RPDEVS Abstract Simulator

Franz J. Preyser*, Bernhard Heinzl, Wolfgang Kastner

Institute of Computer Engineering, Automation Systems Group, TU Wien, Treitlstraße 1-3, 1040 Vienna, Austria; *franz.preyser@tuwien.ac.at

SNE 29(2), 2019, 79-84, DOI: 10.11128/sne.29.tn.10473
Received: March 30, 2019 (Selected ASIM GMMS/STS 2019 Conference Publication); Accepted; May 10, 2019
SNE - Simulation Notes Europe, ARGESIM Publisher Vienna, ISSN Print 2305-9974, Online 2306-0271, www.sne-journal.org

Abstract. The Revised Parallel DEVS (RPDEVS) modeling formalism enhances the Parallel Discrete Event System Specification (PDEVS) by the ability to model 'real' Mealy behavior of components. The term 'real' Mealy behavior can be summarized as immediate output response to an input event without a state transition in between. Although this enhancement simplifies model creation, especially of reusable components, it requires a more complex simulation algorithm. In this paper, we present an RPDEVS abstract simulator that describes the simulation execution of RPDEVS models.

Introduction

The Discrete Event System Specification (DEVS) [1] is a modular and hierarchical modeling formalism for systems that process input events, have an internal state, and may produce output events. Basic components can be specified as atomic DEVS which can be coupled with one other in a block diagram manner. The formal definition of an atomic DEVS is similar to a finite automaton (or sequential machine). In [2], the author describes an atomic DEVS as *DEVS Moore Automaton* embedded in additional logic that provides the necessary time events. Automata theory distinguishes between Moore and Mealy automata. The output events of Moore automata solely depend on the system's current state, whereas the output of Mealy automata may also depend on the current input. In theory, these two types of automata are equivalent in the sense that every automaton of the one type can be replaced by a corresponding automaton of the other type. However, in average the Moore model needs about twice the number of states and transitions than the corresponding Mealy model to represent the same system [3].

Both, in classic DEVS and in its most popular revision PDEVS [4], the output function λ solely depends on the internal state of the system. Thus, these two formalisms only allow the modeling of Moore behavior. If Mealy behavior is needed, it has to be modeled with a workaround, using a *transitory state* (a state with zero lifetime). However, as discussed in [5], the use of transitory states leads to a delay of events regarding processing order, which in turn impedes reusability of components. Due to the reasons mentioned above and the experiences we made with applying both, DEVS [6] and PDEVS [7], we decided to revise PDEVS resulting in RPDEVS published in [8]. Basically, the changes include the support of 'true' mealy behavior and the merging of the three state transition functions δ_{int}, δ_{ext}, and δ_{conf} into one generic state transition function δ. As mentioned above, a Mealy automaton needs about half the states compared to the corresponding Moore automaton. Evaluation of RPDEVS shows that formalization of Mealy models simplifies to a similar extent compared to PDEVS. Also merging the state transition functions condenses model definition, since the different transition functions often match at least in parts. However, the price for simplifying modeling is an increase in the complexity of the simulation algorithm.

In this work, we first recap the RPDEVS formalism, before its simulation algorithm is described and presented as *abstract simulator*.

1 RPDEVS Formalism

Equally to classic DEVS and PDEVS, RPDEVS distinguishes between *atomic* and *coupled* components which can be used for modular and hierarchical structuring of complex models (see Figure 1). As shown in [8], RPDEVS also provides *closure under coupling*, which means that for every coupled component an equivalent atomic component can be designed. This assures that couplings can be used within other couplings as if they were atomics.

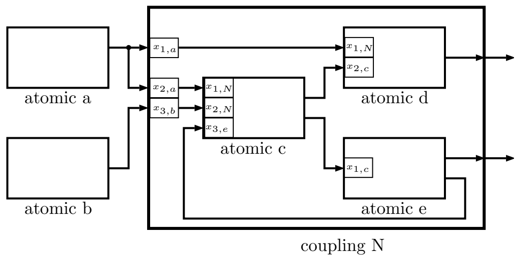

Figure 1: Modular and hierarchical decomposition of a complex model into atomic and coupled RPDEVS components.

