Efficient control representation in Digital Twins: an imperative challenge for declarative languages

Chiara Cimino, *Member, IEEE*, Federico Terraneo, *Member, IEEE*, Gianni Ferretti, *Senior Member, IEEE*, and Alberto Leva, *Member, IEEE*

***Abstract*— Digital Twins (DTs) are enablers for the fast optimisation processes required in the Industry 4.0 context. Declarative equation-based modelling languages, in turn, enable the creation of large-scale simulation-based DTs, as they relieve the analyst from creating the solution code. However, most industrial assets are Cyber-Physical Systems (CPSs), the Cyber part being their digital controls. With the available technology, a precise representation of modulating and logic controls conflicts with DT simulation performance. The result is a barrier to using DTs for system-level optimisation. We analyse the problem, propose a modelling paradigm to solve it and suggest how to integrate that paradigm into equation-based language compilers. We support our proposal by presenting a Modelica/C++ library, that we release as free software, built according to the said paradigm.**

***Index Terms*— Digital Twins; Cyber-Physical Systems; Modelling languages and compilers; Control system modelling; Simulation performance.**

I. Introduction

In the Industry 4.0 context, the role of digital technologies is fundamental [1] and twofold. On the one hand, they give intelligence (typically as controls) to processes and machines, making them Cyber-Physical Systems (CPSs). On the other hand, they provide Digital Twins (DTs) to help design, operate and manage the said systems [2], [3].

The concept of DT is used in various scientific and industrial domains, and as such has numerous interpretations [4]. These range from CAD (Computer Aided Design) documents through dynamic models to data-based decision aids and machine learning, from offline design tools to applications connected in real time to their physical counterparts, and more [5], [6]. Indeed, DTs are nowadays pervading the entire life of manufacturing assets [7], [8].

This paper concentrates on dynamic simulation models that include digital controls. Some interpretations consider such models to be DTs, while others consider them *parts* of a DT. But no matter which definition of DT one takes, and particularly if a real-time connection with the physical twin is required, an efficient simulation of controls is a must, especially in large-scale applications.

In the past, human analysts wrote simulation code in *imperative* languages, starting from the equations of the system to simulate. With the complexity of modern applications, doing so would require an effort incompatible with the time scale of the decisions to take. Nowadays, DTs call for Equation-Based Object-Oriented Modelling (EB-OOM) languages. These allow one to write the model *equations*, and feed these to a compiler that translates them into simulation code automatically. Since the analyst does not write instructions, these languages are termed *declarative* as opposite to *imperative*.

Our point is that the DT of a CPS has a seldom addressed peculiarity: it has to simulate not only the process physics but also something – as said, most typically controls – that is already digital in nature. The *replica* of control code in a simulator (e.g., by co-simulation) is the most natural idea but is computationally inefficient. Abstracted, declarative control models can recover efficiency, but also – as shown in the following – introduce subtle and highly undesired imprecisions.

An alternative solution to join precision and efficiency is the subject of this paper. After the introductory example of Section II, Section III introduces and motivates the mix of technologies on which the proposed solution is based, coming in Section IV to formalise the addressed research questions. Section V presents the solution, namely a model library – compatible with major industry standards – to simulate precisely and efficiently the Cyber part of a CPS, in turn enabling the creation of computationally efficient DTs for that CPS; the library is available at `https://github.com/looms-polimi/SFClib`. Sections VI and VII discuss application examples and related literature, while Section VIII draws some conclusions and outlines future research.

II. An explanatory and motivating example

Consider the nutshell-size CPS in Figure 1. The Physical part (the process) and the modulating control in the Cyber part – a Proportional-Integral (PI) controller – are described by the Differential Algebraic Equation (DAE) system and the transfer function in (1). The logic control in the Cyber part is made

Manuscript received XXX XX, XXXX; revised XXX XX, XXXX, XXX XX, XXXX, and XXX XX, XXXX; accepted XXX XX, XXXX. Date of publication XXX XX, XXXX; date of current versionXXX XX, XXXX. This work was supported by the the Lombardia regional research and innovation project AD-COM, ID 214632.

C. Cimino, F. Terraneo, G. Ferretti and A. Leva are with the Politecnico di Milano, Italy (e-mail: {chiara.cimino, federico.terraneo, gianni.ferretti, alberto.leva}@polimi.it).

of a relay with hysteresis having $\mp A_w$ as output values and $\mp y_{thr}$ as switching thresholds. In Figure 1 the PI controller acts on the control signal $u(t)$ so that the controlled variable $y(t)$ tracks the reference signal $w(t)$, that is set by the logic control block based on the value of $y(t)$. The example is just meant to introduce the addressed problem, hence numbers are inessential, but for completeness $\mu = 1$, $T_i = T = 2$, $K = 5$, $u_{min} = -2$, $u_{max} = 2$, $A_w = 1$, $y_{thr} = 0.95$.

$$y(t) + T\frac{dy(t)}{dt} = u(t), \quad C(s) = K\left(1 + \frac{1}{sT_i}\right). \tag{1}$$

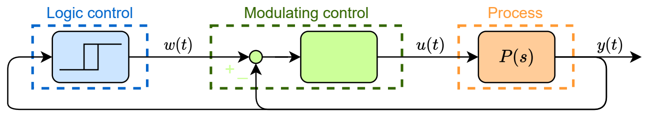


Fig. 1: A minimalistic CPS.

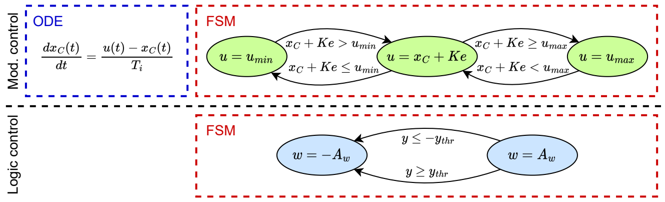


Fig. 2: Declarative representation for modulating and logic control in the CPS of Figure 1.

We assume that modulating and logic control are realised digitally, adopting the IEC (International Electrotechnical Commission) industry standards mentioned later on. Systems not falling in this category (e.g., analogue control devices or clockless logic) are a minimal minority and, in general, are small-size, hence not of interest for our research.

When creating a DT of this CPS, both modulating and logic control can be described in declarative form. Modulating control becomes an Ordinary Differential Equation (ODE) system of the type

$$\begin{cases} \frac{dx_C}{dt} &= f\left(x_C(t), w(t), y(t)\right) \\ u(t) &= g\left(x_C(t), w(t), y(t)\right) \end{cases} \tag{2}$$

where x_C, w, y and u are the controller state vector, the reference signal, the controlled variable and the control signal; such an ODE is typically specified as a block diagram made of transfer functions, coupled to a Finite State machine (FSM) to realise antiwindup, automatic/manual switching, and similar functionalities inessential to list herein. Logic control is naturally specified as an FSM [9]. As for the process, this is typically described [10] by a DAE in the form

$$\begin{cases} F\left(\frac{dx_P(t)}{dt}, u(t), z(t)\right) &= 0 \\ G\left(x_P(t), u(t), z(t)\right) &= 0 \\ y(t) &= H\left(x_P(t), u(t), z(t)\right) \end{cases} \tag{3}$$

where x_P is the state vector and z a set of algebraic variables. Numerous techniques are available for simulating a declarative dynamic model composed of DAEs and ODEs; some are discussed e.g. in [10], while a declarative semantics for representing control-targeted FSMs is presented in [9] together with a possible imperative interpretation. In our example, the declarative representation of modulating and logic control is illustrated in Figure 2, while their imperative realisation corresponds to invoking periodically, at every control period T_s, the code in Algorithm 1.

```
while control_is_active do
    Modulating control ($x_C$ is the PI state);
    $u \leftarrow max\left(u_{min}, min\left(u_{max}, x_C + K(w - y)\right)\right)$;
    $x_C \leftarrow e^{-T_s/Ti} x_C + \left(1 - e^{-T_s/Ti}\right) u$;
    Logic control;
    if $w > 0 \land y \ge y_{thr}$ then $w \leftarrow -A_w$;
    if $w < 0 \land y \le y_{thr}$ then $w \leftarrow A_w$;
```

ALGORITHM 1: Pseudo-code for modulating and logic control in the CPS of Figure 1.

