes it hard to prove, hence increasing the overhead of the development. Adding this overhead might increase the overall cost, nevertheless in some cases, spending these resources would result in an overall gain in the quality and cost of the final product. They are appropriate for systems with critical applications where safety and robustness is a foremost aspect. Nevertheless, these methods do not scale up well, as most formal proving mechanisms cannot provide formal proofs of correctness when the complexity of the system grows.

Instead, Modeling and Simulation (M&S) provides a practical solution in solving the above-mentioned difficulties. M&S is a useful tool for efficient analysis, design, verification and optimization of general dynamic systems. The use of M&S in software engineering reduces costs and risks, and allows for exploring different aspects of the system in a risk-free environment.

Formal M&S uses mathematical models to define the specifications of a system. This approach has shown promising results in making multidisciplinary system development tasks manageable [6]. Model-based design is a computational approach that provides a hierarchical design scheme in which higher abstract levels are branched into levels that contain more details; the system specifications are defined in a formal way at which new and distinguishing functionalities are conceived. Other advantages of Model-based development are applying formal model checking techniques at design time [7,8], the incremental refinement of the initial simulation models, the simulation-based validation and reuse of the existing models, the risk free testing of critical RT applications, and the increased reliability in the design. As discussed earlier, formal approaches as model checking add to the complexity of the design, and on the other hand, it increases the reliability and robustness of the final product. Formal M&S complements and enhances the pure formal method-based approaches, providing an alternative for the design of real-time and embedded systems.

The use of these techniques is often used in the early stages of the development of real-time and embedded systems. However, when the scope of the development moves towards the final architecture to be deployed on the actual target hardware, the early simulation models are abandoned, and the final system is redeveloped from scratch based on the results obtained during the simulation phase. Currently, existing development tools and methods do not support a simulation-based approach or they lack features providing model continuity from the requirement analysis stage of software development up to deployment into the target hardware [9]. Model continuity shortens the development process and speeds up the implementation phase. In the approach, we will show how M&S is a foremost component in the RT application development, and it goes further by utilizing the simulation models on the target platform. Although commercial tools like MATLAB/Simulink [10] and LabVIEW [11] provide good resources for these activities, they are mostly limited to simulation only and they do not directly support model continuity or model checking. On the other hand, approaches like UML-RT [12] can be used to develop software design models, or they are not suitable to be used as simulation models [13]. Also, they do not provide any means for modeling the environment surrounding the RT embedded application, while M&S-based methods do.

We introduce the ideas and applications of DEVSRT (Discrete-Event Systems Specifications in Real-Time), a domain extension to DEVS theory [6] for embedded real-time application development. DEVS provides a formal foundation to M&S that proved to be successful in different complex applications [3]. DEVSRT takes advantage of well-defined M&S properties and constructs of DEVS to design and interface embedded systems with the hardware and the environment under study. The DEVSRT approach provides the following advantages:

- The notion of a deadline is added to the DEVS formalism, making it appropriate for real-time system modeling and design. Based on DEVS computational properties, a set of assumptions is defined to be used in the design of a real-time application. DEVSRT uses DEVS formal outputs as the output signals of the real-time system, therefore a relative deadline is associated with each output produced at the end of each state.
- An efficient interfacing mechanism is added to DEVS theory, in order to satisfy our research motivations: (1) Model continuity from the simulation stage, up to the deployment in the target hardware. (2) The entire system is designed in a hardware-software co-design approach, in which the models represent different hardware and software components and are tested together as an integrated DEVS model. (3) Provides a hardware-in-the-loop simulation platform where some of the models act as the simulated plant components (in which the driver interfaces provide the electrical emulator signals) and are tested with the embedded system (controller) model to be deployed on the hardware.
- The Embedded CD++ tool (E-CD++) [14] is extended to implement the proposed DEVSRT framework for the formal development of embedded real-time applications. The new version of E-CD++ is implemented on a real-time kernel, incorporating real-time services.

We show how DEVSRT satisfies the motivations and present the real-time specifications added to the DEVS formalism to develop real-time applications. DEVSRT has been used to develop various real-time embedded systems on a variety of

hardware platforms (such as FPGAs, embedded boards, networking and robotic devices). On the other hand, a number of real-time prototypes have been built using this approach (e.g. [37,38]). We discuss a prototype application in the field of robotics, and we then discuss the development process and results using our method and tools.

2. Related work

The need for high quality software with no defect for real-time (RT) and embedded applications has evolved techniques in which system specifications are expressed using formal mathematical methods. On the other hand, the issue of consistency and traceability from the design stage to the deployment is also a challenge. There are no well-established techniques in M&S-based design schemes to bridge the gap between the modeling and the hardware deployment phases, nor techniques for mapping the model behavior to a RT task system to adopt task-scheduling algorithms in real-time operating systems. Because of these issues, M&S artifacts are often abandoned and not used for the development of the actual embedded system, resulting in extra development costs.

Various modeling methodologies have been introduced in literature, concerning the design and development of real-time and embedded software systems. Among the model-based approaches UML-RT (the Unified Modeling Language for Real-Time) [12] is an extension of UML modeling language which provides especial aspects for designing real-time systems. A comparison between DEVS and UML-RT [13] showed that features such as time, scheduling and performance coded using UML constructions are not formally defined. Instead, formal modeling methods like DEVS provide sound syntax/semantics for structure, behavior, time representation and composition, which lend themselves to well-defined computation. DEVS, however, is not intended for software design and development, and "it is key to support the transformation of simulation models to their software model counterparts and their complementary roles in handling modeling and computational complexity of embedded systems" [13].

Among other model-based approaches, the BIP (Behavior, Interaction, and Priority) methodology introduced in [15] allows modeling heterogeneous component-based real-time systems. Components are obtained as the superposition of three layers: behavior, specified as a set of transitions; interactions between transitions of the behavior and priorities, used to choose amongst possible interactions.

ECSL (Embedded Control Systems Language) supports software development for distributed embedded controllers [16]. ECSL offers a graphical modeling language built using the Generic Modeling Environment (GME), an open-source meta-programmable domain-specific design environment. It was designed in the context of embedded automotive systems with capabilities such as requirements specification, verification, mapping onto a distributed platform, scheduling and performance analysis.

Ptolemy II [17] is a structured and hierarchical method for modeling heterogeneous systems using a specific model of computation that covers the flow of data and control. SystemC [18] and Esterel [19] are system description languages that can be used for generating simulatable and executable models. They share some features and have their unique characteristics, and some two-way component mapping can be performed between these languages.

Matlab/Simulink® [10] is a commercial tool for modeling and simulating embedded systems, which offers a graphical interface for visual construction and integration of hardware blocks. Simulink® can be integrated with many other tools, such as Stateflow®, Simulink Coder®, and Embedded Coder® for event-based modeling, physical modeling, and code generation. Simulink is mainly used for simulating real-time systems however, the accompanied code generation tool produces C/C++ code for embedded processors. Nevertheless, the generated code has limited usage, does not support all the functionalities of the Simulink blocks and lacks means for verification.

None of the above-mentioned modeling approaches provides direct model continuity, whereas DEVSRT allows for direct use of the simulation models as the final target architecture. DEVSRT also provides a straightforward hardware–software co-design capability [20] (i.e., co-specification, co-synthesis, co-simulation and co-refinement) in a more abstract level as well as the hybrid testing of simulation models with real hardware. The use of DEVS simplifies the transformation of the models from various other formal methods, supporting heterogeneous systems design, implementation, and reuse, which can be useful in the case of embedded system development. DEVS, as a simulation methodology, not only allows for simulation-driven software development but also supports M&S of an entire system and its surrounding environment. This allows for verification of the software in a simulated environment with changing conditions.

2.1. Introduction to DEVS

DEVS [6] is a formal M&S methodology, based on generic dynamic systems, including well-defined coupling of components, hierarchical, modular construction, support for discrete event approximation of continuous systems and support for repository reuse. A real system modeled with DEVS is described as a composite of sub-models, each of them being behavioral (atomic, see Fig. 2.1) or structural (coupled).

A Parallel DEVS (P-DEVS) [28] model [9] is described as a set of basic atomic and coupled models. Atomic models are still the most basic constructions, which can be combined with other models into coupled models. The P-DEVS atomic model has the following structure:

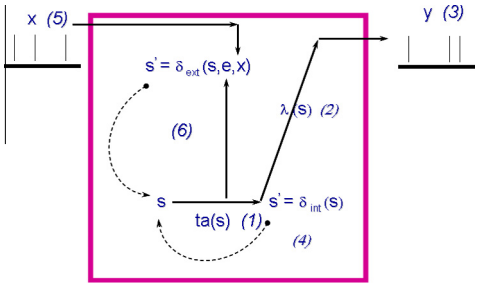

Fig. 2.1. DEVS atomic component state transition sequence.

