Discrete Event Dynamic Systems (2024) 34:95–124
https://doi.org/10.1007/s10626-023-00390-y

MANUSCRIPT

πHyFlow: formalism, semantics, and applications

Fernando Barros[1]

Received: 21 July 2022 / Accepted: 6 December 2023 / Published online: 24 January 2024

Abstract
Simulation models have been described using different perspectives, or worldviews. In the process interaction world view (PI), every entity is modeled by a sequence of actions describing its life cycle, offering a comprehensive model that groups the events involving each entity. In this paper we describe πHyFlow, a formalism for representing hybrid models using a set of communicating processes. This set is dynamic, enabling processes to be created and destroyed at runtime. Processes are encapsulated into πHyFlow base models and communicate through shared memory. πHyFlow, however, can guarantee modularity by enforcing that models can only communicate by input and output interfaces. πHyFlow extends current PI approaches by providing support for HyFlow concepts of sampling and dense (continuous) outputs, in addition to the more traditional event-based communication. Likewise HyFlow, πHyFlow is a modeling & simulation formalism driven by expressiveness and performance analysis. We present πHyFlow semantics, and several applications to illustrate πHyFlow ability to describe a diversity of systems.

Keywords Modeling & simulation · Process interaction worldview · Hybrid models · Operational semantics · Co-simulation · Performance analysis

1 Introduction

The *process interaction worldview* (PI) enables a simple and intuitive description of simulation models. Contrarily to the event interaction approach that offers an unstructured perspective of the systems based on a set of events, commonly represented by event graphs (Schruben 1983), the PI organizes events by entity and by their order of occurrence. The result is a script that is easier to understand and verify than the corresponding event graphs. The PI has its origins on the SIMULA language (Dahl et al. 1966). Some simulation languages supporting PI favors the active client approach (Henriksen 1981). In this view, transitory entities, like clients, are represented by processes, while permanence entities, like servers, are represented as data structures. In the alternative active server, processes model the permanent resources of the systems, like machines, while clients are viewed as passive data that is passed among processes. This view is often considered a requirement for a modular

✉ Fernando Barros
barros@dei.uc.pt

[1] Department of Informatics Engineering, University of Coimbra, Coimbra, Portugal

representation of systems, and it is supported by formalisms like DEVS (Zeigler 1976) for describing discrete event systems, and HyFlow (Barros 2017), to represent hybrid systems. Although modularity leverages hierarchical models, and the ability to represent complex models by a composition of simpler ones, it does not always provide the most adequate level of representation for simple models. In fact, base models in these types of modular representations can only describe one event, forcing, in general, that models with two or more events to be represented as a composition of several base models, one for each event. Considering, as an example, a system with one queue feeding two servers; a single event model needs to decompose the system into three models and describe the synchronization among entities. On the contrary, in PI, the communication can be achieved, in a simple way, through shared memory data structures since simulation processes have non-preemptive semantics. However, simulation languages supporting PI often do not enforce modularity, making it difficult to represent complex systems.

In this paper we present a new formalism to represent hierarchical, modular hybrid models that enables, at the base level, the ability to represent several events, keeping the advantages of PI. Our goal is to combine the simplicity of PI for describing small systems, with hierarchical, and modular constructs that enable a systems-of-systems representation for revealing system behavior and components' interactions (Nielson et al. 2015). Modularity also makes easier to represent systems with a dynamic topology, by providing a one-to-one mapping between changes in system topology and the corresponding structural adaptation of the model. We also show that the ability to create processes in runtime enables PI to support the active client approach, while guaranteeing modular and hierarchical models.

In previous work, we have developed the Hybrid Flow System Specification (HyFlow), a formalism aimed to represent hierarchical and modular hybrid systems (Barros 2017). HyFlow defines sampling and the exact representation of continuous signals as first-order constructs, enabling a simple specification of pull-communication in addition to push-communication, typical of discrete event systems. HyFlow models exhibit a dynamic topology, making it possible to make arbitrary changes in model composition and coupling. In this paper we develop the πHyFlow formalism, an extension of the HyFlow formalism, to represent base hybrid models using the process interaction worldview. πHyFlow can describe simulation processes that can communicate through both sampling and discrete events. The operational semantics of the πHyFlow formalism is also provided.