The simulation accuracy and performance depend considerably on the used control representation. To show this, with the entities just defined we construct three DTs:

- a fully declarative one, that we call "CT" (Continuous Time) as it is a continuous-time dynamic system;
- one that we call "CaA" (Control as Algorithms), where the process is modelled in the continuous time while all controls are described by algorithms;
- one that we call "LCaA" (Logic Control as Algorithms), where only logic controls are represented as algorithms.

We then simulate the three DTs with $T_s = 0.2$: the resulting behaviours of $y(t)$ and $u(t)$ are in Figure 3. Recall that the maximum-fidelity model is CaA (red), as in reality all controls are digital. Though plots almost overlap, we can observe that LCaA (blue) is a good approximation of CaA, while the relay toggling times in CT (green, see the u step-like variations in the bottom plot) diverge from CaA and LCaA as the simulation progresses, and consequently so do $y(t)$ and $u(t)$. This happens because as long as T_s is properly selected, modulating digital controls are well represented by their continuous-time counterparts, and an imprecise evaluation of when modulating control signal change their value has hardly any relevance.

A fundamental point to observe, however, is that the above is not the case with *logic* controls. In reality, logic control components change state at clock instants only. If their representations in a simulation model are conversely allowed to change state at any time, as happens here in the CT case, divergences as that just noticed are the inevitable consequence.

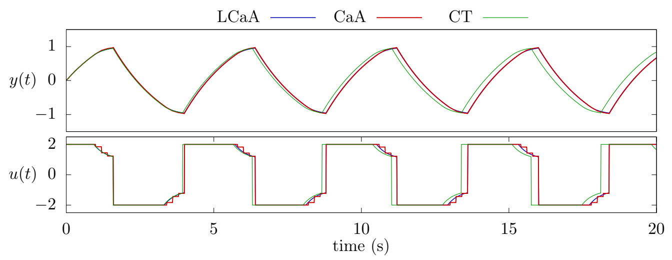


Fig. 3: CT and LCaA to approximate CaA.

	Number of steps			Normalised		
Tolerance	CT	CaA	LCaA	CT	CaA	LCaA
10^{-2}	254	2500	2750	1.00	1.00	1.00
10^{-4}	753	2977	3187	2.96	1.19	1.16
10^{-6}	1585	5496	6042	6.24	2.20	2.20
10^{-8}	2444	8853	9901	9.62	3.54	3.60
10^{-10}	4136	12810	14442	16.28	5.12	5.25
10^{-12}	7306	19491	20867	28.76	7.80	7.59

TABLE I: Simulation results – solver steps for the three DTs.

We now examine the computational burden entailed by the three DTs, looking at how many steps it takes to run the simulations in Figure 3 with the DASSL variable-step solver [11]. We chose to use DASSL throughout the paper for two reasons. First, despite its age, it is still a workhorse in dynamic simulation. Second, it is a general-purpose solver: comparisons made with it are not keen to be biased by the particular simulation problem one considers.

For simple models like this, the number of steps is more informative than the simulation time, as the latter is invariantly so small that, e.g., operating system artefacts have a visible influence; with more complex models, the effect pointed out below would also appear on the simulation time. To trade simulation speed versus accuracy, we sweep the solver relative tolerance from 10^{-2} down to 10^{-12}.

Table I shows the results. One can see that stopping the solver every T_s not only significantly increases the required steps, but also reduces the effectiveness of tolerance as an accuracy/speed trade-off knob. The DT that can be accelerated the most by acting on the tolerance is CT, but as we just saw, this is structurally and by far the least accurate.

In this paper, we propose an extension to EB-OOM languages and compilers to overcome the problem we just evidenced. We also demonstrate the proposal by turning it into a C++ library that enhances the Modelica EB-OOM language.

III. The involved technology – relevant aspects

To fully understand the reason for the performance problem just seen, we need to delve deeper into EB-OOM languages. Since this is easier to do with a specific language as an example, without loss of generality we focus on Modelica owing to its prominent role in DTs [12], [13] and to the availability of both open source and commercial implementations. For details on the language that cannot fit herein, see [14].

The main reason to choose EB-OOM is that, over the years, alternative approaches – most notably, block-oriented modelling – have proven inadequate for industry-size, large-scale, multi-domain cases [15]. Also, EB-OOM naturally lends itself to hosting equation- and algorithm-based models jointly [16], which is a fundamental enabler for our solution.

Furthermore, as our problem arises from logic controls, we need a formalism to describe them. For adherence to industry standards, here we stick to the languages defined in IEC 61131-3 [17], and specifically to Sequential Functional Chart (SFC). When referring to a host architecture is convenient, for the same reason we hereinafter talk about a Programmable Logic Controller (PLC).

It is important to stress that adhering to the IEC standards is not a limitation; on the contrary, it is a way to maximise the industrial applicability of our proposal. Though IEC takes the PLC as the reference architecture, our methodology applies to any continuous-time plant model joined to a digital control system made of processors that communicate over a network and cooperatively execute a control strategy, accessing both individual and shared resources, no matter how that control system is realised.

A further argument in IEC standards' favour comes from the discussion in [9], based on which the SFC language can be viewed as an imperative FSM descriptor independently of how the declarative FSM model is specified, hence not limiting the applicability of our proposal on this front either.

It is finally worth observing that owing to the way EB-OOM allows one to create, manage and run simulation models, the approach is already well accepted and is steadily gaining importance in the industry.

An extensive discussion about this relevant matter would be long and stray from the scope of this paper. However, we believe that a tour of the web site of the Modelica Association [18], looking in particular at the numerous applications presented and the large number of free and commercial libraries developed by many academic and industrial institutions, as well as a visit to the OpenModelica Consortium [19] page, can quite easily convince the reader.

A. Management and effect of events

Modelica models have an `equation` and an `algorithm` section. The first accepts DAE systems denoting time derivative with the `der` operator. Equation- and algorithm-based modelling are combined by introducing *events*. This is done in the `algorithm` section by `when` clauses. When the boolean expression in such a clause *becomes* true, variables declared `discrete` change as per the imperative code in the clause, preserving their value at any other time. There exist time events, where the condition depends only on time, and state events, where it can also depend on any variable in the model. A special case of time event is the `sample(t0,Ts)` clause, that triggers events periodically every T_s, the first one at t_0. For example, Listing 1 simulates the DAE (1) subjected to the sine input $u(t)$ as per line 10, together with the periodic sampling and zero-order holding of its output.

If a variable-step solver is employed, events influence its behaviour because integration must stop every time an event is triggered. We omit further details, yet it is clear that a high number of events slows down the simulation.

B. Connections and encapsulation

Modelica models are hierarchically composed of submodels (or components) that encapsulate their behaviour and the associated data within their interfaces, and are tied to one another by equations generated via `connect` clauses. This is illustrated in Listing 2, that refers to a hypothetical electric circuit model.

