Accurate and Efficient Continuous Time and Discrete Events Simulation in SystemC

Breytner Fernández-Mesa, Liliana Andrade, Frédéric Pétrot
Univ. Grenoble Alpes, CNRS, Grenoble INP*, TIMA, 38000 Grenoble, France
{breytner.fernandez, liliana.andrade, frederic.petrot}@univ-grenoble-alpes.fr

***Abstract*—The AMS extensions of SystemC emerged to aid the virtual prototyping of continuous time and discrete event heterogeneous systems. Although useful for a large set of use cases, synchronization of both domains through a fixed timestep generates inaccuracies that cannot be overcome without penalizing simulation speed. We propose a direct, optimistic, and causal synchronization algorithm on top of the SystemC kernel that explicitly handles the rich set of interactions that occur in the domain interface. We test our algorithm with a complex nonlinear automotive use case and show that it breaks the described accuracy and efficiency trade-off. Our work enlarges the applicability range of SystemC AMS based design frameworks.**

I. Introduction

The need for a practical and unified approach for the virtual prototyping of digital systems in close interaction with their physical environment has led to the definition of the AMS extensions of SystemC more than a decade ago [1]. Given the complexity of the modeled systems, high-level representations of both the discrete event (DE) and continuous time (CT) components are required, so that simulation speed is acceptable. To that aim, SystemC AMS introduced the Timed Data Flow (TDF) model, that advances CT by a statically, user-defined, simulation timestep. The user is then faced with a dilemma: either the timestep is small, the simulation accurate but slow, or the timestep is large, the simulation possibly inaccurate, but fast.

Solving the CT/DE simulation problem in its generality is very challenging. In this work, we give a small but still useful step. We follow the observation that CT/DE interactions are event-based. A continuous value is meaningful to the DE domain only if it generates an event, *i.e.* when it reaches a given threshold. Discrete events are meaningful to the CT domain when they modify the equations and state of the system. Synchronizing at the time of these events is more efficient than doing it at a fixed timestep.

Our solution must fit within the existing SystemC framework, without kernel modifications. Extensions such as SystemC AMS [2] and SystemC MDVP [3] embed the model and solver in a DE process (synchronization process) launched from **end_of_elaboration**, a SystemC callback executed before starting simulation. We follow the same approach. From the SystemC kernel's perspective, the synchronization process is a DE process and it must expose a DE interface: timely react to input events, generate output events, and schedule its own reactivations. This process implements the synchronization algorithm that is the core of this paper. As a final constraint, given the amount of state variables that exists in a processor based digital system, the DE part should never rollback.

*Institute of Engineering Univ. Grenoble Alpes

The paper is organized as follows: Section II briefly reviews the related work, Section III details our CT/DE synchronization algorithm, Section IV presents the experiments and compares with other approaches, and finally Section V concludes.

II. Related work

One major advancement in the solution of the CT/DE simulation problem in SystemC AMS was the introduction of the Dynamic Timed Data Flow (DTDF) paradigm that avoids small fixed timesteps by using an event-triggered approach [2]. Although DTDF has proved helpful in many use cases, it partly relies on the user to schedule synchronization (e.g. through the **set_timestep** and **request_next_activation** functions) and it is not available for the Linear Signal Flow and Electrical Linear Networks models of computation (MoCs) that continue to synchronize through a static TDF layer.

Apart from SystemC AMS, other SystemC-based tools have been proposed to target heterogeneity. SystemC MDVP eases the hierarchical composition of different MoCs following a master-slave relation and detects synchronization causality issues between DE and TDF MoCs before simulation [3], [4]. HetMoC [5] and ForSyDe SystemC [6] rely on a formal framework to enable analyzability and synthesis of heterogeneous models. The work of [7] adds Synchronous Data Flow modeling to SystemC as an attempt to support heterogeneity. None of them implements direct CT/DE synchronization.

Some other tools and frameworks have defined and implemented CT/DE synchronization. SystemC-A implements a lockstep algorithm and supports nonlinear and distributed systems [8], but it modifies the SystemC kernel and does not exactly find the occurrence time of CT output events. The work in [9] proposes a CT/DE framework for the design of co-simulation tools and applies it to SystemC/MATLAB. The work in [10] describes a variety of conservative algorithms for VHDL-AMS. Ptolemy II proposes an optimistic approach and formally defines a set of conditions to avoid causality issues and DE kernel rollback [11].

Based on these works, we propose and implement a solution that is optimistic and causal, avoids DE kernel rollback, manages the instantaneous effect of input events on the CT models, and does not modify the SystemC kernel.