1.1 Atomic RPDEVS

Formally, an atomic RPDEVS M is defined as

$$M = < X^b, S, Y^b, \delta, \lambda, ta >,$$

where the single entities have the following meanings:

$X^b \dots$ set of possible input bags
$S \dots$ set of possible states (=state space)
$Y^b \dots$ set of possible output bags
$\delta: \ Q \times X^b \rightarrow S \dots$ *state transition function*
$\lambda: Q \times X^b \rightarrow Y^b \dots$ *output function*
$ta: S \rightarrow [0, \infty] \dots$ *time advance function*
$Q = \{(s, e) | s \in S, e \in [0, ta(s)]\}$
$e \dots$ *elapsed time* since last event

Theoretically, X^b is a set of multisets with no particular structure. However, for practical implementation where it is feasible to define input ports which can be connected individually to output ports of other components, the set of possible input bags may be structured into sub-bags, one for each input port. Additionally, the sub-bags can be structured according to the source components the corresponding input messages origin from (see Figure 1). This is especially done in the RPDEVS simulation algorithm presented in Section 2, which has to remember the source component of every input message.

The differences of an atomic RPDEVS compared to PDEVS are the input dependency of the output function λ and the single state transition function δ which replaces the three separated transition functions δ_{int}, δ_{ext}, and δ_{conf} (for details about PDEVS, see [4, 1]). Furthermore, in RPDEVS, λ is called on any kind of event, external, internal, and confluent. The explicit distinction between these three event types is dropped and the behavior of an RPDEVS atomic is the same for each of them:

1. Call the output function λ.
2. Recalculate λ as long as the input bag changes due to (re)calculations of lambda at influencing components (*lambda-iteration*).
3. Conduct state transition δ once (*delta-step*).
4. Call the time advance function ta which returns the time to the next internal event.

If different treatment is necessary depending on whether the event was triggered by the arrival of an input (external event), by the expiration of the current state's lifetime (internal event), or by both happening concurrently (confluent event), this has to be incorporated into the definitions of δ and λ. External events can be recognized by a non-empty input bag ($x^b \neq \emptyset$), whereas internal events imply $e = ta(s)$.

The single transition function δ avoids having to define identical behavior multiple times in cases in which the three transition functions partly match.

According to [9], it frequently happens that calculations necessary for the output event in λ are also necessary for the computation of the next state and thus, have to be repeated in δ_{int}. In the classic DEVS simulator *DesignDEVS* [10], they even merge the two functions λ and δ_{int} to prevent unnecessary recalculations. This is not possible for RPDEVS as λ may have to be called multiple times before the state transition can be conducted. Therefore, for practical implementation, we recommend to split the internal state s of an atomic into two parts $s = (s_\delta, s_\lambda) \in S = S_\delta \times S_\lambda$ which allows to redefine λ and δ as follows:

$$\begin{aligned} \lambda: &\quad S_\delta \times [0, \infty) \times X^b \rightarrow Y^b \times S_\lambda, \ (s_\delta, e, x^b) \mapsto (y^b, s_\lambda) \\ \delta: &\quad S_\delta \times S_\lambda \times [0, \infty) \times X^b \rightarrow S_\delta, \ (s_\delta, s_\lambda, e, x^b) \mapsto s_\delta \end{aligned}$$

Thus, when λ already needs to calculate a new state value for generating the output y, it can be buffered into s_λ to be reused in δ.