AM = < X_M, Y_M, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta >, where:
$X_M = \{(p, v) | \ p \in IPorts, v \in X_p\}$ is the set of input ports and values;
$Y_M = \{(p, v) | \ p \in OPorts, v \in Y_p\}$ is the set of output ports and values;
S: is the set of sequential states;
δ_{ext}: $Q \times X_M^b \rightarrow S$ is the external state transition function;
δ_{int}: $S \rightarrow S$ is the internal state transition function;
δ_{con}: $Q \times X_M^b \rightarrow S$ is the confluent transition function;
λ: $S \rightarrow Y_M^b$ is the output function;
ta : $S \rightarrow R_{0,\infty}^+$ is the time advance function; with
Q: = {(s, e) | $s \in S$, $0 \leqslant e \leqslant ta(s)$} the set of total states.

The semantics of the P-DEVS definition are as follows. At any given time, a basic model is in a state *s*. and in the absence of external events, it will remain in that state for a period of time as defined by **ta**(s). When an internal transition takes place, the system outputs the value λ(s), and changes to state $\delta_{int}(s)$. If one or more external events $E = \{x_1, ..., x_n / x \in X_M\}$ occurs before ta(s) expires, i.e., when the system is in the state (s, e) with $e \leqslant$ **ta**(s), the new state will be given by $\delta_{ext}(s, e, E)$. Suppose that an external and an internal transition collide, i.e., an external event E arrives when e = **ta**(s), the new system's state could either be given by $\delta_{ext}(\delta_{int}(s), e, E)$ or $\delta_{int}(\delta_{ext}(s, e, E))$. The modeler can define the most appropriate behavior with the δ_{con} function. As a result, the new system's state will be the one defined by $\delta_{con}(s, E)$.

A coupled model connects the basic models together in order to form a new model. This model can itself be employed as a component in a larger coupled model, thereby allowing the hierarchical construction of complex models. The coupled model is defined as:

CM = <X, Y, D, EIC, EOC, IC>
X: is the set of input ports and values,
Y: is the set of output ports and values,
D: is the set of the component names,
EIC (External input couplings) connects the input events of the coupled model itself to one or more of the input events of its components,
EOC (External output couplings) connects the output events of the components to the output events of the coupled model itself,
IC (Internal coupling) connects the output events of the components to the input events of other components.

2.2. DEVS-based approaches for RT systems development

Different authors have used DEVS for RT systems development. The RT- DEVS formalism [21] is an extension of DEVS for real-time systems simulation. An activity mapping function ψ associated with each state is used to work as a real-time scheduler. The regular time advance function only verifies the correctness of the activity mapping time constraints and compensates time discrepancy problems. The time bound of each activity is specified by a new function called ti.

In [22] RT-DEVS has been used with the addition of the concept of a driver for hardware interaction. The main function of the driver model is the translation of input and output events from the RT-DEVS environment to the hardware and vice-versa. The driver model is added to each atomic component in the DEVS model hierarchy, and the atomic components interact independently with the external environment. This approach adds a processing burden on each atomic component, as well as keeping the Root Coordinator (RC) unaware of the interactions between the atomic components and the environment.

Some research efforts tried to extend DEVS theory for real-time applications. Most of the works focused on real-time simulation solely, where the discontinuity between implementation artifacts and analysis, design, and modeling artifacts is a common deficiency of most of these methods. There exist a number of practical models, which tend to bridge this gap for

case-study applications. ***Furfaro et al. present modular design approach for RT and embedded systems using RT-DEVS in [23]. In [24] RT-DEVS is used for safety critical embedded application development and the challenges are discussed.

In [25], a DEVS-based RT system has been implemented on a "TINI" Chip, which has limited memory and processing ability. A set of well-defined DEVS interfaces made it possible to define a just-as-needed RT environment and run on the chip efficiently. Finally, a case study model has been successfully run on the chip.

In [9] the authors show how an M&S environment based on the DEVS formalism can support model continuity in the design of dynamic distributed real-time systems. The authors prove that the discontinuity between implementation artifacts and analysis, design, and modeling artifacts is a common deficiency of most design methods. The authors restrict model continuity to the models that implement the system's real-time control and dynamic reconfiguration, and emphasize model continuity during the entire process of software development, where the control models can be designed, analyzed, and tested by simulation methods, and then smoothly transitioned from simulation to distributed execution. The proposed methodology supports model continuity by making it possible to deploy and execute the control models (initially designed and tested by simulation) directly into the real target system.

3. DEVSRT formalism

A real-time (RT) simulation is, in fact, a model running in RT, where the simulation interacts with the environment or the target system. Therefore, to use the same simulation model as the eventual RT application, the simulator must be able to handle inputs from external environments (such as hardware peripherals, software modules, network devices, and human operators) in a timely fashion. Our idea is to achieve this by using discrete-event models based on the DEVS formalism, and afterwards to transfer the models into an embedded platform, where they would function as controller interacting with the hardware through formally-defined interfaces added to the simulator. The models can be thoroughly tested using simulation-based verification and are incrementally replaced with hardware components. This provides a Hardware-In-the-Loop Simulation (HILS) platform, where hardware and software can be designed and developed in parallel, allowing for observing un-modeled characteristics of the hardware/software designs.

Fig. 3.1 illustrates the design and development cycle for an RT software application using DEVSRT [3,26]. The following steps are performed (the numbers correspond to the labels in the figure).

Initially (1), we define a specification model of the System of Interest (SoI) using a formal model (using DEVSRT or alternative techniques translated to equivalent DEVS models). Once the DEVSRT specification model is complete, model-checking can be used for validation of the model properties [27] (2). The same models are then used to run DEVSRT simulations of the behavior of the different sub-models under specific loads (3). In brief, we first study system properties analytically, and complement the proofs using simulation, which can also be used for hardware/software co-design (and for training). The same DEVSRT specification model is used to derive test cases (4), which can be also used for the simulation studies.

Deriving test cases from both the model and from the simulation results allows us to check that the models conform to the requirements. The control model is refined (5) based on the results of the HILS, allowing for exploring un-modeled and hidden aspects of the external environment. The incremental replacement (6) of simulation models with hardware surrogates allows for hardware–software co-design. During the HILS (7), the model is interfaced with the hardware by using formal interfacing techniques proposed here. A real-time Executive (6) executes the models on the particular hardware (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.

At this point, those parts that are still un-verified in the formal and simulated environments are tested, increasing the confidence of the engineer into the implemented system (8). Any modifications require going back to the same model specifications, which ensure that we can provide a consistent set throughout the development. This software lifecycle is

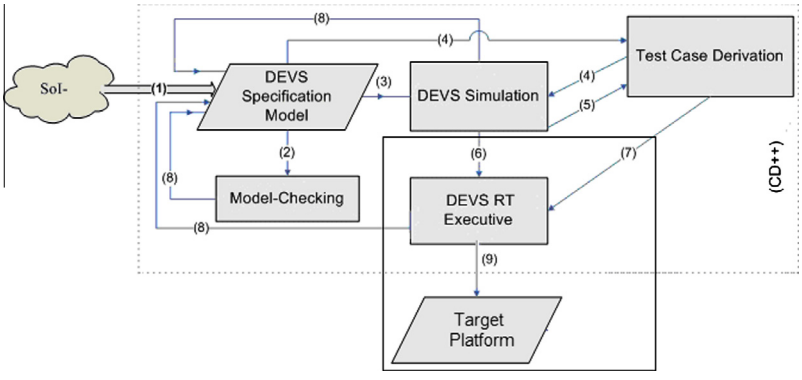

Fig. 3.1. DEVSRT development cycle (modified from [3]).

cyclic, allowing refinement following a spiral approach. On each cycle of the spiral, we end with a prototype application consisting of software/hardware components interacting with simulated components.

DEVSRT is proposed as a real-time DEVS approach [3,25] used to perform the activities described in the figure. Unlike RT-DEVS, this approach applies only minor modifications to the DEVS formalism, allowing for easy reuse of the previous models. As the most critical characteristic of a real-time system is the availability of outputs within the specified deadline, DEVSRT assigns a deadline to each output in an atomic component, and it verifies the deadline when the associated output is produced. Hence, the concept of deadline is embedded in the formalism and it is implemented in the abstract simulation mechanism.

In DEVSRT, instead of defining an activity mapping function (as opposed to RT-DEVS), the outputs of the atomic components are directly reflected to the hardware to echo the behavior of the model on the embedded device. The output function is responsible for triggering an action on the actuator on each internal event (triggered by the time-advance function) providing output to the hardware directly.