As such, connections are the only way for the components of a model to communicate – in the broadest sense – with one another. In the absence of these, focusing on logic control,

```
model DAE_sample_hold
  parameter Real T      = 1;
  parameter Real omega  = 1;
  parameter Real t_sh   = 0.1;
  parameter Real ystart = 0;
  Real u,y(start=ystart);
  discrete Real y_sh;
equation
  y + T*der(y) = u;                // DAE
  u            = sin(omega*time); // Exogenous input
algorithm
  when sample(0,t_sh) then
    y_sh := y;                     // Sample and hold
  end when;
end DAE_sample_hold;
```

Listing 1: Modelica model for the DAE (1).

```
connector Pin
  Voltage v;       // connects make these equal
  flow Current i; // flow -> connects make these sum to 0
end Pin;

model Resistor
  parameter Resistance R=1000;
  Pin a,b;
equation
  a.i+b.i = 0;
  a.v-b.v = R*a.i;
end Resistor

model Circuit
  ...                     // Declaration of components
  Resistor R1(R=100);
  Resistor R2(R=1500);
  Resistor R3;            // default, R=1kOhm
  ...
equation
  ...                     // Circuit connections
  connect(R1.a,R2.a);    // three pins connected:
  connect(R1.a,R3.a);    // all v's equal, i's sum to zero
  ...
end Circuit;
```

Listing 2: Usage of the **connect** statement.

1) there is no means for a model component to instruct others to trigger an event at some time point in the future (think of a sensor event to be caught at the next PLC cycle time),
2) and there is no means for a set of components to act on a same entity (anticipating the content of Section V-B later on, think of an SFC scheme where the same action is referred to by several steps).

As a result, implementing many constructs of IEC languages with EB-OOM ones is currently very cumbersome — a further EB-OOM limitation that we implicitly address in this paper. More in general, with EB-OOM it is difficult to realise fundamental logic primitives such as semaphores, mutexes and the like. A small example concerning the well-known "dining philosophers" problem is discussed in [20]: the reader can imagine the complexity of addressing a real-life case.

IV. Research question

Control engineers are used to designing controls in the continuous time domain, and then writing discrete-time algorithms in industry-standard languages to approximate their behaviour. Here we have a different, almost symmetric problem. First, we need to exploit continuous-time modelling to efficiently represent a ground truth part of which is physically digital (a couple of words that now should not appear as an oxymoron anymore). This exploitation is hindered by logic controls because, in their presence, preserving the time-quantised nature of control actions is essential (just associating a time delay to the firing of a transition, as done in the Modelica Standard Library StateGraph package [21], is not a solution). Second, we need to present to the analyst a control modelling interface as similar as possible to industrial (here, IEC-compliant) development environments. Doing so is hindered by the EB-OOM paradigm itself, as the abstraction of connectors, together with encapsulation, makes cross-component communication and data access cumbersome to represent.

We can therefore express the idea of "efficient control representation" more precisely than we did in the introductory part of the paper and formulate the research questions below.

Q1 Can we have an EB-OOM compiler preserve the alignment of logic events to sampling and actuation instants efficiently?

Q2 Can we allow analysts having experience with IEC and EB-OOM languages (but not with the corresponding tools' internals) to describe cross-component communication and synchronisation in a way that feels natural to them, while still obtaining efficient simulation code?

Answering these questions requires extending the semantics of EB-OOM languages and modifying compilers accordingly. In the following we present our solution, which consists of complementing the simulation code and data structures generated by an EB-OOM tool with an application-independent library of imperative code, plus the corresponding data structures, that take care of the logic part of a control scheme. Our solution can be employed *as is* with any Modelica tool, demonstrating the feasibility and efficiency of the idea.

As a result of this work and some discussions with the OpenModelica developers, that tool will soon incorporate part of our proposal. A complete implementation requires modifications to the compiler that are outside the scope of this paper, and will be addressed by new-generation EB-OOM language compilers such as the one presented in [22].

V. Our proposal

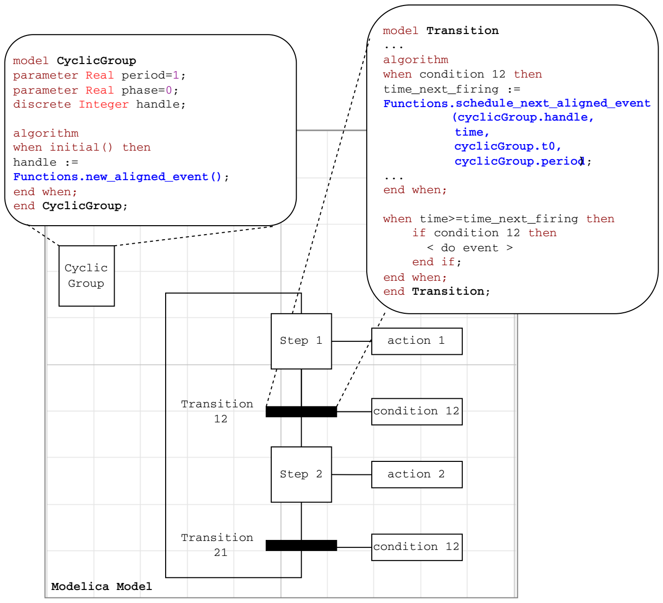
We now explain the proposal operation, articulated in an event time management (Section V-A), and an actions/connections management part, (Section V-B). Section V-A provides our answer to Q1, while Section V-B responds to Q2.

A. Efficient management of logic control events

We want to create an LCaA DT, with the logic control specified as SFC, where events occur *aligned* to a periodic clock, but only in the clock cycles when something happened that would cause some SFC transition(s) to fire. By skipping unnecessary events, we save on the computation time of algorithmic blocks and also allow variable step solvers to take longer integration steps. To this end, and to the benefit of modularity, we introduce the concept of *cyclic group* to denote a set of event-generating entities (in our case, SFC diagrams) that must be executed with the same sampling period T_s and with the first activation at the same instant t_0.

With our abstraction multiple cyclic groups can co-exist in the same simulation; this allows, for example, to simulate a set of distributed systems, each operating with its own clock of a different frequency and/or phase. We could also extend the idea to non-periodic event sources, but this will be the subject of future works.

Implementing this abstraction requires a mixed declarative/imperative solution, that we realise using Modelica and C++, as illustrated in Figure 4 and articulated below.


```
model CyclicGroup
parameter Real period=1;
parameter Real phase=0;
discrete Integer handle;

algorithm
when initial() then
handle :=
Functions.new_aligned_event();
end when;
end CyclicGroup;
```

```
model Transition
...
algorithm
when condition 12 then
time_next_firing :=
Functions.schedule_next_aligned_event
         (cyclicGroup.handle,
          time,
          cyclicGroup.t0,
          cyclicGroup.period);
...
end when;

when time>=time_next_firing then
    if condition 12 then
      < do event >
    end if;
end when;
end Transition;
```

Fig. 4: The proposed mixed declarative/imperative solution in Modelica; boxes show the Modelica wrappers for the C++ code and their usage.