978-3-9819263-4-7/DATE20/©2020 EDAA

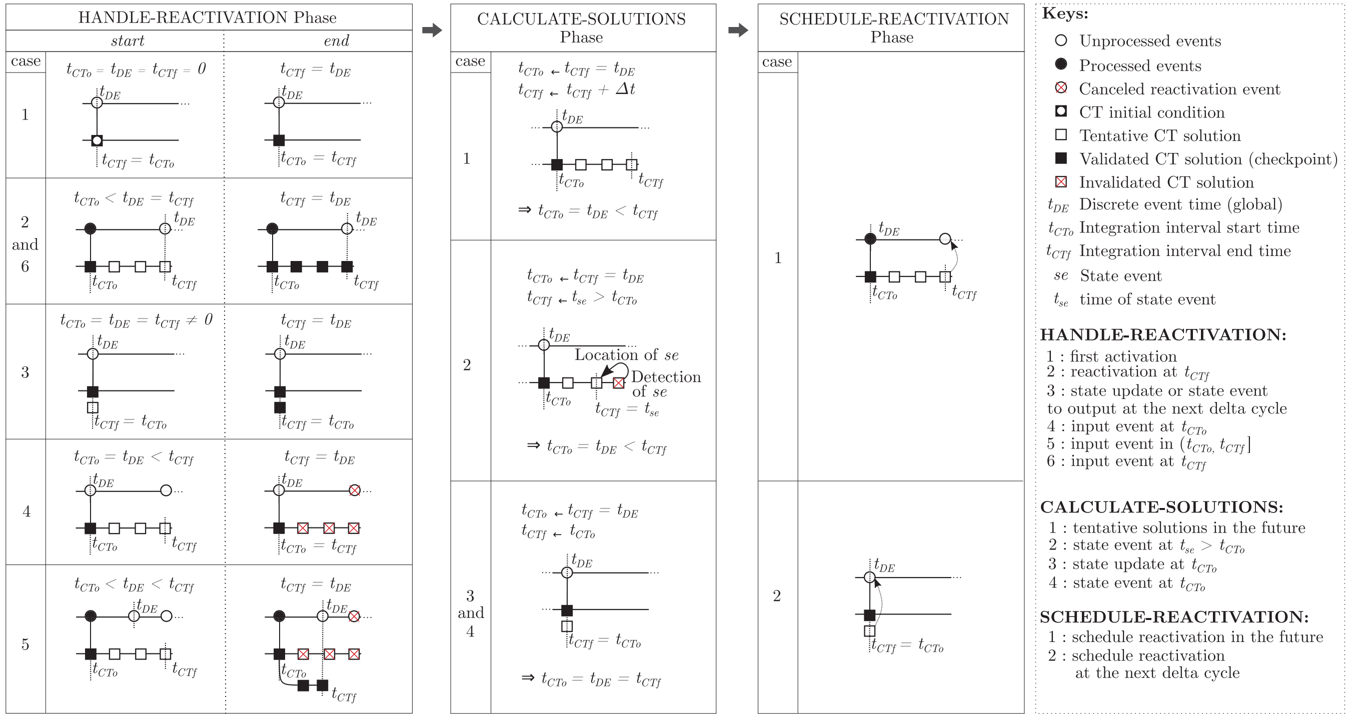

Fig. 1. Graphical representation of the CT/DE synchronization algorithm.

III. CT/DE Synchronization Algorithm

We describe the algorithm in a top-down approach.

A. *Top view*

Alg. 1 constitutes the top view. By being registered as a process, it is activated several times under control of the DE kernel whose time (t_{DE}) constitutes the global simulation time. At each activation, it executes optimistically: solutions cover the interval I_{v} that goes from the current time ($t_{\mathrm{CT_o}} = t_{\mathrm{DE}}$) to some time in the future ($t_{\mathrm{CT_f}}$). The process suspends until $t_{\mathrm{CT_f}}$, time at which a new interval will begin.

Alg. 1 Synchronization-Algorithm

```
1: while true do
2:     HANDLE-REACTIVATION()
3:     CALCULATE-SOLUTIONS()
4:     SCHEDULE-REACTIVATION()
5: end while
```

Fig. 1 pictures the Synchronization-Algorithm. Columns represent its three phases. Handle-Reactivation creates checkpoints of solutions and generates outputs from the CT to the DE domain. Calculate-Solutions evolves the CT state over an interval I_{v}. Schedule-Reactivation schedules the next reactivation and then suspends the process. Each phase handles different cases represented by rows. Inside rows, we find the timeline of the DE kernel at the top and that of the synchronization process at the bottom. Circles on the DE timeline are events. Squares on the CT timeline are CT solutions.

B. Handle-Reactivation *Phase*

The process can be reactivated because of the initialization subphase of simulation, a reactivation scheduled by itself or the notification of an input event. Reactivations can be scheduled either in the future or at the next delta cycle. Input events can be notified at the start, in the middle or at the end of the last integration interval; since they are able to instantaneously modify the CT model, the process must react to them at their exact notification time. Handle-Reactivation deals with 6 cases (Fig. 1):

1) First activation: after initialization, the process has not calculated solutions yet, but it generates outputs and creates a checkpoint based on the system initial condition.