The atomic component of DEVSRT is formally defined by:

$AMRT = < X, S, Y, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta, d >$, where:

$X, S, Y, \delta_{ext}, \delta_{int}, \delta_{con}$ and λ are the same as P-DEVS [28].
$ta : S \rightarrow R^{+}_{0,\infty}$, time advance function (which can work with a discrete-event virtual clock or a RT physical clock of the system)
$d : S \rightarrow R^{+}_{0,\infty}$, is the relative deadline of each state for output production. The deadline starts at the end of the associated state when the output function is invoked to produce an output (that is considered the release time of the output task). The deadline is allocated to each output generated by the output function. Management of the deadline is done by the time-advance function.

To show the proof of closure under coupling of the DEVSRT formalism, it is necessary to demonstrate that a DEVSRT coupled model ($CMRT = < X, Y, D, \{M_i \mid i \in D\}, \{I_i\}, \{Z_{i,j}\}>$) is equivalent to a DEVSRT atomic model ($AMRT = < X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta, d>$). DEVSRT inherited all its specifications from P-DEVS, except the deadline (d) function. Thus, the associated atomic model derived from a coupled model will have the following specifications:

$S = \times Q_i$ where $i \in D$; (D the set of components).

$ta(s) = \text{minimum}\{\sigma_i | i \in D\}$, where $s \in S$ and $\sigma_i = ta(s_i) - e_i$; (e_i is the elapsed time of the current state of the *i*th component).
$d(s) = d(s_i)$, $|i \in D$ and i is the component where $ta(s) = ta(s_i)$.
The rest of the steps for δ_{ext}, δ_{int}, δ_{con}, and λ functions are the same as P-DEVS [28].

In other words, the ta(s) (life time of state *s*) of the coupled model is equal to the closest $ta(s_i)$ of its components ($_i$ is the component that its current state is ending the soonest among all the components). Therefore, the deadline of the output of the coupled model's current state *s* is equal to the deadline of this component.

DEVSRT maintains consistency with the DEVS formalism, allowing reusing DEVS models for RT and embedded system modeling. The coupled model definition in DEVSRT is the same as P-DEVS.

3.1. Real-time interface

In order to make a virtual-time simulator to work in real-time, the logical time representation of the simulator must be tied to the underlying computing system. In a DEVS-based system, the simulation time advances only when there is an event waiting to be serviced (virtual-time), however, in DEVSRT, the time-advance is tied to the clock of the underlying system. In a DEVS simulator, the Root coordinator (RC) is responsible for advancing the simulation time, initiating the events at the specified time and forwarding inputs to the components and outputs to the external environment [6]. In our case, RC only verifies the timing of the events and initiates the simulation cycles based on the wall clock timing scheme. In DEVSRT, the RC does not advance the time, instead it waits for the physical scheduled time of the next event to reach, then it triggers the event by sending the appropriate simulation message.

In order to use a DEVS model as the final target software architecture, DEVSRT employs model outputs as hardware control signals, and defines an interfacing mechanism between the model and the environment. A driver interfacing approach was presented in [22], which is now integrated with the proposed DEVSRT in a more efficient way by removing the extra processing burden from the atomic components. In this approach, the standard DEVS I/O ports of the top-most coupled component interact with the environment. They now include a driver object, working as an interface between the model and the external environment. This way, the model hierarchy remains unchanged and only the driver interfaces are added to the borders of the model. The driver objects are abstract functions that could be

overridden by the model developer, who can independently adapt them for different platforms, providing portability. The other advantage of this approach is making the RC aware of the atomic component interactions, conforming to the DEVS abstract simulator definition by Zeigler et al. [6].

The DEVSRT notation of the Top coupled model in the model hierarchy is modified as follows:

TOPCM = <X, Y, OS, IS, DX, DY, D, {M_d | d ∈ D}, EIC, EOC, IC>, where:
X, Y, D, M_d, EIC, EOC and IC are the same as DEVS
IS = {is | *is* is the input signals from environment} is the set of environment input signals.
OS = {os | *os* is the output signal to environment} is the set of hardware output signals.
DX: IS →Xv: converts external environment input signals to input port value (Xv).
DY: Yv →OS: converts output port value to external environment output signals (OS).

Any interaction between the environment and atomic component is routed through the formal interconnections from the Top coupled component to the atomic component or vice-versa. The interfacing mechanism allows for Hardware-In-the-Loop and Human-In-the-Loop simulation by connecting DEVS components with the hardware or human peripherals. The integration with hardware can be done incrementally, by replacing each model component with the corresponding hardware counterpart (e.g. sensor, actuator...) and providing the driver functions for the model ports previously connected to that model.

Model continuity is ensured, since the original model is finally deployed on the hardware, acting as the embedded software. The benefit of this approach versus code generation approaches is the limitless use of the simulation model features on the hardware and no extra verification of the generated code as the same model and source code are deployed on the hardware. The only components added to the model are the driver interface functions that are gradually verified in the incremental integration steps.

The algorithm of the RC main loop in DEVSRT is as follows:

```
1.  main():
2.  forever for each DEVSRT model /* main loop */
3.    wait for is signals from environment or internal time out
4.    if an external event then
5.      q = DX(is)
6.      send (q, t) msg
7.      send (*, t) msg
8.    else if an internal time out then
9.      send (@, t) msg
10.     send (*, t) msg
11.   else if receive (y, t)
12.     os = DY(y)
13.     send os signal to the hardware
14.   else if receive (done, t)
15.     t_N = t
16.   end if
17. end forever
```

Lines 5 and 12 show the driver object functions; which convert the input and output signal, respectively. The cycle starts with the RC waiting for inputs from the hardware or an internal event to occur (line 3). As soon as an input is received, it is converted into a DEVS predefined input value by the input driver function. Afterwards, an external message is sent to the target atomic component (based on the P-DEVS simulation mechanism [28] an external message is always accompanied by an internal message). If an output message is received, the DEVS output value is converted to a DEVS signal via the DY driver interface function. Note that q, *, and @ represent *input*, *internal*, and *collect* messages, respectively. Input message carries the input (value, time) pair from the RC to the atomic component that it is supposed to be received by. Upon receiving this message, the external function is invoked in the atomic component object. Internal message carries the internal event indication from the root coordinator to the specified atomic component, which invokes the internal function on the atomic component object. Collect message notifies the atomic component regarding producing an output, which invokes the output function on the atomic component object. t_N indicates the next change time in the system.

The external and internal message handling functions in the *Simulator* object (see Fig. 4.2) are the same as P-DEVS. The *Simulator* object is the processing engine of the atomic component, which carries out the transitions and output function. The following pseudo code represents the *collect* message handling function in a DESVRT Simulator object:

```
1.    when receive (@, t):
2.    if (t = t_N) then
3.      y = λ(s)
4.      if (t_now ⩽ t_L + ta(s)+ d(s))
5.        send (y, t) to the parent coordinator
6.      else
7.        error //deadline missed
8.      end if
9.      send (done, t) to the parent coordinator
10.   else if
11.     error
12.   end if
13.   end when
```

The simulator is responsible to verify the timing of the output. Thus, in line 4 the deadline function d(s) associated to the current state is called to verify whether the output is produced on time. The d(s) function returns the relative deadline of the output from the end of the current state s, thus it is added with (t_L + ta(s)) that indicates the end of the current state (t_{now} is the current simulation time and t_L is the time of the last event in the system). If the deadline is missed, an error signal is raised, informing the system about a late deadline, thus, the system can decide which action to pursue. The modeler can override a custom method to handle the missed deadline issue.

4. Implementation of DEVSRT on CD++

The DEVS formalism proposes a framework for model construction and defines an abstract simulation mechanism that is independent of the model itself. This mechanism provides a high-level implementation detail for the DEVS framework, and it can be feasibly implemented by computer software.

E-CD++ [14] is a RT implementation, based on the CD++ simulator [29] (a DEVS-based framework for M&S), and RT-CD++ [30] (an extension of CD++ for real-time simulation). E-CD++ supports modeling real-time systems by converting the CD++ virtual time-advance function to real-time, and it provides an RT simulation platform for verification of such models. It also supports the FDS-DEVS framework [31], where model components can change dynamically during the simulation. During this research, the proposed DEVSRT M&S framework is implemented on the E-CD++ software by modifying its simulation engine to execute real-time models more precisely and interacting with environment, based on the driver interfaces proposed earlier. To allow for direct replacement of models with external entities, the I/O ports of E-CD++ models implement the formal interfacing mechanism of DEVSRT. The underlying middleware is replaced with a real-time kernel and the runtime objects are imported to this platform as RT tasks. To follow the development cycle proposed in the previous section, the model development interfaces are also upgraded and several embedded functionalities are added.