- On the Modelica side, each cyclic group holds its period T_s and t_0, making it known to all the contained SFCs and thus to their transitions;
- at simulation startup, the cyclic group registers itself to the C++ side by calling a Modelica wrapper function, and receives a handle that is therefore known to all the contained transitions;
- in Modelica, when a transition detects that its firing condition has become true, it calls the C++ function $schedule\_next\_aligned\_event$ that executes Algorithm 2.
  - If the next firing time for the transition's cyclic group is in the past, that transition is the first to "book" a firing at the next clock cycle: the C++ side computes the next firing time as the next integer multiple of the cyclic group period and returns it to the transition.
  - If the next firing time is in the future, some transition has already booked firing for the cyclic group: the already calculated next firing time is returned to the transition. This ensures event synchronisation, and in the case of simultaneous firings also ensures that no short-time event hauls occur — which could conversely happen if each transition computed its next firing time individually, potentially obtaining slightly different values for numerical reasons.
- when the next firing time comes, the transition Modelica code checks that the firing condition is still true, and if so fires. This approach may trigger some unnecessary time events when glitches in the firing condition shorter than a clock period occur, but experiments show significant performance gains compared to firing events periodically.

```
**Function** $schedule\_next\_aligned\_event$
    **Input**: $handle, current\_time, t0, period$;
    **Output**: $next\_event\_time$;
    $next\_event\_time \leftarrow get\_saved\_time(handle)$;
    **if** $current\_time > next\_event\_time$ **then**
        $next\_event\_time \leftarrow$
          $ceil((current\_time - t0)/period) * period + t0$;
        $set\_saved\_time(handle, next\_event\_time)$;
```

ALGORITHM 2: Pseudo-code of the C++ algorithm to schedule the next event aligned to a sample period.

As said, we want the user to assemble an SFC model in Modelica graphically, with a look and feel as similar as possible to the typical IEC development environment. To this end, we first created the connectors in Listing 3.

```
connector StepInput
  input Boolean fire;
end StepInput;
connector StepOutput
  input Boolean fire;
  output Boolean active;
end StepOutput;
connector TransitionInput
  input Boolean active;
  output Boolean fire;
end TransitionInput;
connector TransitionOutput
  output Boolean fire;
end TransitionOutput;
```

Listing 3: The defined connectors.

The boolean **`active`** serves for steps to enable downstream transitions, while **`fire`** indicates an event by changing its logic state. The above given, the Modelica code for a transition is shown and commented in Listing 4. Analogously, the Modelica code for a step is in Listing 5.

As the reader may notice, the SFC evolution we realise is without stability search, i.e., when at a given control step the transitions that need firing have fired, the state of the SFC is not re-evaluated to identify possible further firings. Evolution algorithms with stability search, on the contrary, proceed to execute such further firings until a no-firings – i.e., "stable" – SFC state is reached. We motivate our choice first with strict adherence to the SFC standard, and then with the fact that the great majority of PLC tools stick to the same approach for safety. It is hard – if ever possible – to guarantee that an evolution algorithm with stability search eventually finds a no-firings condition, and it is even harder to provide any worst-case the computation time. In systems with real-time constraints as controls, the problems above are best avoided.

A possible all-Modelica implementation of our solution is shown in Listing 6. Incidentally we got in touch with the OpenModelica developers, and this is the way OpenModelica will soon interpret the semantics of the statement **`when sample(t0,Ts) and <condition>`**. The integration in Modelica of the cyclic group concept, that also provides modularity by preventing numerical issues arising out of event synchronisation, is on the roadmap.

```
model Transition
  SFC.Interfaces.TransitionInput IN;
  SFC.Interfaces.TransitionOutput OUT;
  BooleanInput C; // logic condition input
  // get cyclic group data from first CyclicGroup object
  // found by traversing the model hierarchy upwards
  outer SFC.SFCelements.CyclicGroup cyclicGroup;
  discrete Real time_next_firing;
equation
  OUT.fire = IN.fire;
algorithm
  // upstream step(s) active & condition true: book
     firing
  when pre(IN.active) and pre(C) then
    time_next_firing :=
      Functions.schedule_next_aligned_event
        (cyclicGroup.handle, time, cyclicGroup.t0,
         cyclicGroup.period);
  end when;
  // booked time reached & condition still true: fire
  when time>=time_next_firing then
    if C then OUT.fire := not OUT.fire; end if;
  end when;
initial algorithm
   // set time_next_firing surely in the past
   time_next_firing := -Modelica.Constants.inf;
   IN.fire          :=  false;
end Transition;
```

Listing 4: Modelica code for an SFC transition.

```
model Step
  SFC.Interfaces.StepInput IN;
  SFC.Interfaces.StepOutput OUT;
  Boolean active;
  Real t(start=0) "time since last activation";
  BooleanOutput X "true if step active";
  parameter Boolean initialStep = false;
  discrete Real dur_last_activity(start=0,fixed=true);
protected
  discrete Real t_last_activation(start=0,fixed=true);
equation
  active = X;
  t = if X then time - t_last_activation else 0;
  OUT.active=X;
algorithm
  when change(IN.fire) then // firing upstream
    X:=true;
    t_last_activation:= time;
  end when;
  when change(OUT.fire) then // firing downstream
     X:=false;
     dur_last_activity := t;
  end when;
initial algorithm
  X := initialStep;
end Step;
```

Listing 5: Modelica code for an SFC step.

B. *Managing actions and inter-component connections*

In SFC, each action has a *qualifier* that specifies the relationship between its execution and the activity of the step(s) to which it is connected. There are many qualifiers, but all of them can be obtained by combining the four fundamental qualifiers **N**, **S**, **R**, **P** and convenient temporal predicates based on the step activity times (variable **t** in Listing 5).

- **N** (Non-stored) makes the action active if the connected step is; the activity state of an action connected to several steps with the **N** qualifier only is the OR of the activity states (**X** in Listing 5) of those steps.
- **S** (Set) sets an action active, and the action stays active also after the citing step ceases to be.
- **R** (Reset) keeps the action inactive as long as the citing step is active, overriding all other qualifiers.

```
model Cycle_all_Modelica_example
  Real x(start=0,fixed=true);
  Real z(start=1,fixed=true);
  Real a(start=1,fixed=true);
equation
  // sawtooth x (slope pi, period 0.5) for testing
  der(x) = 0.1*Modelica.Constants.pi;
  when x>Modelica.Constants.pi/2 then
     reinit(x,0);
  end when;
  // fire a state event on integer multiple of 0.1
  // if x>1, and update the next multiple
  when time>pre(a)*0.1 and x>1 then
     a = integer(div(time, 0.1) + 1);
     if (a-1==pre(a) and x>1) then // event
        z = pre(z)+1; // increment z to signify
     else
        z = pre(z);
     end if;
  end when;
end Cycle_all_Modelica_example;
```

Listing 6: All-Modelica cycle time management.

- **P** (Pulse) only applies to actions for which the idea of "execute once" makes sense, such as incrementing a counter; the action is performed when the citing step becomes active.

Managing actions requires all the steps that influence the activity of an action to be connected to that action. This cannot be realised by Modelica **connect** statement, as doing so would enormously complicate Modelica diagrams, requiring connections that have nothing to do with the SFC syntax (observe instead Figure 4, where all the seen connections would also appear in an SFC tool). Hence, for this part of the proposal, the use of external imperative code and data structures is a necessity.

For **N**, **S** and **R** actions, the activation state is managed by the methods **on_step_activation** and **on_step_deactivation** of the C++ **action** class, reported in Algorithm 3. On the Modelica side, **N**, **S** and **R** actions extend the base class **Base_action_NSR** shown in Listing 7 and redeclare the qualifier to call the C++ action methods properly. For **P** actions things are a bit more complex, as in IEC tools, these are typically specified employing some 61131-3 language other than SFC, most often Ladder Diagram or Structured Text. It is not difficult to emulate Structured Text in Modelica, but the user has to enter textual code directly into his/her model, and for such activity, the ergonomics of Modelica and IEC tools are very different. At present we provide partial support for **P** actions in the form of "typical" ones, i.e. setting, incrementing or decrementing a numeric variable. For such **P** actions – the great majority, incidentally – we provided *ad hoc* library components. For the general case, to date the user has to enter algorithmic Modelica code: we shall address the problem in future works. For details on the presented library, as well as for a synthetic usage manual and examples, the reader is referred to the repository `https://github.com/looms-polimi/SFClib`.