2) Self-reactivation with solutions in the future: during the previous execution, the process calculated solutions over the interval $I_{\mathrm{v}} = (t_{\mathrm{CT_o}}, t_{\mathrm{CT_f}}]$ and scheduled a reactivation at $t_{\mathrm{CT_f}}$. It reactivates at $t_{\mathrm{DE}} = t_{\mathrm{CT_f}}$ and solutions are valid from start to end of I_{v}; it generates outputs and creates a checkpoint.

3) Self-reactivation with solutions in the present: reactivation occurs in a delta cycle ($t_{\mathrm{CT_o}} = t_{\mathrm{DE}} = t_{\mathrm{CT_f}} \neq 0$). Although there was already a valid solution at $t_{\mathrm{CT_o}}$, the process has calculated another solution at $t_{\mathrm{CT_o}}$ that is also valid (see Sec. III-C); it generates outputs and creates a checkpoint on the new solution.

4) Reactivation by an input at the start of I_{v}: the process activated at $t_{\mathrm{CT_o}}$, calculated solutions over the interval $I_{\mathrm{v}} = (t_{\mathrm{CT_o}}, t_{\mathrm{CT_f}}]$, and scheduled a reactivation at $t_{\mathrm{CT_f}}$; but an input event activates the process for the second time at $t_{\mathrm{CT_o}} = t_{\mathrm{DE}} < t_{\mathrm{CT_f}}$ (delta cycle). Input events invalidate solutions — checkpoint restoration at $t_{\mathrm{CT_o}}$. Outputs at $t_{\mathrm{CT_o}}$ were already

generated in the previous activation where the checkpoint on $t_{\mathrm{CT_o}}$ was created, so they are not generated again.

5) Reactivation by an input in the middle of I_{v}*:* reactivation occurs at some t_{DE} such that $t_{\mathrm{CT_o}} < t_{\mathrm{DE}} < t_{\mathrm{CT_f}}$ and solutions are invalid from t_{DE} forward. The process restores the checkpoint at $t_{\mathrm{CT_o}}$, recalculates solutions from $t_{\mathrm{CT_o}}$ to t_{DE} based on the previous inputs (CATCH-UP), generates outputs, and creates a checkpoint at t_{DE}. Solutions from t_{DE} forward will be based on the new inputs. A checkpoint of the immediately previous inputs is needed when catching up. CATCH-UP (Alg. 3) avoids the process to be left behind t_{DE}, which would violate causality.

6) Reactivation by an input at the end of I_{v}*:* reactivation occurs at $t_{\mathrm{DE}} = t_{\mathrm{CT_f}}$; solutions are valid from start to end of the interval. As new inputs affect only the integration interval that begins, the situation is handled exactly as in case 2.

Alg. 2 and 3 condense the previous discussion. X is the CT state, X_{cp} is a checkpoint on X, I is the set of inputs, and I_{cp} is a checkpoint on the previous inputs. GENERATE-OUTPUTS is a function specific to the CT model that maps the state, inputs and global simulation time to the set of outputs.

Alg. 2 HANDLE-REACTIVATION

```
    // Case 4
 1: if t_CTo = t_DE < t_CTf then
 2:     X ← X_cp                          // Restore state
 3:     t_CTf ← t_CTo = t_DE              // Restore interval limits
 4: else                                  // Cases 1, 2, 3, 5 and 6
        // Case 5
 5:     if t_CTo < t_DE < t_CTf then
 6:         X ← X_cp                      // Restore state
 7:         t_CTf ← t_CTo                 // Restore interval limits
 8:         CATCH-UP()
 9:     end if
        // Cases 1, 2, 3, 5 and 6
10:     GENERATE-OUTPUTS(X, I_cp)
11:     X_cp ← X                          // Create state checkpoint
12:     I_cp ← I                          // Create input checkpoint
13: end if
```

Alg. 3 CATCH-UP

```
 1: t_CTf ← t_DE                          // Advance t_CTf
 2: INTEGRATE(t_CTo, t_CTf, X, I_cp)      // Evolve the state
```

C. CALCULATE-SOLUTIONS *Phase*

The CT state evolves in two ways: continuously over an interval or discontinuously through a finite number of delta cycles. It is of interest to inform the DE domain when the state meets any user-defined condition (*state event*). Similar to other events, state events can trigger the execution of processes, which is why it is essential to notify them at their exact time. One subtlety is that the state event conditions may depend not only on the state, but also on the inputs from the DE domain. An input event that modifies the state event conditions may produce one or more of them to be met instantaneously without the state even changing. In this case, we say that the state event has been triggered by an input event.