Fig. 4.1 illustrates the E-CD++ development framework with DEVSRT modeling implemented during this research. The embedded platform with the external environment is shown in this layered approach, representing the cross-platform development of models. The E-CD++ execution engine uses the Xenomai real-time kernel [33] with multi-tasking services to implement DEVSRT. The user models and the driver objects are merged with the E-CD++ core objects, and the entire combination is compiled to produce an executable. Xenomai provides an RT kernel resting between the hardware and Linux OS, and offers several pervasive hard RT services to user space applications and is seamlessly integrated with GNU/Linux environment.

The use of Xenomai provides RT task scheduling and inter-task synchronization to the runtime engine, allowing for more efficient use of the underlying hardware platform. As discussed in Fig. 3.1, we check the schedulability of the models using a transformation into timed automata and model-checking [41]. This platform is also integrated with Imprecise Computations theory [42]. The proposed I-DEVS (imprecise DEVS) formalism uses a dynamic scheduling algorithm based on the criticality of the RT tasks to manage overload situations in the system by degrading the system's output accuracy in order to meet hard deadlines. Finally, the Xenomai kernel provides low level RT task scheduling and inter-task synchronization to the runtime engine. Xenomai applies a fixed priority-based preemptive scheduling, while also incorporating round-robin scheduling among same priority tasks. We use equal priority for all input tasks as the default configuration; however, specific priorities can be assigned to the input tasks in order to control the priority of the simultaneous inputs. The open-source nature of the tool provides flexibility in choosing different scheduling algorithms for input tasks and priorities associated with the tasks. Generally, the default functionality is to ignore the outputs that missed their associated deadlines; however, the modeler can override a function to handle the missed deadline for each output.

In order to improve and speedup model development, E-CD++ incorporates Eclipse programming environment with user-friendly interfaces suitable for real-time and embedded execution. The Generic Graphical Advanced environment for DEVS modeling and simulation (GGAD) graphical user interface based on DEVS-Graph [32] which was in the definition also integrated E-CD++ Eclipse IDE. This tool allows for graph-based DEVS model hierarchy, interconnections, and behavior

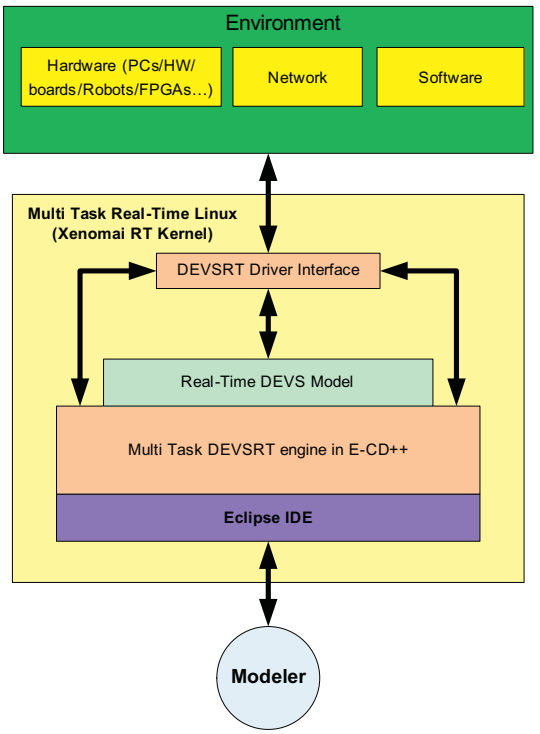

Fig. 4.1. E-CD++ with DEVSRT Development Framework.

representation to automate the model generation. Since E-CD++ executable is to be deployed on a different platform (embedded hardware), cross-compilation for the project is also provided, as well as means of communication to the target platform in order to download executable binary files, running the executable, and debugging remotely.

E-CD++ inherited the main object entities of CD++, applying the proposed DEVSRT approach by modifying object behaviors or adding new entities to the software architecture of CD++. The four main components of E-CD++ are the Main Runtime System, the Modeling Subsystem, the Runtime Subsystem and the Messaging Subsystem (Fig. 4.2).

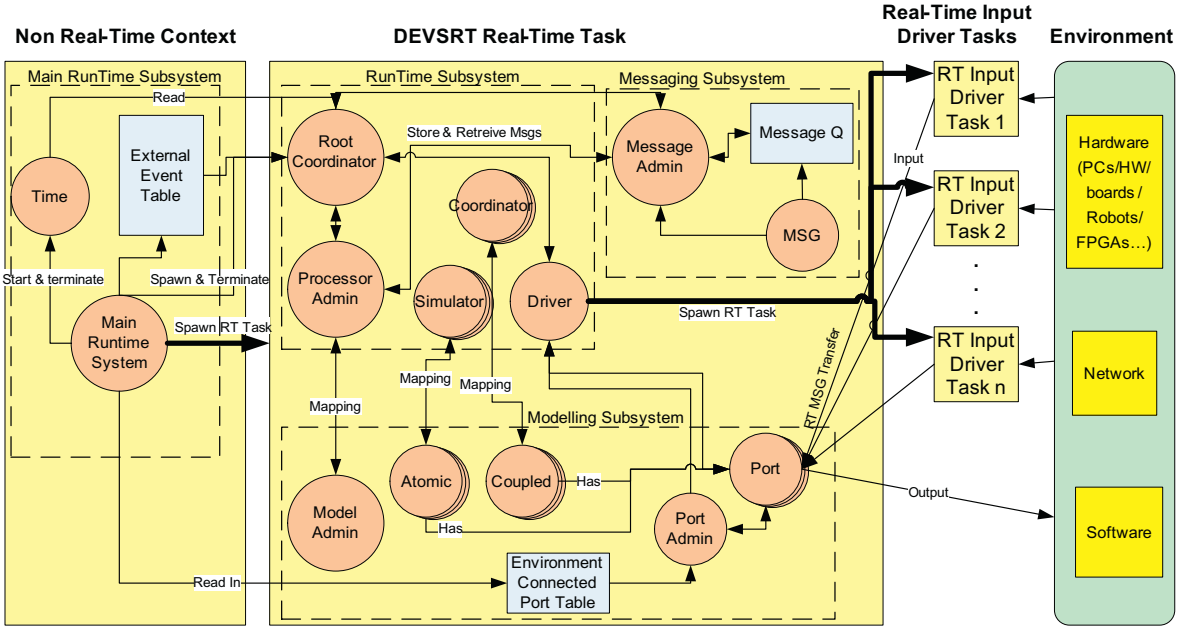

Fig. 4.2. E-CD++ software structure.

The Main Runtime System manages the overall aspects of the real-time execution and provides timing functions with microsecond precision. This is done by incorporating Xenomai native clock functions [33] in the E-CD++ *Time* class, which is itself instantiated by the Main Runtime System class. The Main Runtime System is the first object that is created in non-real-time context, and it spawns the Runtime Subsystem as a Xenomai real-time task. In general, it performs the following tasks in sequence:

- Registers Atomic component objects.
- Registers the Top coupled component ports that are connected to the external environment.
- Reads in the external events (from an existing event-file) and builds an external event table.
- Reads in the model-file and builds the model hierarchy.
- Spawns the main real-time task in which the Root Coordinator (RC) is created to start the DEVSRT execution cycle.

The Runtime Subsystem consists of Simulators, Coordinators, and the Processor Admin. In E-CD++, the Simulators work on run-time engines that correspond to atomic components, and they perform the main job of executing the internal transition and output function after receiving the proper messages. The RC is a special Coordinator that manages the real-time event scheduling. It initializes the global *Driver* object which spawns the real-time input driver tasks (which are associated with input ports of the Top coupled component in the DEVS model hierarchy) declared by the user.

4.1. Performance evaluation

In order to verify the efficiency of the implementation and to prove the performance gains of the multi-tasking approach on a real-time middleware, the proposed implementation is tested with synthetic models and compared with the previous RT-CD++ [30] implementation. The tests are performed using two different sets of DEVStone synthetic models [39] with different depth and width in the model hierarchy. The results of the tests are compared with the reported results of the previous evaluation of RT-CD++ published in [34,40].

In order to compare the two implementations in an equal and fair condition, the synthetic model proposed in [34] was duplicated in the new E-CD++ implementation. The model is composed of one coupled component and several atomic components in each level of the hierarchy. Fig. 4.3a illustrates the Top coupled component along with its inter-connections. The model can have multiple levels with the same architecture and several atomic components in each level. Fig. 4.3b shows the last level, which only has one atomic component.