To end this section, we spend a few words on the relevant aspect of whether and how the created DTs for CPSs can be subject to verification. In this respect we first notice that, as proven in the literature [23], [24], imperative control code realised in the SFC language can be subject to formal

```
Function on_step_activation
    Input: $qualifier$;
    Output: $action\_on$;
    if $qualifier = N$ then
        $active\_N\_phases \leftarrow active\_N\_phases + 1$;
    if $qualifier = S \wedge active\_R\_phases = 0$ then
        $was\_set \leftarrow TRUE$;
    if $qualifier = R \wedge active\_R\_phases = 0$ then
        $active\_R\_phases \leftarrow active\_R\_phases + 1$;
        $was\_set \leftarrow FALSE$;
    $action\_on \leftarrow active\_R\_phases =$
      $0 \wedge (active\_N\_phases > 0 \vee was\_set)$;
Function on_step_deactivation
    Input: $qualifier$;
    Output: $action\_on$;
    if $qualifier = N$ then
        $active\_N\_phases \leftarrow active\_N\_phases - 1$;
    if $qualifier = R \wedge active\_R\_phases = 0$ then
        $active\_R\_phases \leftarrow active\_R\_phases - 1$;
    $action\_on \leftarrow active\_R\_phases =$
      $0 \wedge (active\_N\_phases > 0 \vee was\_set)$;
```

ALGORITHM 3: Pseudo-code for the C++ methods to manage the state of N, S and R actions when steps become active or inactive.

```
partial model Base_action_NSR
  parameter String action_name = "action1";
  BooleanInput phase_active;
protected
  replaceable constant Integer qualifier;
  Integer handle_bool, handle_act;
  Boolean phase_active_neg;
equation
  phase_active_neg = not phase_active;
algorithm
  when initial() then
    handle_bool :=register_boolean_variable(action_name);
    handle_act := register_action("act_" + action_name);
    if phase_active then
       set_boolean_variable(handle_bool,
           on_phase_activation(handle_act, qualifier));
    end if;
  end when;
  when phase_active then
    set_boolean_variable(handle_bool,
        on_phase_activation(handle_act, qualifier));
  end when;
  when phase_active_neg then
    set_boolean_variable(handle_bool,
        on_phase_deactivation(handle_act, qualifier));
  end when;
end Base_action_NSR;
```

Listing 7: Modelica base class for N, S and R actions.

verification. The quoted reference is just an example; in the literature there are many that we do not review herein, as we only need to show that such a verification is viable. Also, EB-OOM tools structurally guarantee the correspondence between the equations in a model and the code produced by their translation [25], [15].

As a consequence of the above statements, for which there is vast support in the literature, representing SFC –– and prospectively any IEC language –– in EB-OOM paves the way to formal verification for the DT of a CPS (not for the contained control code alone, as per the present industrial state of the art). A few and quite preliminary attempts in this direction already exist, see e.g. [26], but a historical obstacle to such developments is computational burden — an issue that our proposal can help mitigating from the model side. All of this matter is extremely interesting and promising but apparently not within the scope of this paper; it will be considered as the subject of future research work.

As a final note, most Modelica tools permit to synchronise the execution of a model to external events. This is a useful feature if a DT must run in parallel with its PT, and since in that case the real time pace must be kept, it also provides another argument in favour of seeking computational efficiency like our proposal does.

VI. Application examples

In this section, we present two application examples built along the proposed approach and implemented with the developed Modelica library. The first one refers to a process application and aims to illustrate the operation of the used Modelica/C++ compound. The second one refers to a manufacturing context and aims to show the flexibility and scalability of the devised modelling solution.

A. Example 1

The system addressed in this example is depicted as P&ID – Piping & Instrumentation Diagram, see e.g. [27] for information about this widely adopted industry standard that we cannot include herein – in Figure 5.

The system is composed of a first tank, where a bulk component is kept at a prescribed temperature, and of a second tank, where an additive is mixed with the bulk; the obtained product is then unloaded. After each sixth operation, the second tank is cleaned by loading a dedicated fluid, activating the mixer, and flushing.

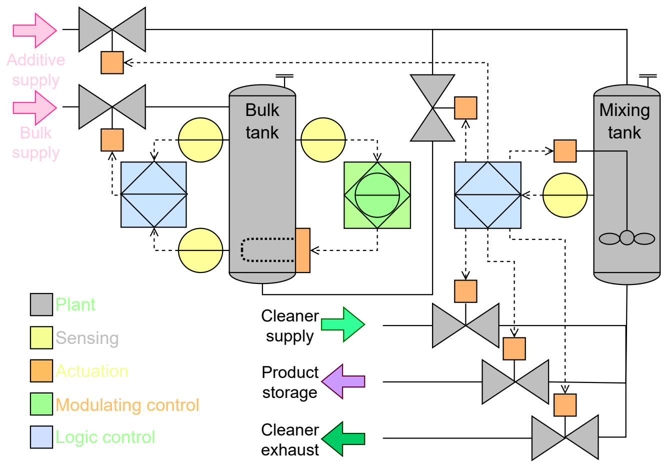


Fig. 5: Application example 1 – P&ID.

The level in the bulk tank is kept within a minimum and a maximum value by operating the on/off “bulk supply” valve, while a Proportional-Integral-Derivative (PID) controller acts on a heater to govern the temperature of the contained fluid.

Figure 6 reports all the Modelica diagrams of the system simulator. The top-level one is composed of Plant (P), Modulating Control (MC) and Logic Control (LC), respectively. The

P scheme does resemble the P&ID of Figure 5; the MC scheme holds the PID and its I/O blocks; the LC contains two schemes, one for controlling the bulk level and one for the recipe batch sequence. Observe the divergence/convergence structure to manage the periodic cleaning (the library obviously contains all the elements to represent such SFC constructs). As can be seen, Figure 6 closely recalls the typical IEC development environment despite containing declarative code.

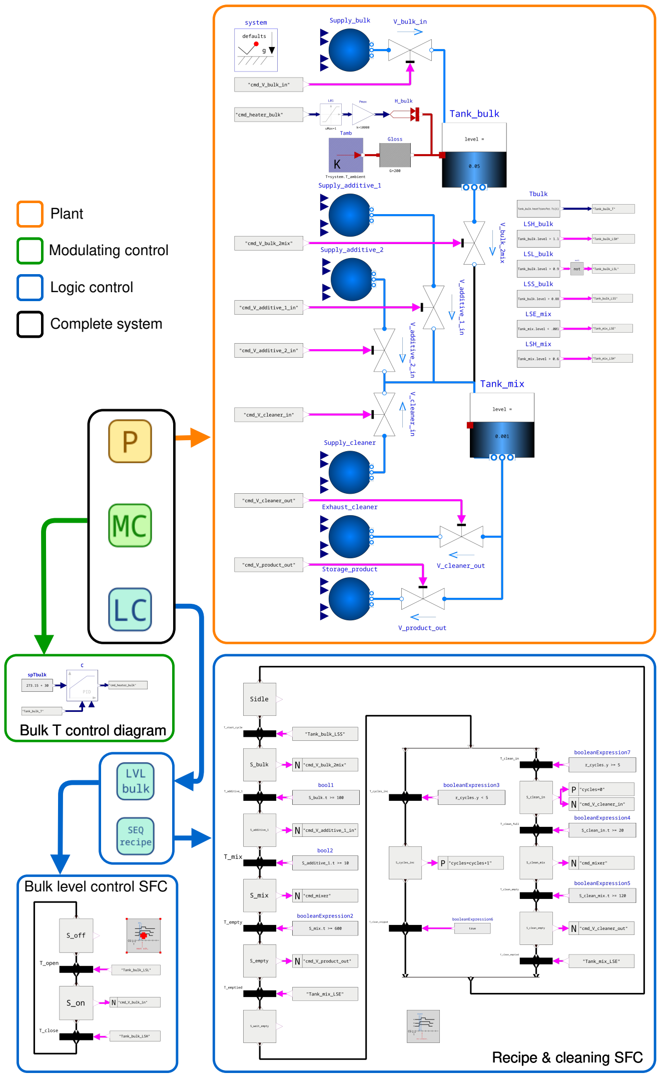


Fig. 6: Application example 1 – Modelica diagrams.