1.2 Coupled RPDEVS

The formal definition of a coupled RPDEVS is identical to that of a coupled PDEVS (see [4]):

$$N = < X^b, Y^b, D, \{M_d\}_{d \in D}, \{I_d\}_{d \in D_N}, \{Z_{i,d}\}_{i,d \in D_N} >$$

with $D_N = D \cup \{N\}$ and

$X^b \dots$ set of possible input bags
$Y^b \dots$ set of possible output bags
$D \dots$ index set
$M_d \dots$ child component of N for each $d \in D$
$I_d \subset D \cup \{N\} \dots$ influencer set of d
$Z_{i,d} \dots$ output translation function

The output translation function $Z_{i,d}$ translates the output events of component i into input events for component d. Theoretically, $Z_{i,d}$ could also alter output events. However, in practice it just forwards events. If the destination component is a coupling, the output translation functions of that coupling further forwards the events to the destinations within the coupling. This is repeated until finally the events reach atomics.

As already mentioned in Section 1.1, the multiset of possible input bags can be structured by input port and source component. In the following, we will not consider ports, but separate the input bags according to the influencers the messages originate from. Such a structuring for a component d has the form

$$X_d^b = \prod_{i \in I_d} X_{i,d}^b,$$

with $X_{i,d}^b$ being the multiset of possible input messages from component i ($Z_{i,d} : Y_i^b \to X_{i,d}^b$). Consequently, every input bag x_d^b of a component d has the form

$$x_d^b = (x_{i_1,d}^b, x_{i_2,d}^b, \dots, x_{i_l,d}^b), \quad I_d = \{i_1, i_2, \dots, i_l\}.$$

Thereby, $x_{i_k,d}^b$ is the translated result of the output function of influencer i_k: $x_{i_k,d}^b = Z_{i_k,d}(y_{i_k}^b)$, $\forall k = 1, 2, \dots, l$.

2 RPDEVS Abstract Simulator

To complete the introduction of RPDEVS started in [8], the definition of an abstract simulator is given. Like in classic DEVS and parallel DEVS, the code consists of a *simulator* part responsible for executing an atomic, a *coordinator* part responsible for executing a coupling, and a *root-coordinator* responsible for the overall model execution. Furthermore, we stick to the format known from [1], using message passing. There are five types of messages used:

`i-message` The initialization message is sent to every component at simulation start. It is used to initialize state variables and gather the times of the first internal events at the single components.

`*-message` In PDEVS, this is the *internal state transition message* because there the output function λ is inseparably connected to the internal and confluent state transitions δ_{int} and δ_{conf}. However, in RPDEVS, λ is calculated in an iterative manner and on every kind of event. Thus, this message is solely used to trigger the λ iteration.

`y-message` The `y-message` is used to transport the output message calculated in λ to the parent coordinator where it is forwarded to the input bag of the receiving component.

`x-message` In RPDEVS, the `x-message` is used to trigger the state transition. Whenever a component receives an `x-message`, it executes δ and then calculates the time of its next internal event t_n.

`done-message` This message is used for synchronization. When the coordinator triggers child components to do their initialization or to conduct their state transition, it has to wait until all of them are done before simulation can proceed.

2.1 Simulator

The simulator of an atomic RPDEVS is nearly identical to the one of an atomic PDEVS (see [1], p. 285):

```
RPDEVS-simulator
  variables:
    parent     // parent coordinator
    tl         // time of last event
    tn         // time of next event
    RPDEVS     // assoc. model with total
               // state (s,e), time advance
               // function, lambda and delta
   (s_i,e_i)  // initial total state
    y          // output message bag

  when receive i-message(i,t)
   (s,e) = (s_i, e_i)
    tl = t - e
    tn = tl + ta(s)
    send done-message(done, tn) to parent

  when receive *-message(*,x,t)
    e = t - tl
    y = lambda(s,e,x)
    send y-message(y,t) to parent

  when receive x-message(x,t)
    s = delta(s,e,x)
    tl = t
    tn = tl + ta(s)
    send done-message(done, tn) to parent
end RPDEVS-simulator
```

The most important differences compared to the PDEVS simulator are the additional parameter `x` of the `*-message`, and the absence of the case distinction between *internal*, *external*, and *confluent* event when receiving an `x-message`. Like in [11], a `done-message` is used for synchronization during the potentially parallel execution to prevent the problems with Zeigler's PDEVS algorithm described in [12].