Given a specified depth d and width w, we end up having k coupled components with $w - 1$ atomic components inside each model (except for the last coupled model, which only includes one atomic component). An input to this model propagates to each sub-component and is forwarded to the last level. This triggers the external transition function in each atomic component. All of the atomic components follow the same behavior, in which they are in a passive state, until an input is received. The external transition (invoked by the input) changes the state to a temporary state with zero time-advance, which produces an output, and then it transitions to a passive state, waiting for the next input. This cycle continues as long as there is an input to the system.

The goal is to measure the processing overhead incurred by the simulation engine with respect to the processing time incurred by the model. The overhead of executing a model is mainly associated with the message transfer scheme, the handling of input and message queues, and the time-advance management. The major processing in a DEVS model is performed in the output, external, and internal transition functions in the atomic component. Hence, to produce a computation extensive model, a mix of instructions during 50 ms is programmed in the external and internal transition functions of all of the atomic components in order to emulate a heavy processing situation (as defined by the DEVStone benchmark). To make the comparison platform-independent, the models are executed on the same workstation with the same computing power. The

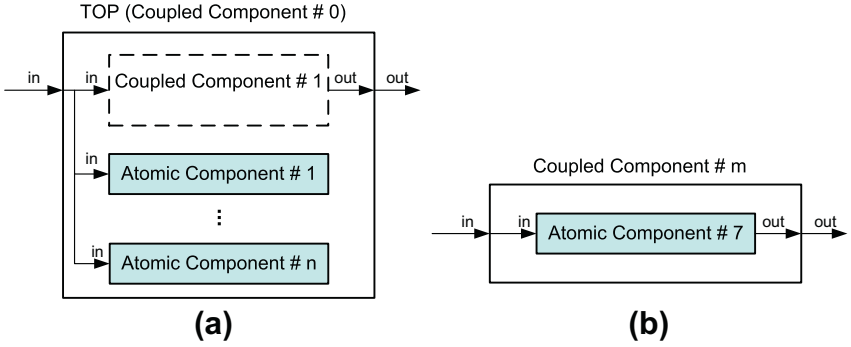

Fig. 4.3. Synthetic model architecture (modified from [34]).

percentage of overhead of the system relative to the model execution time is measured and compared. The percentage of software overhead is calculated using the following equation:

$$\text{Overhead } \% = \frac{\text{Total Processing Time} - \text{Total Transitions Processing Time}}{\text{Total Processing Time}} \times 100 \tag{4.1}$$

In the following figures, we show two sets of the varying tests we carried out. They show the overhead according to the changes in the number of components in each level and the depth of model hierarchy (number of levels). The first test uses four models with fixed number of components in each level (equal to 12) and variable depths of 3, 6, 9, and 12 levels for each model. A set of 10 inputs were injected to the models during a fixed execution time of 40 s. Fig. 4.4 represents the overhead percentage calculated using Eq. (4.1) for the above models in E-CD++ and compared with the available results of RT-CD++ in [34].

The second example presented here shows the results for the models of the same type with fixed number of levels (i.e., 4) and variable number of components per level (i.e., 4, 6, 8, 10, and 12). The same numbers of inputs are injected to the models and the models are executed for a period of 100 ms. Fig. 4.5 shows the results of this test.

As it can be seen from the above charts, the overhead percentage in E-CD++ is significantly lower than RT-CD++ in all scenarios (which was already low, with an average overhead below 4%). The use of a real-time middleware and a multi-tasking approach allowed us to improve this overhead approximately 10 times. The efficient tasks scheduling service offered by the Xenomai kernel speeds up the execution of the software. On the other hand, the chart demonstrates the efficiency of the simulation algorithm of DEVSRT, which does not add significant processing burden to the E-CD++ execution engine. Another observed feature of this implementation is the constant overhead over different sizes and architectures of models.

5. Case study

As a proof of concept, we show the application of the methodology and tools in a case study focused on the development of a robot controller. The model in this section will be used to demonstrate the model continuity, HILS, and hardware–software co-design contributions of the proposed methodology and tools. The controller model was developed using DEVSRT and implemented on E-CD++ to perform various tests and apply different features.

The software application is deployed on E-puck [35], a mobile robot with academic purpose. The E-puck has eight infrared distance proximity sensors to detect obstacles around it. There are eight LEDs mounted around the robot's body. It also has two motors connected to the two wheels on both sides, which make the robot capable of moving forward, backward, and spinning in any direction. Fig. 5.1a shows the e-puck robot and Fig. 5.1b illustrates the placement of IR sensors and LEDs on the robot.

In [37] we used the hardware to carry out an advanced interoperability experiment in which the controller for the E-puck was split between two different DEVS engines running concurrently. In that one, the E-puck was used to experiment on the interoperability of a continuous controller built in the Power_DEVS tool working together with a sequential controller built ECD++. The model introduced here does not use such a distributed configuration, (which was the main purpose of that research) and it uses an autonomous configuration. The model handles events during obstacle avoidance. The controller also scans the input ports for possible inputs during bypassing obstacles.

A DEVSRT controller model was designed to steer the robot in a field, while avoiding obstacles. The model contains an atomic component representing the behavior of the controller. The model uses 8 input ports (*InIR0*,...,*InIR7*) that receive periodic inputs from proximity sensors. The model also uses two output ports: *OutMotor* (transfers the output commands to the motors) and *OutLED* (transfers the LED on/off commands).

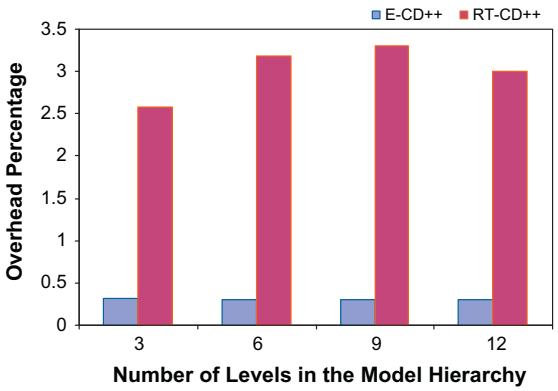

Fig. 4.4. DEVS percentage of overhead with variable depth.

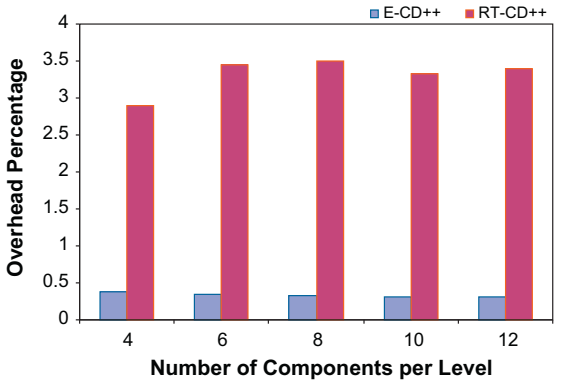

Fig. 4.5. Percentage of overhead with variable width.

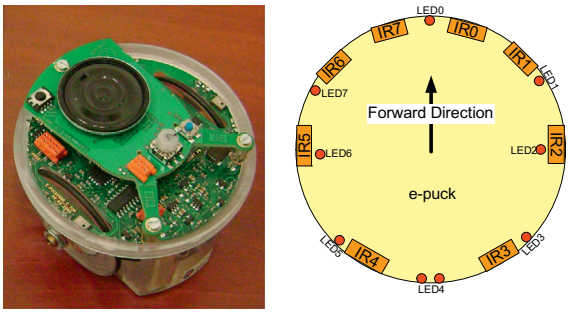

Fig. 5.1. (a) e-Puck robot and (b) placement of sensors and LEDs.

The controller executes the following 5 different actions based on the inputs received from the sensors: *move forward*, *turn 45° left*, *turn 45° right*, *turn 90° left*, *turn 90° right*, *turn 180° and stop*. Initially, the robot starts moving forward while receiving periodic inputs from proximity sensors and analyzes them. As soon as it detects an obstacle, the former performs one of the turning actions based on the position of the obstacle. The robot keeps turning until it finds an empty space on the front. The controller also uses LEDs to signal the action that is being performed. For example, if the robot is moving forward, the front LED (LED0) turns on and if it is turning 45° to left, LED7 turns on.

Table 5.1 lists the outputs of the DEVS model and their associated actions to be performed on the robot hardware. The driver interfaces transform numeric values to the command signals on the robot.

Fig. 5.2 illustrates the DEVS Graph representing the controller's behavior. The state diagram summarizes the behavior of a DEVS atomic component by presenting the states, transitions, inputs, outputs and state durations graphically [36]. The circles represent states and the double circle is the initial state. The name and duration of a state is shown in the circle. The continuous edges between the states represent the external transitions, which includes the names of the input ports, the input value and any condition on the input (with format "port?value"). The dotted lines represent the internal transitions and the associated outputs (with format "port!value").