To assess the obtained efficiency, we simulated 4 hours of system operation with different cycle times for the two LC components (the two SFC diagrams in Figure 6). Table II shows simulation times and events, while Figure 7 reports the bulk and mixing tank levels with the effect of the periodic cleaning (top) and the bulk tank temperature with its set point (bottom).

As can be seen, the number of events is practically independent of the cycle times (that would trigger 28800 to 144000 events in the simulated time span) and is ruled (as desired) by the number of transition firings. Besides yielding good efficiency in general, this makes tolerance an effective knob

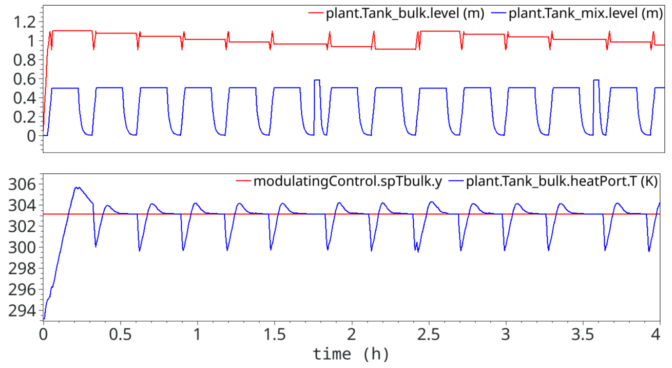


Fig. 7: Application example 1 – sample of simulation results (tank levels, top, and bulk temperature vs. set point, bottom).

Cycle times [s]			
Bulk level	Recipe sequence	Simulation time [s]	No. of events
2.00	0.50	1.581	268
2.00	0.25	1.635	263
2.00	0.10	1.648	263
0.50	0.50	1.840	269
0.50	0.25	1.912	269
0.50	0.10	1.932	269

TABLE II: Application example – simulation times and events.

for the accuracy/speed trade-off, despite logic controls.

B. Example 2

The second example we present refers to an assembly line, that is, to a very frequently encountered situation in manufacturing assets.

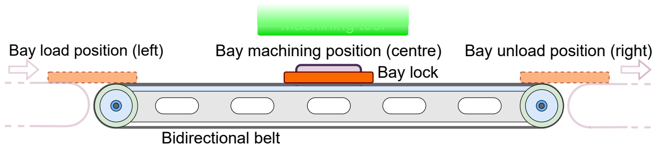


Fig. 8: Application example 2 – machining station.

The elementary component of the addressed system is a machining station like those available at the Industry 4.0 Laboratory laboratory at the Politecnico di Milano. A synthetic scheme for such a station is in Figure 8, and its operation can be summarised as follows.

- When started up, the station first moves its bay to the loading (left) side, and then starts waiting for a part to come from upstream.
- When a part appears at the load position, the station brings the bay to the machining (centre) position, locks it, and signals that machining can be carried out.
- When machining is over, the station unlocks the bay and moves it to the unloading (right) position, waiting for the machined part to be taken by the downstream part of the production line.
- Once the part is taken, the station brings the bay back to the loading (left) side and starts over waiting for a new part from upstream.

The bay position is controlled by a cascade structure with an inner velocity PI loop and an outer proportional position one. The Modelica model for a station, comprising the modulating block diagram and the SFC logic, is illustrated in Figure 9; the model also includes a random generator to pick a machining time in an assigned distribution.

Figure 10 shows an example plant model with 15 stations (plus split and join elements not described here for brevity) that totals 1739 equations. Figure 11 presents a sample of the obtained simulation results (bay position, top, and machining activity, bottom, at one of the machining stations).

Simulating 10 minutes of system operation took 24.7s, which is about 24.3× real time, notwithstanding the nontrivial size of the model. Most relevant, despite a quite significant presence of logic controls (18 SFC diagrams to govern 15 stations plus 2 split and 1 join element) and a quite fast sampling rate owing to the mechatronic nature of the system (all SFCs were made to run at a 10ms cycle time), only 1183 events were generated out of the possible 60000 — a saving of 98%. Needless to say, also in this case the cycle time used for the logic control part has hardly any influence on the simulation time.

VII. Related work

Besides being extensively applied as a fundamental pillar of Industry 4.0, the concept of DT has received enormous attention in the scientific literature; we briefly discuss herein just a few samples of so vast a *corpus*, to evidence and collocate the contribution of our research. According to the recent survey reported in [28], about 1000 papers per year on DTs were published between 2019 and 2021, for a total of 2934. The analysis in the paper just quoted focuses first on application field, hierarchy, discipline, dimension, universality and functionality, and then considers four model dimensions (geometry, physics, behaviour and rule).

The ultimate goal of DT taxonomy-oriented works, like the one just mentioned, is to deconstruct and investigate the idea and the process of DT modelling under several viewpoints, related to the nature and the intended use of the DT. In the quoted reference, for example, the said viewpoints are six: model construction, assembly, fusion, verification, modification, and management. However, such analysis works appear to not set a strong enough focus on the intrinsically Cyber-Physical nature of engineering systems — not even when discussing the model behaviour dimension, which is inherently dynamic, especially at the "system" and "system of systems" (SoS) levels [29]. It is interesting to observe, in addition, that surveys like [28] also highlight that to date no DT tool fosters the integration of the devised dimensions, in accordance with the idea of "DT multiverse" evidenced in [12] together with the advantages that would come from harmonising the various DT interpretations.

In the opinion of the authors, therefore, to date the literature – and even more the applications – somehow underestimate the need for integrating the various DT concepts available, and in particular the contribution that an EB-OOM approach can give in the direction just evidenced; as a consequence of the above remark, this paper intends to show precisely how the EB-OOM approach can be exploited to achieve a straightforward and computationally efficient model fusion for the description of the dynamic behaviour of a CPS.

As a support for the last statement, recent trends in CPS simulation evidence the need for scalability [30], not only in the model size but also concerning the level of detail [31], of capability in addressing large-scale systems [32], complex and distributed-parameters dynamics [33], and of integration/-communication with control, for both design [2], [34] and checking purposes [35], [36]. Moreover, in modern industrial CPSs, also correct use of control architectures is becoming crucial [37], making their choice itself a model-based problem, where DT solutions based on co-simulation can be time-critical [38]. It is finally important to remark that even if one strictly assumes that a DT needs real-time connection with its physical twin (thereby excluding its use for design, incidentally), exploiting EB-OOM for simulation efficiency still yields benefits. To mention just a notable example, it facilitates what-if analysis [39], both in general and in the CPS case [40] that we address.