CALCULATE-SOLUTIONS evolves the state continuously and discontinuously, and detects state events triggered by the evolution of the state and by input events:

1) Continuous evolution without state events: the synchronization process invokes a CT solver to calculate solutions over the interval $I_{\mathrm{v}} = (t_{\mathrm{CT_o}}, t_{\mathrm{CT_f}}]$ and schedules a reactivation at the end of the interval.

2) Continuous evolution with state events: the process locates state events while calculating solutions over an interval $(t_{\mathrm{CT_o}}, t_{\mathrm{CT_f}}]$. This interval is divided in integration steps by the CT solver. It suffices to test the state event detection conditions at each integration step (INTEGRATE); once detected, a root finding method locates the time of occurrence (t_{se}). The process sets $t_{\mathrm{CT_f}} \leftarrow t_{\mathrm{se}}$ and schedules a reactivation.

3) Discontinuous evolution: assume the following preconditions: there are available solutions at $t_{\mathrm{CT_f}}$, the global time t_{DE} has advanced to $t_{\mathrm{CT_f}}$, and there is an input event at $t_{\mathrm{CT_f}}$.

Let us denote the state of the CT system at $t_{\mathrm{CT_f}}$ as X. Discontinuous evolution refers to a discontinuous change from X to some other state X' produced by an input event. The dilemma is that both X and X' are valid states for the same time instant. X results from the system evolution up until $t_{\mathrm{CT_f}}$. X' results from the execution of an input event.

The process executes all discontinuous updates (EXECUTE-UPDATES) before calculating new solutions. If there is at least one update, it sets $t_{\mathrm{CT_f}}$ to $t_{\mathrm{CT_o}} = t_{\mathrm{DE}}$ in order to generate outputs at the next delta cycle. There can be multiple finite discontinuous updates at a particular time.

4) State events triggered by input events: assume the same preconditions of the previous case. An input event may produce X to instantly meet the state event conditions. CALCULATE-SOLUTIONS tests for events immediately after EXECUTE-UPDATES. If it detects at least one, it skips integration and sets $t_{\mathrm{CT_f}}$ to $t_{\mathrm{CT_o}} = t_{\mathrm{DE}}$ to output the event.

Alg. 4 provides the pseudocode. EXECUTE-UPDATES and IS-EVENT are specific to the CT model. EXECUTE-UPDATES produces discontinuous updates to the state triggered by input events; it returns true if there is at least one update and false otherwise. IS-EVENT returns true if the given state and inputs meet at least one of the state event conditions and false otherwise. X is mutable in EXECUTE-UPDATES.

INTEGRATE finds the solutions from $t_{\mathrm{CT_o}}$ to $t_{\mathrm{CT_f}}$ and assigns them to X. It tests IS-EVENT after each integration step and returns whether or not the event is detected and the time values that enclose it. It should implement a CT solver appropriate to the equations and error requirements of the application. LOCATE implements a root finding algorithm; it returns the time of occurrence of a state event and the related state given an interval enclosing it, the state, and the inputs. It has access to IS-EVENT. X is mutable in both functions. Reference [12] describes methods for solving differential equations and finding roots and the odeint Boost library [13] implements a set of them in C++.

Alg. 4 CALCULATE-SOLUTIONS

1: $t_{\mathrm{CT_o}} \leftarrow t_{\mathrm{CT_f}} = t_{\mathrm{DE}}$ // Integration interval start time
2: $t_{\mathrm{CT_f}} \leftarrow t_{\mathrm{CT_f}} + \Delta t$ // Tentative end time
// Case 3 Discontinous evolution
3: $is_update \leftarrow$ EXECUTE-UPDATES(X, I)
// Case 4 State event triggered by input events
4: $is_s_event_by_input \leftarrow$ IS-EVENT(X, I)
// Cases 3 and 4
5: **if** is_update **or** $is_s_event_by_input$ **then**
6: $t_{\mathrm{CT_f}} \leftarrow t_{\mathrm{CT_o}}$ // Interval of length 0. Delta cycle.
7: **else** // Cases 1 and 2. Integration.
8: $(event_detected, t_{\min}, t_{\max}) \leftarrow$ INTEGRATE$(t_{\mathrm{CT_o}}, t_{\mathrm{CT_f}}, X, I)$
// Case 2 Continuous evolution with state events.
9: **if** $event_detected$ **then**
10: $t_{\mathrm{se}} \leftarrow$ LOCATE$(t_{\min}, t_{\max}, X, I)$
11: $t_{\mathrm{CT_f}} \leftarrow t_{\mathrm{se}}$ // Interval ends at event time
12: **end if**
13: **end if**