Table 5.1
DEVS output mapping table.

Port name	Port value	Hardware command	Comment
OutLED	100	Turn all LEDs off	
	0,10,20,...,70	Turn LED off	The most significant digit is the number of Led to be turned off
	1,11,21,...,71	Turn LED on	The most significant digit is the number of Led to be turned on
OutMotor	0	Set horizontal and rotational speed to 0 m/s	Stop
	1	Set horizontal speed to 0.5 m/s	Move forward
	2	Set rotational speed to 1 r/s	Turn 45° left
	3	Set rotational speed to −1 r/s	Turn 45° right
	4	Set rotational speed to −1 r/s	Turn 90° right
	5	Set rotational speed to 1 r/s	Turn 90° left
	6	Set rotational speed to 1 r/s	Turn 180°

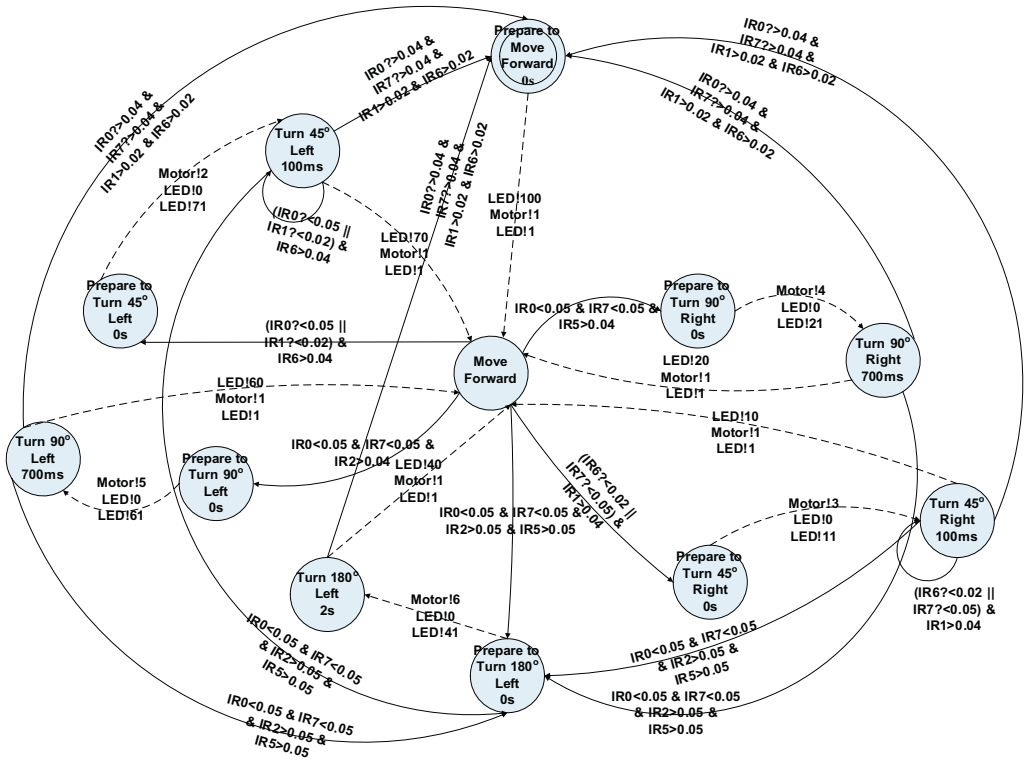

Fig. 5.2. Epuck atomic component state diagram.

The controller always watches for any obstacle in front of the robot by checking the values of sensors IR0, IR1, IR6 and IR7. Initially the robot moves forward and if there is no obstacle, it continues doing so. As soon as one of the IR sensors detects an obstacle, the controller verifies IR6 sensor, and if there are no obstacles, it performs a 45° turn to the left. Otherwise, it checks the IR1 value and if it is open, the robot turns 45° to the right. If both IR1 and IR6 are blocked, it looks at IR2 sensor and if there is an open space, the robot performs a 90° turn to the right. The same happens when IR2 shows an obstacle and IR5 an empty space, (the robot turn 90° to the left). If all of the sensors are blocked, the robot tries turning to the opposite direction (180°).

In this example, a period of 50 ms is defined for the IR sensor inputs, polling the values of the IR sensors and injecting them to the model, which in response invokes the e-puck atomic component's external transition function.

Below is the model file of the controller coupled-model.

```
components: epuck@Epuck
out: outmotor outled
in: inir0 inirl inir2 inir3 inir4 inir5 inir6 inir7
link: inir0 ir0@epuck
link: inirl irl@epuck
link: inir2 ir2@epuck
link: inir3 ir3@epuck
link: inir4 ir4@epuck
link: inir5 ir5@epuck
link: inir6 ir6@epuck
link: inir7 ir7@epuck
link: motor@epuck outmotor
link: led@epuck outled
inir0: 00:00:00:100
inirl: 00:00:00:100
inir2: 00:00:00:100
inir3: 00:00:00:100
```

(*continued on next page*)

```
18      inir4: 00:00:00:100
19      inir5: 00:00:00:100
20      inir6: 00:00:00:100
21      inir7: 00:00:00:100
22      [epuck]
23      preparationTime: 00:00:00:000
24      turn45Time: 00:00:00:100
25      turn90Time: 00:00:00:700
26      turn180Time: 00:00:02:000
```

Line 1 declares the name of the E-puck DEVS components inside the Top. Lines 2 and 3 declare the output and input ports of the model, respectively. Lines 4–13 define the interconnections between the ports and lines 14–21 declare the period of inputs for the IR sensor driver tasks. Line 22 starts the declaration of *epuck* atomic component, and lines 23–26 declare the duration of states within the *epuck* atomic component.

As a first experiment, a random environment was modeled and the controller model behavior was observed. The model was first tested using virtual-time simulation mode, in which we added a distance generator model, which produces random IR sensor values and inputs them to the controller model. The controller model reacts to the combination of values every 1 s, generated by this model. Fig. 5.3 is the Atomic Animation diagram generated by CD++Modeler, which shows the input and output trajectory in a specified period. The port names and scales are shown on the left panel, and the values sent or received in the ports and the event times are shown on the right panel. As we can see, at time 0 "Move Forward" output is produced by the "outmotor" port, and value of 100 is produced at the "outled" port, causing the LED1 to turn on (signaling the forward

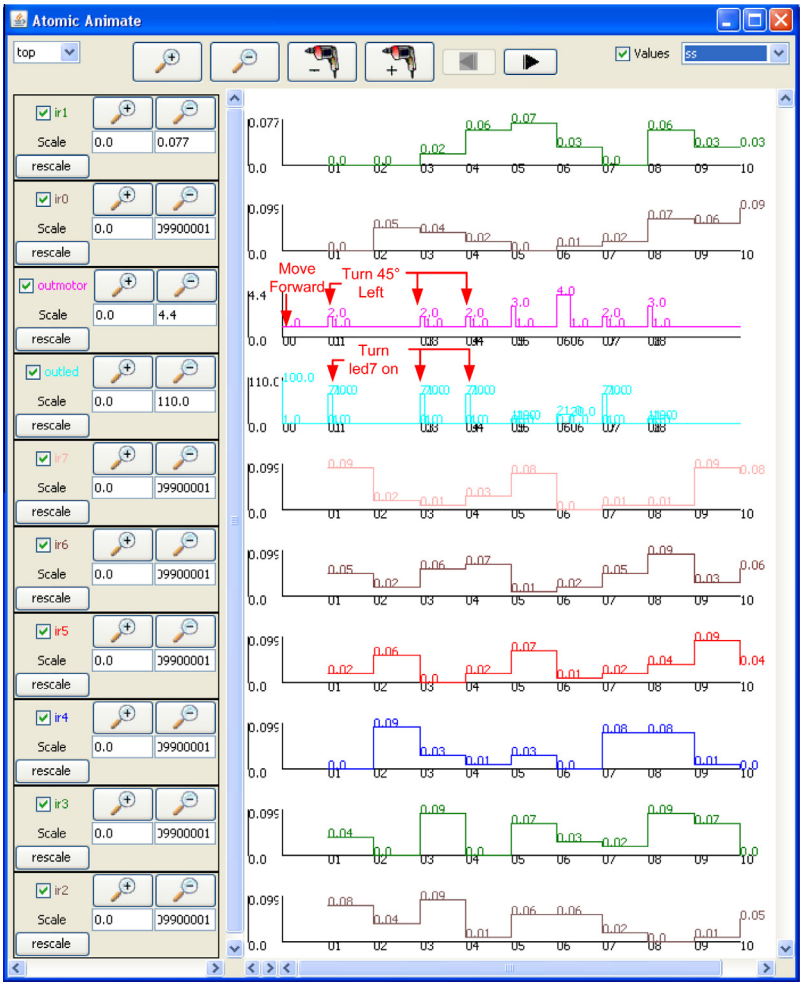

Fig. 5.3. Atomic animation diagram for e-puck random distance test.