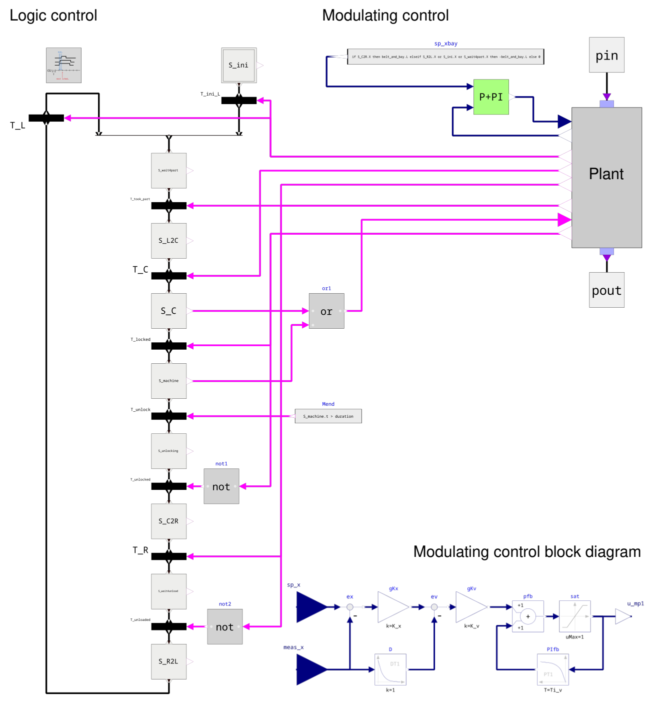


Fig. 9: Application example 2 – Modelica model for a machining station with control; the pin and pout connectors represent the part inlet (bay load) and outlet (bay unload).

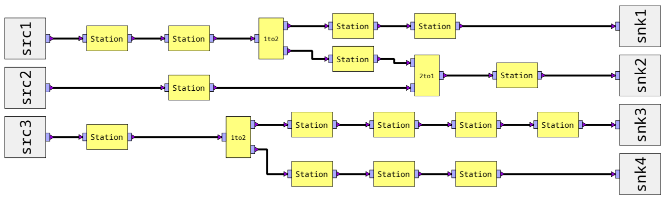


Fig. 10: Application example 2 – 15-station plant model.

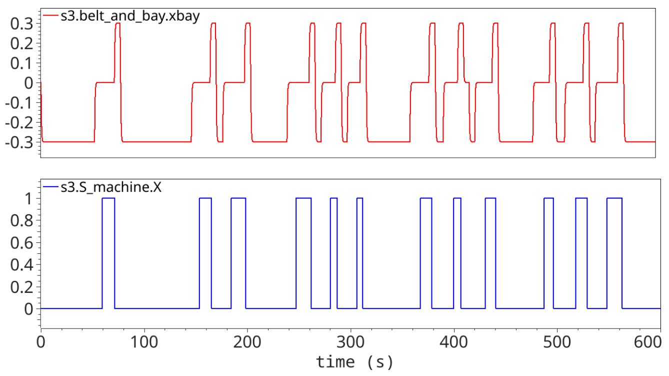


Fig. 11: Application example 2 – sample of simulation results (bay position and machining activity at station 3).

In such a *scenario*, summing up, the multi-domain [41] and declarative-imperative [12] capabilities of EB-OOM are gaining importance. However, as for CPS simulation, to date the developments in EB-OOM tools tend to focus either on integration with external applications [42] or on clock-based modelling [43], incurring costs that we proved relevant in terms of computational efficiency, and possibly also impacting the choice of numerical solvers [44]. As such, the proposal we formulate in this paper does fill a performance gap relative to problems of undoubted engineering interest.

VIII. Conclusions and future research

We started by arguing that when creating simulation-centric DTs in the Industry 4.0 context, the presence of digital controls results in a harsh precision/performance trade-off. We also argued that an extension to declarative (EB-OOM) languages and compilers toward hosting imperative constructs in a user-transparent manner could mitigate the problem. After presenting our solution, we can say that our conjectures were correct. As a result, we can offer to the community a mixed imperative/declarative modelling paradigm, and a way to realise it in the form of a mixed-language library. In that library – that we are releasing as free software within a 3-clause BSD licence – we used Modelica and C++, but the underlying ideas are evidently general. We believe that making the creation of (logic) control models in declarative (EB-OOM) environments look similar to developing the same controls with industrial (IEC) tools strongly eases the work of analysts with control and simulation competence. We also hope that our proposal can help widen the set of people with the said joint competence beyond the domains where control is so mission-critical to make it a necessity — a cultural challenge already undertaken and discussed by the Model-Based Systems Engineering community, see e.g. [45], [46].

Future work will aim to extend the control representation capabilities of our paradigm, addressing in the first place the other IEC 61131 languages. We also plan to extend our event handling core so as to model clock nonidealities like skew and jitter, as well as to support non-periodic system, thereby extending the coverage of our proposal to the wider IEC 61499 context. Research is finally underway to integrate our proposal into new-generation Modelica compilers.

Acknowledgment

The authors would like to thank prof. Francesco Casella (Politecnico di Milano) for providing insightful comments, and prof. Bernhard Bachmann (Fachhochschule Bielefeld) for inspiring the all-Modelica cycle time solution.

References

[1] G. Aceto, V. Persico, and A. Pescapé, “A survey on information and communication technologies for Industry 4.0: state-of-the-art, taxonomies, perspectives, and challenges,” *IEEE Communications Surveys & Tutorials*, vol. 21, no. 4, pp. 3467–3501, 2019.

[2] Q. Liu, H. Zhang, J. Leng, and X. Chen, “Digital Twin-driven rapid individualised designing of automated flow-shop manufacturing system,” *International Journal of Production Research*, vol. 57, no. 12, pp. 3903–3919, 2019.

[3] F. Pires, A. Cachada, J. Barbosa, A. Moreira, and P. Leitão, “Digital Twin in Industry 4.0: technologies, applications and challenges,” in *Proc. 17th IEEE International Conference on Industrial Informatics*, Helsinki, Finland, 2019, pp. 721–726.

[4] E. Negri, L. Fumagalli, and M. Macchi, “A review of the roles of Digital Twin in CPS-based production systems,” *Procedia Manufacturing*, vol. 11, pp. 939–948, 2017.

[5] W. Kritzinger, M. Karner, G. Traar, J. Henjes, and W. Sihn, “Digital Twin in manufacturing: a categorical literature review and classification,” *IFAC-PapersOnLine*, vol. 51, no. 11, pp. 1016–1022, 2018.

[6] M. Sjarov, T. Lechler, J. Fuchs, M. Brossog, A. Selmaier, F. Faltus, T. Donhauser, and J. Franke, “The Digital Twin concept in industry – a review and systematization,” in *Proc. 25th IEEE International Conference on Emerging Technologies and Factory Automation*, Vienna, Austria, 2020, pp. 1789–1796.

[7] F. Ocker, C. Urban, B. Vogel-Heuser, and C. Diedrich, “Leveraging the asset administration shell for agent-based production systems,” *IFAC-PapersOnLine*, vol. 54, no. 1, pp. 837–844, 2021.

[8] F. Ocker, B. Vogel-Heuser, H. Schon, and R. Mieth, “Leveraging Digital Twins for compatibility checks in production systems engineering,” in *Proc. 28th IEEE International Conference on Industrial Engineering and Engineering Management*, Singapore, 2021, pp. 103–107.

[9] J. Provost, J. Roussel, and J. Faure, “A formal semantics for Grafcet specifications,” in *Proc. 7th IEEE International Conference on Automation Science and Engineering*, Trieste, Italy, 2011, pp. 488–494.

[10] F. Cellier and E. Kofman, *Continuous system simulation*. Heidelberg, Germany: Springer Science & Business Media, 2006.

[11] L. Petzold, “Description of DASSL: a differential/algebraic system solver,” Sandia National Laboratories, Livermore, CA, USA, Tech. Rep., 1982.

[12] C. Cimino, G. Ferretti, and A. Leva, “Harmonising and integrating the Digital Twins multiverse: a paradigm and a toolset proposal,” *Computers in Industry*, vol. 132, pp. 103 501:1–103 501:11, 2021.