Alg. 5 SCHEDULE-REACTIVATION

1: WAIT$((t_{\mathrm{CT_f}} - t_{\mathrm{DE}})$ **or** $input_events_list)$

D. SCHEDULE-REACTIVATION *Phase*

The SystemC **wait** function suspends a process until a given time has elapsed or one or more events of a list occur (Alg. 4). In our case, the elapse time is the difference between $t_{\mathrm{CT_f}}$ and the global simulation time t_{DE}; $t_{\mathrm{CT_f}} - t_{\mathrm{DE}} > 0$ schedules a reactivation in the future (case 1) and $t_{\mathrm{CT_f}} - t_{\mathrm{DE}} = 0$ produces a delta cycle (case 2). The list of events ($input_events_list$) is created by traversing the input ports while invoking their **value_changed_event** method to get a reference to the event; ports are traversed just once before launching the process.

E. Wrap-up

HANDLE-REACTIVATION, CALCULATE-SOLUTIONS and SCHEDULE-REACTIVATION execute in loop. At the end of SCHEDULE-REACTIVATION, the process will reactivate either with solutions in the future or with one solution in the present. Solutions in the future lead to cases 2, 4, 5 and 6 of HANDLE-REACTIVATION. One solution in the present invariably leads to case 3. No matter the case, HANDLE-REACTIVATION ensures that $t_{\mathrm{CT_f}}$ equals t_{DE} at the end, just before invoking CALCULATE-SOLUTIONS. This phase, in turn, ends up advancing $t_{\mathrm{CT_f}}$ further in the future or not, which determines the two cases of SCHEDULE-REACTIVATION.

IV. CASE STUDY: AUTOMATIC TRANSMISSION CONTROL OF A VEHICLE'S LONGITUDINAL DYNAMICS

We model the continuous time longitudinal dynamics of a vehicle and its discrete events transmission control unit (Fig. 2). The CT Vehicle Model is embedded in a SystemC module and it interacts with the DE Transmission Control Unit module through the SYNCHRONIZATION-ALGORITHM.

A. CT Vehicle Model

The longitudinal dynamics of a vehicle is represented by the equations in Fig. 2 that account for the angular acceleration of the engine Eq. (1), the vehicle speed Eq. (2), and the pressure of the braking system Eq. (3) [14]. The model contains the IS-EVENT function that returns true when the speed crosses a given pair of thresholds (See IV-B) and the GENERATE-OUTPUTS function that maps the speed value and gear input to an output *speed_threshold_sg* signal indicating the speed position (up or down) w.r.t. the pair of thresholds.

B. DE Transmission Control Unit Model

The transmission control unit selects a gear position based on the speed to transmit an appropriate amount of power from the engine to the wheels. It has 4 gear positions (State machine 1 in Fig. 2). Each gear imposes an up and down threshold to the speed and any violation to them results in a gear upshift or downshift, respectively. To filter out false positives, the unit waits for the speed to remain at least $0.08\,\mathrm{s}$ above (or below) the threshold before shifting (State machine 2 in Fig. 2) [15].

The *gear_sg* and *speed_threshold_sg* signals link the CT and DE models. For each gear position in the control unit, a particular value of the gear ratio R_{tr} is used in Eq. (2). The *speed_threshold_sg* signal controls state machine 2.

C. Interactions between the continuous time and discrete events subsystems

Fig. 3 shows the speed response to a constant throttle. At time 0, the transmission control unit is in 1st gear and the speed begins to increase. When it reaches the up threshold, the event is communicated to the control unit to shift to 2nd gear. With a smaller gear ratio, less power goes from the engine to the wheels and the vehicle accelerates more slowly. The 2nd to 3rd upshift occurs when the speed reaches the 2nd gear's up threshold. So on and so forth. To gain better insight, let us describe the interactions in the frontier where the gear shifts from 1st to 2nd in five parts (Fig. 4):

1) Before the threshold crossing at time $2.607\,\mathrm{s}$*:* the control unit is in steady state at 1st gear. While the process calculates solutions, it detects the up threshold crossing and locates it at time $2.607\,\mathrm{s}$ (CALCULATE-SOLUTIONS, case 2). It schedules a reactivation at this time and suspends (SCHEDULE-REACTIVATION, case 1). The kernel advances time to $2.607\,\mathrm{s}$.

2) During the threshold crossing at time $2.607\,\mathrm{s}$*:* the process reactivates and outputs the state event (HANDLE-REACTIVATION, case 2). Since we use a fixed $\Delta t = 1\,\mathrm{s}$, it calculates tentative solutions over $I_{\mathrm{v}} = (2.607\,\mathrm{s}, 3.607\,\mathrm{s}]$ (CALCULATE-SOLUTIONS, case 1). It then schedules a reactivation at $3.607\,\mathrm{s}$ and suspends (SCHEDULE-REACTIVATION, case 1). The DE kernel informs the event to the control unit which activates at the next delta cycle, goes from steady to upshifting state, programs a $0.08\,\mathrm{s}$ timeout and suspends.