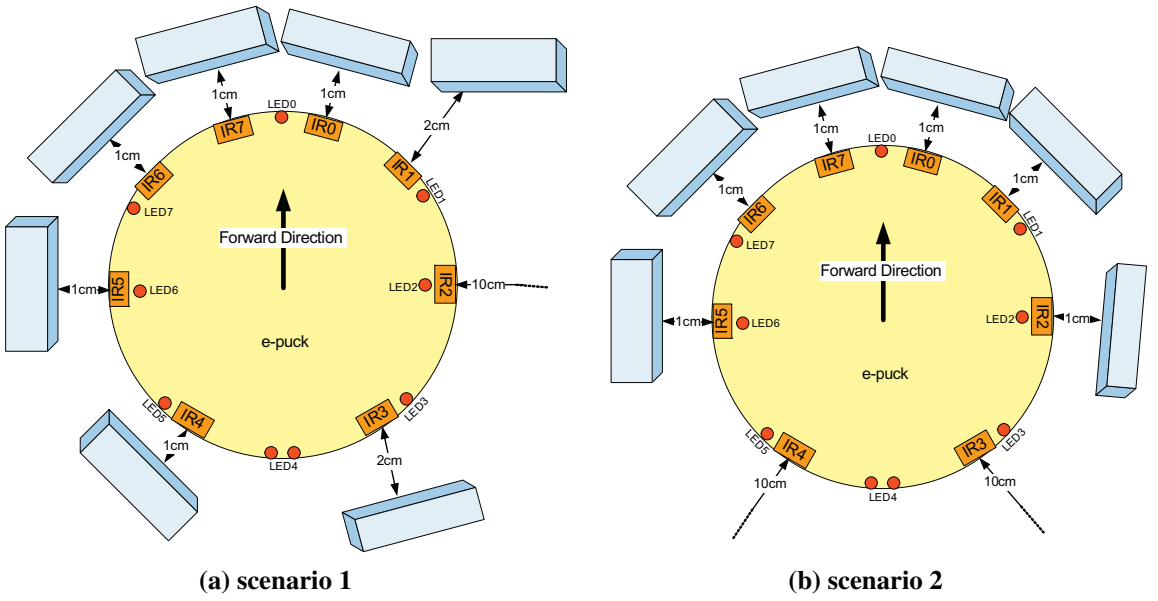

Fig. 5.4. Event-file scenarios.

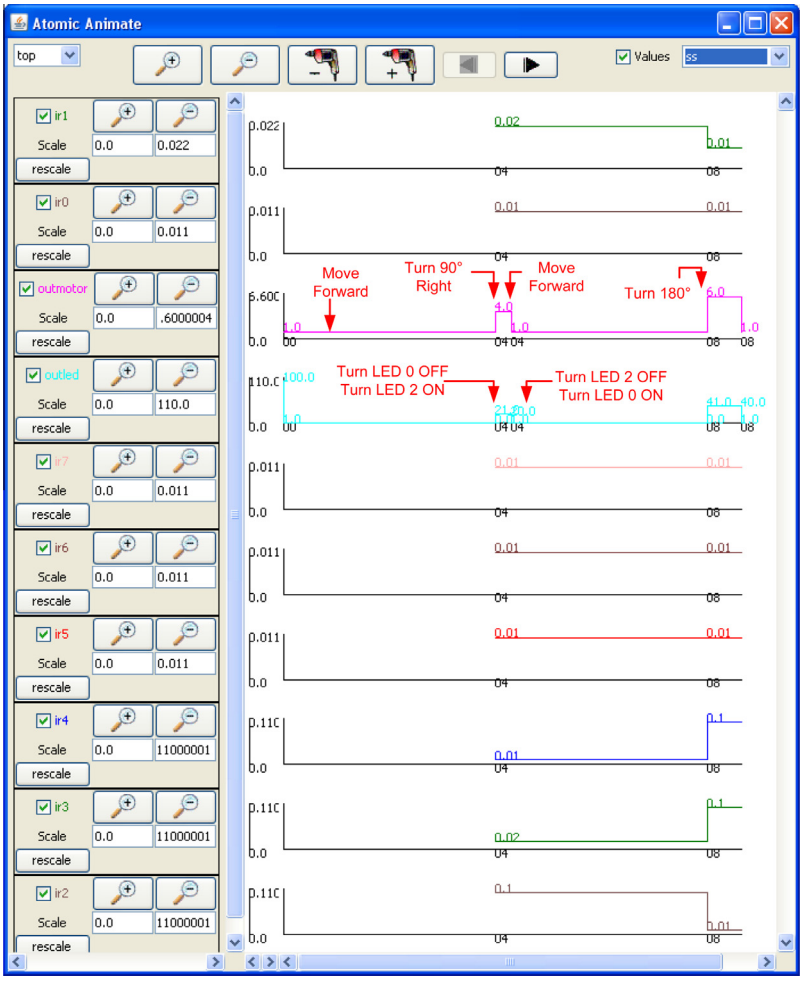

Fig. 5.5. Atomic animation diagram for e-puck random distance tests after applying the obstacle scenarios.

moving action). After 1 s "ir0" receives a value of 0 indicating that there is an obstacle on the right corner; thus, the robot must turn to the left side, producing the value of 2 indicating a 45° turn to left. The rest of the diagram can be traced by following the inputs from IR sensors and the associated outputs produced in response to the inputs from "outled" and "outmotor" ports, on the timelines.

We also show two scenarios designed by generating obstacles using events in the event-file. Fig. 5.4 illustrates the two sample scenarios in which obstacles block the robot's path.

Fig. 5.5 shows the I/O of the model using the CD++Builder. The two scenarios shown in Fig. 5.4 are injected to the model at times 4 and 8 s, respectively. The outputs of the "outmotor" and "outled" ports are shown in the third and fourth row and the associated commands are indicated in the figure. The robot starts with moving forward (by producing value 1 from port "outmotor"). At time 4 s, the first set of inputs (Fig. 5.4a) are injected to the system (shown in the "ir0" to "ir7" rows at time 4 s), forcing the robot to react by turning 90° right (output value of 4 from "outmotor" port) and turning led2 on (output value of 21 from "outled" port). The second input set (Fig. 5.4b) is injected at time 8 s, triggering a 180° turn (indicated in Fig. 5.5 in the "outmotor" row with output value of 6) and turning the rear led (led4) on (the value of 41 is produced from "outled" port).

After verifying the model behavior in various scenarios like the ones discussed above, the model was deployed in a real robot, where it was executed in real-time mode in which, the driver interfaces were activated and performed the transformation of I/O. The same behavior was observed and the robot could find its way through the obstacles.[1]

6. Conclusions

M&S techniques offer significant support for the design and verification of complex embedded real-time applications. DEVS provides a sound methodology for developing discrete-event applications, which can be easily applied to improve the development of real-time applications. These advantages include risk-free and reliable testing, hybrid modeling and knowledge sharing from other modeling formalisms, model reuse, and the possibility of analyzing different levels of abstraction in the system.

In this paper, DEVS formalism has been extended for developing model-based embedded systems by introducing the DEVSRT formalism. The formal and intrinsic advantages of DEVS are combined with RT features to propose a design scheme for such applications. Issues such as Hardware-In-the-Loop Simulation (HILS) or Human-In-the-Loop Simulation are addressed in this framework by introducing formal interfacing mechanisms between the DEVS model and the target embedded environment. The benefits of simulation-based verification are employed by DEVSRT, allowing for pervasive verification of the system under development in a risk-free setting, exploring varying test scenarios. The concept of model continuity and reuse of simulation models in the development of final embedded software architecture are addressed (the shortcoming of the available approaches). DEVSRT provides a high-level abstract hardware-software modeling scheme, where different components of the target system can be modeled together. The co-modeling approach allows for co-simulation and verification of hardware and software segments of an embedded system in a unified framework, while an incremental replacement of the models with hardware surrogates explores the un-modeled aspects of the devices with the controller component.