2.2 Coordinator

The more interesting part of the abstract simulator is the *coordinator*. We start with the definition of all necessary variables followed by the `i-message` and `done-message` procedures:

```
RPDEVS-coordinator
  variables:
    parent     // parent coordinator
    tl         // time of last event
    tn         // time of next event
    RPDEVS     // associated coupled model
               // including index set D,
               // influencer sets I_d, and
               // output transl. fcts. Z_id
    event-list // list of elements (d,tn_d),
               // sorted ascending by tn_d
    IMM        // imminent children
    y_coupling // output message of coupling
    x_dr       // sub input bags:
               // d... sender, r... receiver
    x_r        // input bag of component r
    y_dN       // sub output bag of coupling
               // N, d... sender
    INF        // set of influenced children
               // (with changed input bag)
    INF'       // INF for next lambda-iter.
    DELTA      // set of children who need to
               // conduct a state transition
    CHECK      // components with withdrawn
               // input messages

  when receive i-message(i,t)
    DELTA = D
    for-each d in D do
      send i-message(i,t) to child d
    wait until DELTA = {}
    sort event-list according to tn_d
    tl = max{tl_d : d in D}
    tn = min{tn_d : d in D}
    send done-message(done, tn) to parent

  when receive done-message(done, td) from d
    event-list.(d,tn_d) = (d,td);
    remove d from DELTA
```

At simulation start, the coordinator receives an `i-message` from its parent coordinator. The parent of the uppermost coordinator is the *root-coordinator* (see Section 2.3). The `i-message` is forwarded to all child components `d∈D` which causes them to calculate their time of next internal event `tn_d`. Then, the coordinator waits until all children have sent their `done-message` (i.e. `DELTA={}`) before the time of the next internal event `tn` of the coupling can be determined.

If a component is imminent (i.e. its time of next event `tn=t`), it receives a `*-message` from its parent coordinator. This message initiates the λ iteration in the coupling. The goal of the λ iteration of a coupling is generating its output message `y_coupling`.

```
when receive *-message(*,x,t)
  y_coupling = {}
  for-each (d,tn_d) in event-list with tn_d=t
    add d to IMM, DELTA and INF
    remove (d,tn_d) from event-list
  for-each r in D with N in I_r
    if x_Nr != Z_Nr(x)
      x_Nr = Z_Nr(x)
      add r to INF and DELTA
      if x_Nr={}
        add r to CHECK
  for-each r in INF
    x_r = {x_dr : d in I_r, x_dr != {}}
  while CHECK != {}
    pick and remove r from CHECK
    if x_r={}
      if r not in IMM
        remove r from INF and DELTA
        for-each d in D with r in I_d
          x_rd = {}
          remove x_rd from x_d
          add d to CHECK
        if r in I_N
          y_rN = {}
  INF'={}
  for-each r in INF
    send *-message(*,x_r,t)
```

In the `*-message` of the coordinator first, the imminent children are determined and collected in `IMM`, `INF`, and `DELTA`. Then, the components' input bag changes caused by the couplings input are calculated. All components whose input bag changed are added to `INF` and scheduled for state transition by adding them to `DELTA`. Finally, `*-messages` are sent to the affected child components triggering their λ execution.