This paper is organized as follows. Section 2 introduces the πHyFlow formalism, and the operational semantics of πHyFlow base and network models. Section 3 describes some πHyFlow models of continuous, discrete and hybrid systems. Section 4 gives a brief description of πHyFlow++, a C++ implementation of πHyFlow. Related work is discussed in Section 5. Conclusion is given in Section 6.

2 The πHyFlow formalism

πHyFlow is a modeling and simulation (M&S) formalism aimed at performance evaluation. It supports dense outputs, generalized sampling, discrete events, hierarchical and modular models, and dynamic topology networks. πHyFlow extends the base HyFlow formalism by enabling base models to be described using M&S processes (or coroutines in computer science terminology). M&S processes enable an easy specification of models as a set of concurrent scripts defining the life-cycle of the entities describing the system under study. The simplicity is achieved at the implementation level where processes can be mapped into coroutines that

implicitly define an underlying state machine, freeing modeler from describing states and state transitions. Processes coordinate through the shared memory defined by the parent base model. Before presenting πHyFlow formal definition, and operational semantics, we provide first an informal overview in the next section.

2.1 πHyFlow overview

πHyFlow defines two types of models: base and network. The former supports a set of processes that interact through shared state variables. The communication between base models is made by a modular interface that includes support for sampling, continuous/dense flows, and discrete flows (events). Figure 1 provides an overview of a base model with modular input (X) and output (Y) interfaces, and a set of processes π_1, ..., π_n. The function ζ handles external messages arriving at the base model. The shared (partial) state (p-state) p is used to enable process interaction. Base model output $\{\Lambda_p\}$ is computed from the outputs of all processes. The set of processes is dynamic, being possible to create or destroy processes at runtime.

As a motivation example, we analyze a variation of the classical dining philosophers problem (Peterson 1981). We consider a dining room with a round table with N forks and N seats. Philosophers are allowed to enter/leave the room, but they also can renege service if they wait too long in line for a seat. The base model defines two shared variables: a sequence of $forks \in \{(f_1, \ldots, f_N) | \ f_i \in \{\bot, \top\}\}$, and a set of unoccupied *seats*. All forks and seats are available at simulation begin. A philosopher process can be informally described by Pseudo-code 1.

A philosopher has an id (line 1), and upon arrival (s)he requests a seat with the time duration limit $line$ (line 2). If a seat does not become available (line 3), (s)he reneges service through the output port "renege" (line 4) and finishes (line 5). Otherwise, the philosopher gets a (random) available seat (line 7, where "$\leftarrow$" is the assignment operator) and identifies the left and right forks (lines 8-9). The philosopher repeats k times a sequence of grabbing forks (lines 11-12), eating (line 13), releasing forks (line 14), and thinking (line 15). After this cycle, the philosopher releases the seat (line 17), and exits the dining room through the output port "exit" (line 18), and finishes (line 19).

Philosophers are created by the "door" process defined in Pseudo-code 2. We consider that the base model defines an additional sequence of integers stored in variable $buffers$[in] that holds the philosophers' ids arriving at input port "in". This buffer is handled by the input function ζ, not detailed here. The "door" loops forever (lines 2-6) executing the following

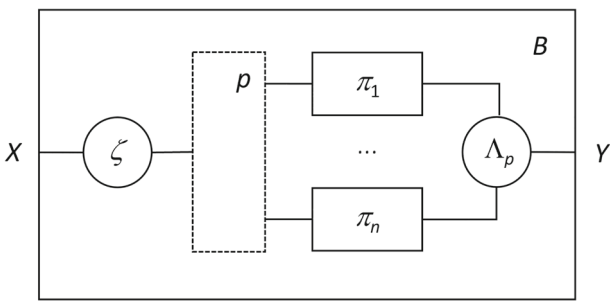

Fig. 1 Base model internal structure

Pseudo-code 1: Philosopher's process.

```
    philosopher(id)
2:      wait-duration-or-until (line, |seats| > 0)
        if (|seats| = 0)
4:          out (renege, id)
            end-philosopher
6:      end-if
        seat ← seats.pop()
8:      l ← left(seat)
        r ← right(seat)
10:      repeat (k)
        wait-until (forks[l] ∧ forks[r])
12:      forks[l] ← forks[r] ← ⊥
        wait-duration (eat)
14:      forks[l] ← forks[r] ← ⊤
        wait-duration (think)
16:      end-repeat
        seats.push(seat)
18:      out (exit, id)
    end-philosopher
```

sequence: i) wait until there is a waiting philosopher (line 3); ii) remove a philosopher's id from $buffers$[in] (line 4); iii) create a philosopher to handle the id (line 5). πHyFlow models can also describe continuous flows, enabling the representation of hybrid systems, as shown in the next sections.