A case study of a robot controller has been proposed in which the cyclic development of the controller software from the simulation model has been demonstrated, as well as the testing under virtual environment and real conditions. Expanding the case study can be a good idea and also comparing our proposed DEVSRT with other approaches such as MATLAB/Simulink is also considered. We have already considered two different scenarios of obstacles and showed the details of DEVSRT model, state diagram, implementation, and the results on ECD++ software. We already published two papers showing the development of collaborative models using DEVSRT on ECD++ and RTDEVS on PowerDEVS [37] and another one using DEVSRT on ECD++ and Modelica [38]. These two publications present a comparative development approach in DEVSRT with PowerDEVS and Modelica that can be considered as a comparison between our approach with other RT approaches.

References

[1] J.W.S. Liu, Real-Time Systems, Prentice-Hall, Upper Saddle River, NJ, 2000.
[2] Steve Heath, Embedded Systems Design, Newnes, 2002.
[3] Gabriel A. Wainer, Discrete-event modeling and simulation; a practitioner's approach, CRC/Taylor & Francis, 2009. ISBN: 9781420053364.
[4] Gabriela Nicolescu, Pieter J. Mosterman, Model-based design for embedded systems, CRC Press, 2010. ISBN: 978-1-4200-6784-2.
[5] Jean François Monin, Michael Gerard Hinchey, Understanding Formal Methods, Springer, 2003. ISBN: 1852332476.
[6] B. Zeigler, T. Kim, H. Praehofer, Theory of Modeling and Simulation, Academic Press, 2000. ISBN-10: 0127784551.
[7] Hae Sang Song, Tag Gon Kim, Application of real-time DEVS to analysis of safety-critical embedded control systems: railroad crossing control example, Simulation 81 (2) (2005) 119–136.
[8] H. Saadawi, G. Wainer, Verification of real-time DEVS models, in: Proceedings of DEVS Symposium 2009, San Diego, CA, 2009.
[9] X. Hu, B.P. Zeigler, Model continuity in the design of dynamic distributed real-time systems, IEEE Transactions on Systems, Man and Cybernetics, Part A 35 (6) (2005) 867–878.
[10] The MathWorks website. <http://www.mathworks.com> (visited May 2011).
[11] LabVIEW Website. <http://www.ni.com/labview/> (accessed June 2011).
[12] B. Selic, The emerging real-time standard [UML], in: Proc. 6th Int. Workshop Object-Oriented Real-Time Dependable Systems, Rome, Italy, 2001, pp. 3–9.

[13] D. Huang, H. Sarjoughian, Software and simulation modeling for real-time software-intensive systems, in: Proceedings of 8th IEEE Symp. on Distributed Simulation and Real-time Applications, 2004, pp. 196–203.
[14] Henry Yu, Gabriel A. Wainer, eCD++: an engine for executing DEVS models in embedded platforms, in: Proceedings of the 2007 SCS Summer Computer Simulation Conference, San Diego, CA, 2007.
[15] A. Basu, M. Bozga, J. Sifakis, Modeling heterogeneous real-time components in BIP, Software Engineering and Formal Methods (2006) 3–12. SEFM 2006.
[16] K. Balasubramanian, A. Gokhale, G. Karsai, et al, Developing applications using model-driven design environments, Computer 39 (2) (2006) 33–40. ISSN: 0018-9162.
[17] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, Y. Xiong, Taming heterogeneity the Ptolemy approach, Proceedings of the IEEE Transaction 91 (2003) 127–144.
[18] OpenSystemCInitiative website, Core SystemC Language and Examples, Release 2.2. <http://www.systemc.org/downloads/standards/> (accessed January 2011).
[19] F. Boussinot, R. de Simone, The ESTEREL language, Proceedings of the IEEE Software Journal 79 (9) (1991) 1293–1304.
[20] Felice Balarin et al, Hardware–Software Co-design of Embedded Systems. The POLIS Approach, Kluwer Academic Publishers, 1997.
[21] J.S. Hong, H.H. Song, T.G. Kim, K.H. Park, A Real-Time Discrete Event System Specification Formalism for Seamless Real-Time Software Development, Springer, Netherlands, 1997.
[22] S.M. Cho, T.G. Kim, Real-time DEVS simulation: concurrent, time-selective execution of combined RT-DEVS model and interactive environment, in: Proceeding of 1998 Summer Simulation Conference, Reno, Nevada.
[23] A. Furfaro, L. Nigro, A development methodology for embedded systems based on RT-DEVS, Innovations in Systems and Software Engineering 5 (2) (2009) 117–127.
[24] H. Song, T. Kim, Application of RT-DEVS to analysis of safety critical embedded control system: railroad crossing example, Simulation 81 (2) (2005) 119–136.
[25] X. Hu, B.P. Zeigler, J. Couretas, Devs-On-A-Chip: implementing DEVS in embedded java on a tiny internet interface for scalable factory automation, in: Proceedings of the 2001 IEEE Systems, Man, and Cybernetics Conference, pp. 3051–3056.
[26] Gabriel Wainer, Rodrigo 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, 2011. April.
[27] H. Saadawi, G. Wainer, M. Moallemi, Principles of DEVS models verification for real-time embedded applications, in: Pieter Mosterman, Katalin Popovici (Eds.), Chapter in the Book "Real-Time Simulation Technologies: Principles, Methodologies, and Applications", CRC Press, 2011.
[28] A. Chow, D. Kim, B. Zeigler, Parallel DEVS: a parallel, hierarchical, modular modeling formalism, in: Proceedings of Winter Simulation Conference, Orlando, Florida, 1994.
[29] G. Wainer, CD++: a toolkit to define discrete-event models, Software, Practice and Experience, vol. 323(3), Wiley, 2002. pp. 1261–1306.
[30] G. Wainer, E. Glinsky, Model-based development of embedded systems with RT-CD++", in: Proceedings of the WIP Session, IEEE Real-Time and Embedded Technology and Applications Symposium, Toronto, ON, Canada, 2004.
[31] H. Shang, G.A. Wainer, A flexible dynamic structure DEVS algorithm towards embedded systems, in: Proceedings of the Summer Computer Simulation Conference, San Diego, California, 2007, pp. 339–345.
[32] H. Praehofer, D. Pree, Visual modeling of DEVS-based multiformalism systems based on higraphs, in: Proceedings of the Winter Simulation Conference, Los Angeles, CA, 1993, pp. 595–603.
[33] Xenomai Real-Time Framework for Linux. <www.xenomai.org> (visited May 2011).
[34] E. Glinsky, G. Wainer, Performance analysis of real-time DEVS models, in: Proceedings of the Winter Simulation Conference, San Diego, CA, 2002, pp. 588–594.
[35] F. Mondada, M. Bonani, X. Raemy, J. Pugh, C. Cianci, A. Klaptocz, S. Magnenat, J.-C. Zufferey, D. Floreano, A. Martinoli, The e-puck* a robot designed for education in engineering, in: Proceedings of the 9th Conference on Autonomous Robot Systems and Competitions, Castelo Branco, Portugal, 2009, pp. 59–65.
[36] G. Christen, A. Dobniewski, G. Wainer, Modeling state-based DEVS models in CD++, in: Proceedings of MGA, Advanced Simulation Technologies Conference 2004 (ASTC'04), Arlington, VA, USA.
[37] Mohammad Moallemi, Gabriel Wainer, Federico Bergero, Rodrigo Castro, Component-oriented interoperation of real-time DEVS engines, in: Proceedings of the 44th Annual Simulation Symposium (ANSS '11). Society for Computer Simulation International, San Diego, CA, USA, 2011, pp. 127–134.
[38] Carla Martin Villalba, Alfonso Urquia, Mohammad Moallemi, Gabriel Wainer, DEVS-GRAPH in modelica for real-time simulation, In: Proceedings of European Conference on Modeling and Simulation, ECMS, Koblenz, Germany, 2012.
[39] Ezequiel Glinsky, Gabriel A. Wainer, DEVSTONE: a benchmarking technique for studying performance of DEVS modeling and simulation environments", in: Proceedings of IEEE DS-RT, Montréal, QC, 2005.
[40] Gabriel A. Wainer, Ezequiel Glinsky, Marcelo Gutierrez-Alcaraz, Studying performance of DEVS modeling and simulation environments using the DEVStone benchmark, SIMULATION: Transactions of the Society for Modeling and Simulation International 87 (7) (2011) 555–580.
[41] Hesham Saadawi, Gabriel A. Wainer, Principles of DEVS models verification, SIMULATION: Transactions of the Society for Modeling and Simulation International (2011).
[42] Mohammad Moallemi, Gabriel Wainer, I-DEVS: imprecise real-time and embedded DEVS modeling, in: Proceedings of Spring Simulation Conference, DEVS Symposium, Boston, USA, 2011 (Best Paper).