3) Between the threshold crossing and the first gear shift at time $2.687\,\mathrm{s}$*:* the DE kernel advances time to the next event at $2.687\,\mathrm{s}$, the $0.08\,\mathrm{s}$ timeout.

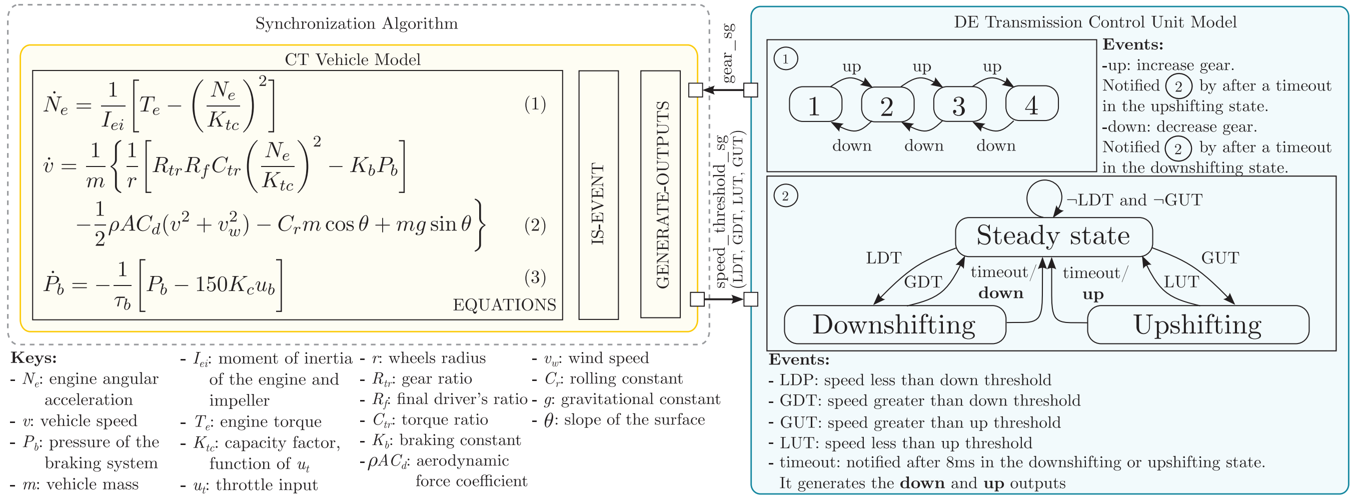

Fig. 2. CT/DE vehicle model.

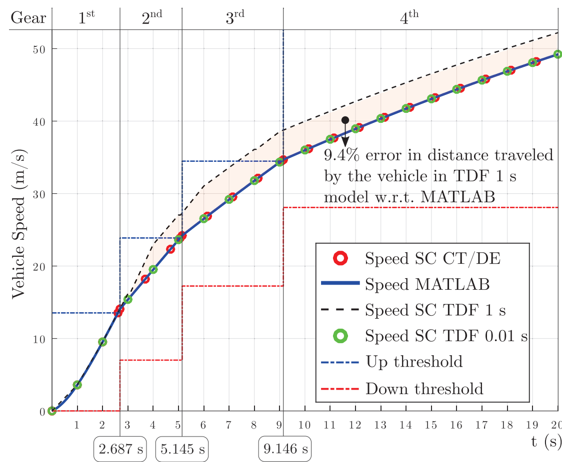

Fig. 3. Vehicle speed dynamics and gear shifts for a constant throttle input.

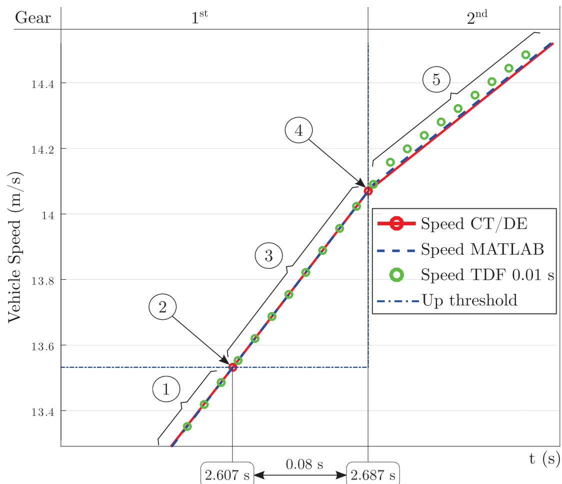

Fig. 4. Detailed CT/DE interactions at the shift frontier from 1st to 2nd gear.