These λ executions result in output messages transported via `y-messages` back to the coordinator. In the coordinator's `y-message` procedure, all output messages of all triggered child components are gathered and converted using the output translation functions `Z_dr`. Depending on the coupling relations, they are converted either into input messages for other children or into coupling output messages. Thereby, all child components whose input bag has changed are collected in `INF'`. After the last element in `INF` has responded to the coordinator with a `y-message`, the components in `INF'` are shifted into `INF`. If `INF` is not empty after that, again a `*-message` is sent to every component in `INF` and their response, in form of `y-messages` is awaited. However, if `INF` is empty at the end of the `y-message` procedure, it means no input bag has changed during the last λ iteration, i.e. they are stable. Thus, the λ iteration of the coupling has terminated and the coordinator can send a `y-messages` to its parent. In [8], it is shown that for models without algebraic loops, the λ iteration always terminates after a maximum of $n = |D|$ iterations. In some cases, algebraic loops can even be solved by the simulation algorithm (see RS flip-flop in [13]).

During the course of λ iterations, it may happen that input messages for child components that were produced in previous iterations may have to be withdrawn from the respective input bag. Thereby, it may occur that the input bag becomes completely empty although it was not in the preceding iteration. These components then need to be checked separately because they may already have produced output in reaction to a non-empty input bag (Mealy behavior) and thereby may have influenced other components. This task is handled via the set `CHECK`.

A coordinator may represent a coupling that is used as component in a parent coupling. In this parent coupling, there is also a λ iteration in progress. Thus, the parent coordinator may send multiple `*-messages` to its child coordinators. This is why the `*-message` procedure of the coordinator also has to check whether formerly received coupling inputs still exist in the new iteration (using `CHECK`).

```
when receiving y-message(y_d,t) from d
  remove d from INF
  if d in I_N
    y_dN = Z_dN(y_d)
  for-each r in D with d in I_r
    if x_dr != Z_dr(y_d)
      x_dr = Z_dr(y_d)
      if x_dr={}
        add r to CHECK
      add r to INF' and DELTA
  if INF = {}
    INF = INF'
    INF' = {}
    for-each r in INF
      x_r = {x_dr : d in I_r, x_dr != {}}
    while CHECK != {}
      pick and remove r from CHECK
      if x_r={}
        if r not in IMM
          remove r from INF and DELTA
          for-each d in D with r in I_d
            if x_rd != {}
              x_rd = {}
              remove x_rd from x_d
              add d to INF, DELTA and CHECK
          if r in I_N
            y_rN = {}
    for-each r in INF
      send *-message(*,x_r,t) to component r
    if INF = {}
      y_coupling={y_dN : d in I_N, y_dN!={}}
      send y-message(y_coupling,t) to parent
```

Receiving an `x-message` means that the λ iteration is finished and the state transitions can be conducted. This is done by sending an `x-message` to every imminent child component and to every child component with non-empty input bag. These components have been gathered in `DELTA` during the λ iteration. After the `x-messages` are sent, the coordinator waits until all of them are done with their state transition. Afterwards, the time of the next internal event can be calculated and the set `IMM` is cleared.

```
when receive x-message(x,t)
  for-each r in DELTA
    send x-message(x_r,t) to r
  wait until DELTA = {}
  sort event-list according to tn_d
  tl = t
  tn = min{tn_d: d in D}
  IMM = {}
  send done-message(done, tn) to parent
end RPDEVS-coordinator
```

2.3 Root Coordinator

Finally, on top of the uppermost coordinator is the root coordinator. It starts the simulation by sending an `i-message` to its child coordinator. Then it advances the simulation time to the time of next event, initiates the λ iteration by sending a `*-message`, waits until the λ iteration is finished and then triggers the state transition by sending an `x-message`. This is repeated until the simulation time `t` exceeds the final time `tend`.

```
RPDEVS-root-coordinator
  variables:
    tstart // simulation start time
    tend   // simulation end time
    t      // current simulation time
    child  // direct subordinate coordinator

  t = tstart
  send i-message(i,t) to child
  wait for done-message(done,tn) from child
  t=tn
  while t < tend
    send *-message(*,t) to child
    wait for y-message(y,t) from child
    send x-message({},t) to child
    wait for done-message(done,tn) from child
    t=tn
end RPDEVS-root-coordinator
```

3 Conclusion

In this work, we first recapped RPDEVS and pointed out the parallels of PDEVS and RPDEVS to Moore and Mealy automata. Furthermore, we demonstrated how the input bags can be formally split up into sub-bags, one for each influencer. This separation is used by the abstract simulator as it makes it easier to detect input bag changes due to λ recalculations in the influencers. The structure of the abstract simulator is basically similar to the one of Zeigler's PDEVS abstract simulator [1]. For synchronization purposes though, we also added the `done-message` of Chow's algorithm [11].