[13] Y. Wu, K. Zhang, and Y. Zhang, “Digital twin networks: a survey,” *IEEE Internet of Things Journal*, vol. 8, no. 18, pp. 13 789–13 804, 2021.

[14] P. Fritzson, *Introduction to modeling and simulation of technical and physical systems with Modelica*. Hoboken, NJ, USA: John Wiley & Sons, 2011.

[15] S. Mattsson, H. Elmqvist, and M. Otter, “Physical system modeling with Modelica,” *Control Engineering Practice*, vol. 6, no. 4, pp. 501–510, 1998.

[16] H. Lundvall and P. Fritzson, “Event handling in the OpenModelica compiler and run-time system,” in *Proc. 46th Conference on Simulation and Modelling of the Scandinavian Simulation Society*, Trondheim, Norway, 2005.

[17] International Electrotechnical Commission, “IEC 61131-3 Programmable controllers – part 3: programming languages, edition 3.0,” 2013.

[18] Modelica Association home page, https://modelica.org/.

[19] OpenModelica Consortium home page, https://openmodelica.org/.

[20] H. Lundvall and P. Fritzson, “Modelling concurrent activities and resource sharing in Modelica,” in *Proc. 44th Scandinavian Conference on Simulation and Modeling*, Västerås, Sweden, 2003.

[21] M. Otter, K. Årzén, and I. Dressler, “StateGraph–a Modelica library for hierarchical state machines,” in *Proc. 4th international Modelica conference*, Hamburg, Germany, 2005, pp. 569–578.

[22] G. Agosta, E. Baldino, F. Casella, S. Cherubin, A. Leva, and F. Terraneo, "Towards a high-performance Modelica compiler," in *Proc. 13th International Modelica Conference*, Regensburg, Germany, 2019, pp. 313–320.

[23] K. Fujino, K. Imafuku, Y. Yuh, and N. N. Hirokazu, "Design and verification of the SFC program for sequential control," *Computers & Chemical Engineering*, vol. 24, no. 2-7, pp. 303–308, 2000.

[24] S. Shah, E. Endsley, M. Lucas, and D. Tilbury, "Reconfigurable logic control using modular FSMs: Design, verification, implementation, and integrated error handling," in *Proc. 2002 American Control Conference*, Anchorage, AK, USA, 2002, pp. 4153–4158.

[25] D. Kågedal and P. Fritzson, "Generating a Modelica compiler from natural semantics specifications," in *Summer Computer Simulation Conference*, Reno, NV, USA, 1998, pp. 299–307.

[26] H. Lundvall, P. Bunus, and P. Fritzson, "Towards automatic generation of model checkable code from Modelica," in *Proc. 45th Conference on Simulation and Modelling of the Scandinavian Simulation Society*, Copenhagen, Denmark, 2004, pp. 23–24.

[27] M. Toghraei, *Piping and Instrumentation Diagram development*. Hoboken, NJ, USA: John Wiley & Sons, 2019.

[28] F. Tao, B. Xiao, Q. Qi, J. Cheng, and P. Ji, "Digital Twin modeling," *Journal of Manufacturing Systems*, vol. 64, pp. 372–389, 2022.

[29] F. Tao, W. Liu, M. Zhang, T. Hu, Q. Qi, H. Zhang, F. Sui, T. Wang, H. Xu, Z. Huang *et al.*, "Five-dimension Digital Twin model and its ten applications," *Computer Integrated Manufacturing Systems*, vol. 25, no. 1, pp. 1–18, 2019.

[30] C. Cimino, G. Ferretti, and A. Leva, "The role of dynamics in digital twins and its problem-tailored representation," *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 10 556–10 561, 2020.

[31] T. Broenink and J. Broenink, "A variable detail model simulation methodology for cyber-physical systems," in *Proc. 32nd European Conference on Modelling and Simulation*, Wilhelmshaven, Germany, 2018, pp. 219–225.

[32] M. Andreev, A. Gusev, N. Ruban, A. Suvorov, R. Ufa, A. Askarov, J. Bemš, and T. T. Králík, "Hybrid real-time simulator of large-scale power systems," *IEEE Transactions on Power Systems*, vol. 34, no. 2, pp. 1404–1415, 2018.

[33] B. Scaglioni and G. Ferretti, "Towards Digital Twins through object-oriented modelling: a machine tool case study," *IFAC-PapersOnLine*, vol. 51, no. 2, pp. 613–618, 2018.

[34] G. Wan and P. Zeng, "Codesign of architecture, control, and scheduling of modular cyber-physical production systems for design space exploration," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 4, pp. 2287–2296, 2022.

[35] H. Carlsson, B. Svensson, F. Danielsson, and B. Lennartson, "Methods for reliable simulation-based PLC code verification," *IEEE Transactions on Industrial Informatics*, vol. 8, no. 2, pp. 267–278, 2012.

[36] B. Fernández Adiego, D. Darvas, E. Blanco Viñuela, J. Tournier, S. Bliudze, J. Blech, and V. González Suárez, "Applying model checking to industrial-sized PLC programs," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 6, pp. 1400–1410, 2015.

[37] M. Sehr, M. Lohstroh, M. Weber, I. Ugalde, M. Witte, J. Neidi, S. Hoeme, M. Niknami, and E. Lee, "Programmable logic controllers in the context of Industry 4.0," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 5, pp. 3523–3533, 2020.

[38] G. Schweiger, C. Gomes, G. Engel, I. Hafner, J. Schoeggl, A. Posch, and T. Nouidui, "An empirical survey on co-simulation: promising standards, challenges and research needs," *Simulation Modelling Practice and Theory*, vol. 95, pp. 148–163, 2019.

[39] F. Pires, B. Ahmad, A. Moreira, and P. Leitão, "Recommendation system using reinforcement learning for what-if simulation in Digital Twin," in *Proc. 19th IEEE International Conference on Industrial Informatics*, Palma de Mallorca, Spain, 2021, pp. 1–6.

[40] J. Lee, B. Bagheri, and H. Kao, "A cyber-physical systems architecture for Industry 4.0-based manufacturing systems," *Manufacturing Letters*, vol. 3, pp. 18–23, 2015.

[41] P. Fritzson, "Modelica: equation-based, object-oriented modelling of physical systems," in *Foundations of Multi-Paradigm Modelling for Cyber-Physical Systems*, H. V. P. Carreira, V. Amaral, Ed. Heidelberg, Germany: Springer, 2020, pp. 45–96.

[42] L. Hatledal, A. Styve, G. Hovland, and H. Zhang, "A language and platform independent co-simulation framework based on the functional mock-up interface," *IEEE Access*, vol. 7, pp. 109 328–109 339, 2019.

[43] C. Chen, H. Cao, S. Su, H. Chen, Y. Gong, and G. Chen, "A unified modelling method for cyber-physical systems based on Modelica," *International Journal of Wireless and Mobile Computing*, vol. 16, no. 4, pp. 350–357, 2019.

[44] E. Kofman, F. Cellier, and G. Migoni, "Continuous system simulation and control," in *Discrete-Event Modeling and Simulation*, G. Wainer and P. Mosterman, Eds. Boca Raton, FL, USA: CRC Press, 2018, pp. 75–107.

[45] S. Liscouet-Hanke, H. Jahanara, and J. Bauduin, "A Model-Based Systems Engineering approach for the efficient specification of test rig architectures for flight control computers," *IEEE Systems Journal*, vol. 14, no. 4, pp. 5441–5450, 2020.

[46] X. Zhang, B. Wu, X. Zhang, J. Duan, C. Wan, and Y. Hu, "An effective MBSE approach for constructing industrial robot digital twin system," *Robotics and Computer-Integrated Manufacturing*, vol. 80, p. 102455, 2023.