4) During the gear shift at time $2.687\,\mathrm{s}$*:* the timeout reactivates the control unit, which shifts from 1st to 2nd and suspends. The shift reactivates the synchronization process at the next delta cycle: it identifies an activation in the middle of the last integration interval ($t_{\mathrm{DE}} = 2.687\,\mathrm{s} \in I_{\mathrm{v}} = (2.607\,\mathrm{s}, 3.607\,\mathrm{s}]$), restores the checkpoint at $2.607\,\mathrm{s}$, catches up to $2.687\,\mathrm{s}$, and generates outputs (HANDLE-REACTIVATION, case 5). Then, it tries to calculate tentative solutions, but it cannot because the gear is accompanied by new up and down thresholds; the speed is no longer above the up threshold but below, which is a state event triggered by an input change (CALCULATE-SOLUTIONS, case 4). It schedules a reactivation at the next delta cycle and suspends (SCHEDULE-REACTIVATION, case 2). At reactivation, it outputs the event (HANDLE-REACTIVATION, case 3), calculates tentative solutions (CALCULATE-SOLUTIONS, case 1), schedules a reactivation and suspends (SCHEDULE-REACTIVATION, case 1). The DE kernel updates the *speed_threshold_sg* signal and informs the event to the control unit.

5) After the gear shift at time $2.687\,\mathrm{s}$*:* the synchronization process continues to calculate solutions until finding the next threshold crossing, in a similar fashion to step 1.

D. Implementation of the CT vehicle model as a TDF module

TDF modules define time domain processing (**processing** method) [2] and they can be used to discretize CT dynamics. We implement the exact same nonlinear CT system inside one of such modules to compare the SystemC AMS Proof-of-Concept [16] TDF synchronization to ours. The UML class diagram of Fig. 5 describes this model. To focus on the essentials, we omit most of the methods of **sca_tdf::sca_module**, attributes of **VehicleEquations**, and definitions of **EngineTorqueCalculator**, **TorqueConstantsCalculator**, and **ThresholdsCalculator**.

VehicleTDF, **VehicleEquations**, and **OdeSolver** are the three main classes. **VehicleTDF** contains a CT model (`eqs`), a CT `solver`, and implements **processing** to generate outputs and evolve the state. **VehicleEquations** encapsulates the parameters and the CT state (data type **State**); it implements the state event condition **get_th_crossing**, the **get_derivatives**

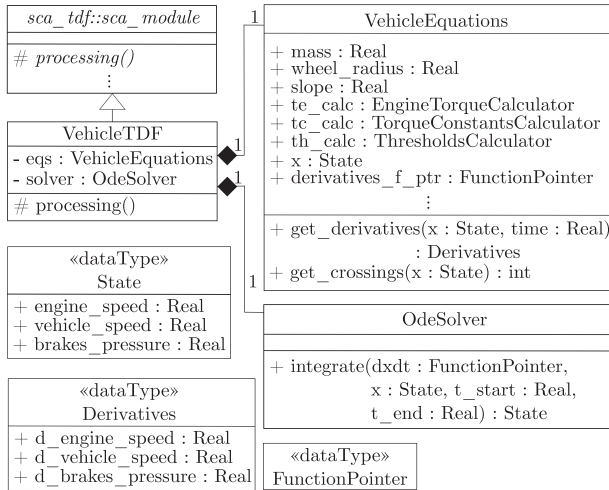

Fig. 5. Partial UML class diagram for the vehicle's CT TDF model.

TABLE I
ACCURACY AND EFFICIENCY IN THE VEHICLE MODEL SIMULATION.

Model	MATLAB	TDF 0.01 s	TDF 1 s	SYNC. ALGORITHM
Wall-clock time (s)	7.7	0.43	0.033	**0.030**
Speed-up factor	1	18	230	**260**
Distance traveled (m)	640	640	700	**640**
Error in the distance w.r.t. MATLAB (%)	0	0	9.4	**0**

function, and a pointer to such a function (**derivatives_f_ptr**). **OdeSolver** implements the **integrate** method that takes in a pointer to the derivatives function, the state, and the start and end time of the interval; and outputs the state at the end time. Integration is implemented by the odeint Boost library [13].

At each timestep, the **processing** function generates outputs and calculates the state. It updates the speed and threshold crossings output ports. It invokes the **integrate** method of the `solver` object for an interval starts at the current SystemC time and ends at the current time plus the module's timestep.

E. Discussion on the synchronization strategies

We take the distance traveled by the vehicle —integral of the speed, area under the curve— as a metric of accuracy and the simulation wall-clock time as a metric of efficiency (Tab. I). The reference is the MATLAB model. The TDF model with a timestep of 0.01 s executes 18 times faster than MATLAB and provides accurate results. The TDF model with a timestep of 1 s executes 230 times faster but its results have 9.4% error. The model synchronized by the SYNCHRONIZATION-ALGORITHM executes 260 times faster without error.