When implementing the algorithm, especially when facilitating parallelism, aspects like consistent global variable manipulation and execution order have to be taken into account. However, this degree of detail would go beyond the scope of this paper.

Nevertheless, there already exists a proof-of-concept implementation of an RPDEVS simulator. We reprogrammed the simulation engine of the open-source classic DEVS simulator *PowerDEVS* and named it *PowerRPDEVS*. It is available on *SourceForge* [14]. In PowerRPDEVS, a sequential version of the algorithm is implemented. Exploitation of parallelism in the PowerRPDEVS engine is an issue for future work.

Acknowledgement
We thank all partners of the project ASPeCT for their contributions. The presented work is funded by the Austrian Research Promotion Agency within the program *Produktion der Zukunft*, project number 858655.

References

[1] Zeigler BP, Praehofer H, Kim TG. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press. 2000.

[2] Joslyn C. The process theoretical approach to qualitative DEVS. *Proc 7th Conf on AI, Simulation, and Planning in High Autonomy Systems*. 1996;.

[3] Klimovich AS, Solov'ev VV. Transformation of a Mealy Finite–State Machine into a Moore Finite–State Machine by Splitting Internal States. *J Comput Syst Sci Int*. 2010;49(6):900–908.

[4] Chow ACH, Zeigler BP. Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In: *Proc. of WSC'94*. 1994; pp. 716–722.

[5] Preyser FJ, Heinzl B, Raich P, Kastner W. Towards Extending the Parallel-DEVS Formalism to Improve Component Modularity. In: *Beiträge zum WS der ASIM /GI-Fachgruppen STS und GMMS 2016*; pp. 83–89.

[6] Preyser FJ. An Approach to Develop a User Friendly Way of Implementing DEV&DESS Models in PowerDEVS. Masterthesis, TU Wien. 2015.

[7] Raich P, Heinzl B, Preyser F, Kastner W. Modeling Techniques for Integrated Simulation of Industrial Systems Based on Hybrid PDEVS. In: *Proc. of MSCPES'16*, 1. 2016; pp. 1–6.

[8] Preyser F, Heinzl B, Kastner W. RPDEVS: Revising the Parallel Discrete Event System Specification. In: *Proc. of MATHMOD 2018*; pp. 269–274.

[9] Goldstein R, Breslav S, Khan A. Informal DEVS conventions motivated by practical considerations. In: *Proc. of DEVS 2013*; pp. 10:1–10:6.

[10] Goldstein R, Breslav S, Khan A. Practical aspects of the DesignDEVS simulation environment. *Simulation*. 2017;94(4):301–326.

[11] Chow AC, Zeigler BP, Kim DH. Abstract Simulator for the Parallel DEVS Formalism. In: *Proc. of the Fifth Annual Conference on AI, and Planning in High Autonomy Systems*. 1994; pp. 157–163.

[12] Schwatinski T, Pawletta T. An advanced simulation approach for parallel DEVS with ports. In: *Proc. of SpringSim '10*. 2010; pp. 147–154.

[13] Fiedler C, Preyser F, Kastner W. Simulation of RPDEVS Models of Logic Gates. In: *Proc. of ASIM-Workshop Simulation technischer Systeme/Grundlagen und Methoden in Modellbildung und Simulation*. 2019; p. 6.

[14] PowerRPDEVS on sourceforge:. `https://sourceforge.net/projects/powerrpdevs/`. [accessed: 2019-01-04].