TDF restricts event notifications to a fixed timestep, independently of their exact time of occurrence; since notifications are not accurate, neither are simulation results. Small timesteps produce more context changes between the TDF process and the DE kernel; they are many and their cost accumulates. Our SYNCHRONIZATION-ALGORITHM avoids inaccuracies by reacting to input events and exactly locating output events. It increases execution speed by synchronizing only when needed. I.e, it breaks the accuracy and efficiency trade-off.

V. CONCLUSION AND FUTURE WORK

Our synchronization algorithm enables accurate and efficient CT/DE simulation in SystemC, advantages that we demonstrate by means of a complex automotive case study. It constitutes the basis of a generic CT/DE synchronization strategy for SystemC AMS. As such, we need to fit it in along the SystemC AMS ecosystem and to design an interface to seamlessly plug in CT MoCs and solvers. This would enlarge the applicability range of SystemC AMS for virtual prototyping of heterogeneous systems.

REFERENCES

[1] A. Vachoux *et al.*, "SystemC-AMS Requirements, Design Objectives and Rationale," in *2003 Design, Automation & Test in Europe (DATE)*. IEEE, 2003, pp. 388–393.

[2] IEEE Computer Society, *1666.1-2016 - IEEE Standard for Standard SystemC(R) Analog/Mixed-Signal Extensions Language Reference Manual*, IEEE.

[3] C. B. Aoun *et al.*, "Pre-Simulation Elaboration of Heterogeneous Systems: The SystemC Multi-Disciplinary Virtual Prototyping Approach," in *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*. IEEE, 2015, pp. 278–285.

[4] L. Andrade *et al.*, "Pre-Simulation Symbolic Analysis of Synchronization Issues between Discrete Event and Timed Data Flow Models of Computation," in *2015 Design, Automation & Test in Europe (DATE)*. IEEE, 2015, pp. 1671–1676.

[5] S. H. A. Niaki *et al.*, "Formal Heterogeneous System Modeling with SystemC," in *Proceeding of the 2012 Forum on Specification and Design Languages*. IEEE, 2012, pp. 160–167.

[6] J. Zhu *et al.*, "HetMoC: Heterogeneous Modelling in SystemC," in *2010 Forum on Specification & Design Languages (FDL 2010)*. IET, 2010.

[7] H. D. Patel *et al.*, "Towards a Heterogeneous Simulation Kernel for System-Level Models: a SystemC Kernel for Synchronous Data Flow Models," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 8, pp. 1261–1271, 2005.

[8] C. Zhao *et al.*, "An Extension to SystemC-A to Support Mixed-Technology Systems with Distributed Components," in *2011 Design, Automation & Test in Europe (DATE)*. IEEE, 2011.

[9] L. G. Iugan *et al.*, "A generic conceptual framework based on formal representation for the design of continuous/discrete co-simulation tools," *Design Automation for Embedded Systems*, vol. 19, no. 3, pp. 243–275, 2015.

[10] H. R. Ghasemi *et al.*, "An Effective VHDL-AMS Simulation Algorithm with Event Partitioning," in *18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design*. IEEE, 2005, pp. 762–767.

[11] J. Liu, "Continuous Time and Mixed-Signal Simulation in Ptolemy II," Dept. of EECS, University of California, Berkeley, CA, Tech. Rep., 1998.

[12] R. L. Burden *et al.*, *Numerical Analysis*. Belmont: Brooks/Cole, 2011. [Online]. Available: http://www.worldcat.org/isbn/9780538733519

[13] "Chapter 1. Boost.Numeric.Odeint," [Online]. Available: https://www.boost.org/doc/libs/1_66_0/libs/numeric/odeint/doc/html/index.html, (visited on 2019/13/09).

[14] P. Shakouri *et al.*, "Simulation Validation of Three Nonlinear Model-Based Controllers in the Adaptive Cruise Control System," *Journal of Intelligent & Robotic Systems*, vol. 80, no. 2, pp. 207–229, 2015.

[15] "Modeling an Automatic Transmission Controller," [Online]. Available: https://www.mathworks.com/help/simulink/slref/modeling-an-automatic-transmission-controller.html, (visited on 2019/26/07).

[16] "COSEDA Technologies GmbH — SystemC AMS Proof-of-Concept," [Online]. Available: https://www.coseda-tech.com/systemc-ams-proof-of-concept, (visited on 2019/02/09).

[17] F. Cremona *et al.*, "Hybrid co-simulation: its about time," *Software & Systems Modeling*, vol. 18, no. 3, pp. 1655–1679, 2019.

[18] D. Broman *et al.*, "Requirements for hybrid cosimulation standards," in *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*. ACM, 2015, pp. 179–188.