Pseudo-code 2: Door's process.

```
    door()
2:      while (true)
            wait-until(|buffers[in]| > 0)
4:          id ← buffers(in).remove()
            create philosopher(id)
6:      end-while
    end-door
```

The advantages of πHyFlow base models over HyFlow ones are significant. The shared memory serves as in input buffer freeing processes from being automatically preempted. It enables, for example, processes to specialize on a particular set of inputs. In the example above, only the process "door", defined in Pseudo-code 2, is preempted when a new philosopher "id" arrives, freeing the other processes from reacting to this event. In general, πHyFlow processes can specialize in any set of pre/post/preemption conditions, while the corresponding HyFlow base model needs to handle all conditions simultaneously, making it more complex. Processes also implicitly define a state/phase change, given their ability to hold on a particular statement/index. The index being an abstraction of the coroutine line/program count. The modeler does not need to define process state/phase machine since the concept of state/phase is implicit in the sequential nature of the process description.

Formalism operational semantics is provided through the concepts of component and simulator. A component is associated with a base model, interprets model definition, and performs model implicitly-defined time-behavior. Similarly, the semantics of a process model is defined by a simulator that maps model definition into a coroutine-like behavior and performs model simulation.

A network/executive component is associated with a network/executive model and performs the simulation according to the corresponding model. Network and executive components are responsible for interpreting network models, including, how samples and events are exchanged between components, and how network topology can be adjusted during simulation run. Since base and networks components have the same interface, they can be seamlessly integrated, enabling the definition of hierarchical components.

Figure 2 depicts the relationship between (M)odels, (C)omponents and (S)imulators. The network component "N" is associated with model "M_1", and controlled by the executive component "E" associated with executive model "$M_{\eta 1}$". Component "E" has currently simulators "A" and "B" that share the same process model "M_2". "A" and "B" have access to the shared state defined by parent component "E" and, in addition, they also have their own private state.

Besides "E", "N" is currently composed by base components "X" and "Y", associated with model "M_3". Different components can share the same model. Component state, however, is not shared among components, being exclusively managed locally by each component, and by their children's simulators. Figure 2 also depicts a relevant property of the πHyFlow formalism, its ability to keep modular components during simulation execution. As pointed above, this property enables co-simulation, and it also provides support for dynamic topologies.

The next subsections define the semantics of the πHyFlow formalism that is targeted to provide a simple definition of simulation models. Base models are defined in Section 2.2. Process models are defined in Section 2.3. Base components are defined in Section 2.4. Process simulators are described in Section 2.5. The network model, and the executive model are described in Section 2.6. The executive component is defined in Section 2.7. The network component is defined in Section 2.8. The component simulation algorithm that is responsible to manage the global time, and drive the simulation is described in Section 2.9.

2.2 πHyFlow base model

A πHyFlow base model defines a modular entity enclosing a set of processes that communicate through a shared p-state. Each process keeps its own (private) p-state and can perform read/write operations on the shared p-state. Processes are non-preemptive making them implementable by coroutines. While threads have long been available in most programming languages, the native support for coroutines in the C++ high performance language is recent, being introduced by C++20 (ISO/IEC 14882 standard). Formally, a πHyFlow base model associated with name B is defined by:

$$M_B = (X, Y, P, P_0, \zeta, \Pi, \pi, \sigma, \{\Lambda_p\}),$$

where:

$X = X^{\mathrm{c}} \times X^{\mathrm{d}}$ is the set of input flow values, with
  X^{c} is the set of continuous input flow values,
  X^{d} is the set of discrete input flow values, and
  $X^{\varnothing} = X^{\mathrm{c}} \times X^{\mathrm{d}} + \varnothing$),
$Y = Y^{\mathrm{c}} \times Y^{\mathrm{d}}$ is the set of output flow values, with
  Y^{c} is the set of continuous output flow values,
  Y^{d} is the set of discrete output flow values, and
  $Y^{\varnothing} = Y^{\mathrm{c}} \times (Y^{\mathrm{d}} + \varnothing)$,
P is the set of partial shared states (p-states),

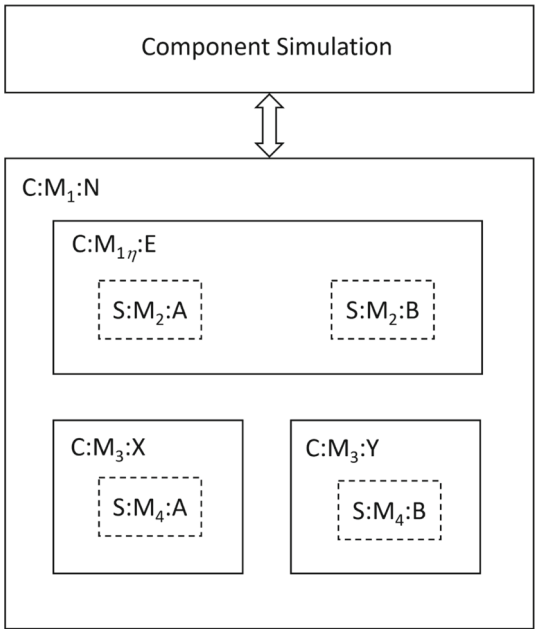

Fig. 2 Internal structure for simulating network component "N"

P_0 is the set of (valid) initial p-states,
$\zeta : P \times X^{\varnothing} \longrightarrow P$ is the input function,
Π is a set of processes names,
$\pi : P \longrightarrow \mathcal{P}(\Pi)$ is the current-processes function, where $\mathcal{P}$ is the power set operator,
$\sigma : \mathcal{P}\Pi \longrightarrow \mathcal{P}^*(\Pi)$ is the ranking function, where $\mathcal{P}^*(\Pi)$ is the set of all sequences based on set Π, constrained to: $\sigma(C | C \subseteq \Pi) = (c_1, \ldots, c_n) \Rightarrow \{c_1, \ldots, c_n\} = C \wedge |\sigma(C)| = |C|$,
for all p ∈ P:
$\Lambda_p : \underset{i \in (\sigma \circ \pi)(p)}{\times} Y_i^{\varnothing} \longrightarrow Y^{\varnothing}$ is the output function associated with p-state p.

Since processes have no entry points, the representation of model input needs to be stored in the shared p-state so it can be accessed by the processes. The input function ζ is responsible for updating the current p-state when the model receives a value either through sampling or event communication. The set of processes is dynamic, being the current set given by function π. Given processes can access the shared p-state only one can be active at any time. The ranking function σ decides process resume order. The output function $\lambda = \{\lambda_p\}$ maps the outputs of all processes into the output associated with the base model. In the next section we provide the formal description of a πHyFlow process.

2.3 πHyFlow process model

A process is a sequence of actions that usually take some amount of virtual (simulation) time to be executed. Processes are coordinated by the base model. The base model chooses a process

that can be executed and resumes it. After executing, the process suspends itself and gives the control back to base model. This explicit invocation is necessary since simulation processes are non-preemptive. Given a base model $M_B = (X, Y_B, P_B, P_{0,B}, \zeta, \Pi, \pi, \sigma, \{\Lambda_p\})$, the model of a process $\varpi \in \Pi$ is defined by:

$$M_{\varpi}^{B} = (Y, I, P, P_0, \kappa, \{\rho_i\}, \{\omega_i\}, \{\varkappa_i\}, \{\delta_i\}, \{\Lambda_i^{\mathrm{c}}\}, \{\lambda_i^{\mathrm{d}}\}),$$

where:

Y is the set of output flow values,
  Y^{c} is the set of continuous output flow values
  Y^{d} is the set of discrete output flow values
I is the set of indexes,
P is the set of p-states,
P_0 is the set of (valid) initial p-states,
$\kappa : P \longrightarrow I$ is the index function,
for all $i \in I$:
  $\rho_i : P \longrightarrow \mathbb{H}_0^{+\infty}$ is the time-to-input function,
  $\omega_i : P \longrightarrow \mathbb{H}_0^{+\infty}$ is the time-to-output function,
  $\varkappa_i : P \times P_B \longrightarrow \{\top, \bot\}$ is the condition function,
  $\delta_i : S \times P_B \longrightarrow P \times P_B$ is the transition function,
  $\Lambda_i^{\mathrm{c}} : S \times P_B \longrightarrow Y^{\mathrm{c}}$ is the continuous output function,
  $\lambda_i^{\mathrm{d}} : P \times P_B \longrightarrow Y^{\mathrm{d}}$ is the partial discrete output function,
  $\Lambda_i^{\mathrm{d}} : S \times P_B \longrightarrow Y^{\mathrm{d}} \cup \{\varnothing\}$ is the discrete output function defined by:
  $\Lambda_i^{\mathrm{d}}((p, e), p_B) = \begin{cases} \lambda_i^{\mathrm{d}}(p, p_B) & if (e = \omega_i(p)) \\ \varnothing & otherwise \end{cases}$
  with $S = \{(p,\ e) |\ p \in P,\ 0 \le e \le \nu_{\kappa(p)}(p)\}$, the state set,
  and $\nu_i(p) = \min\{\rho_i(p), \omega_i(p)\}$, $i = \kappa(p)$, is the time-to-transition function,

For time specification, πHyFlow uses the set of hyperreal numbers $\mathbb{H}$, that enables to express causality, by assuming that a transition occurring at time t, changes process p-state at time $t + \varepsilon$, where $\varepsilon \in \mathbb{H}$ is an infinitesimal.

A process defines only its output, while the *input* is *inferred*, as mentioned before, from base model p-state. A process defines its own (private) p-state, for reducing inter process dependency. A process dynamic behavior is ruled by six structured/split functions, being the segments currently active determined by the index function κ. The active function segments associated with p-state $p \in P$ are $(\rho_i, \omega_i, \varkappa_i, \delta_i, \Lambda_i^{\mathrm{c}}, \lambda_i^{\mathrm{d}})|_{i=\kappa(p)}$.

The time-to-input-function $\{\rho_i\}$ specifies the interval for sampling (reading) a value. Since each process specifies its own reading interval, sampling is made asynchronously, and it can be made independently by any process. The time-to-output-function $\{\omega_i\}$ specifies the interval to produce (write) a discrete flow. The condition function $\{\varkappa_i\}$, checks whether the process has conditions to run given base model p-state and its own p-state. While $\{\rho_i\}$ and $\{\omega_i\}$ specify a time interval for process re-activation, $\{\varkappa_i\}$ checks if the process can be re-activated at the *current time*. Function $\{\delta_i\}$ specifies process and base model p-states after process re-activation. Function $\{\Lambda_i^{\mathrm{c}}\}$ specifies process continuous output flow, and $\{\Lambda_i^{\mathrm{d}}\}$ specifies process discrete output flow. The former can be non-null at every time instant, while the latter can only be non-null at a finite number of time instants during a finite time interval. We note that $\{\Lambda_i^{\mathrm{c}}\}$ can provide an exact description for an arbitrary continuous signal based on a discrete formalism. Some approaches are limited to piecewise constant representations of continuous signals (Bastian et al. 2011), Lee and Zheng (2005).

Process description is made around the concept of process index that enables a partitioning of several functions, like, conditions and transitions. Although process description looks complex, the index can be mapped into the *program count* concept that is implicitly defined in programming languages sequence of statements. πHyFlow processes have thus a straightforward implementation. The simplicity of this approach is shown in Pseudo-codes 1 and 2.

A simulation process gets most of its modeling expressiveness by enabling the dynamic switching of the active functions that describe its behavior. The semantics of base models and processes are given in the next sections.

2.4 πHyFlow base component

The semantics of πHyFlow base models is described using the concept of *component*. The alternative concept of *iterative system specification* (Barros 2002) could also be used. We found, however, that components can express model semantics in a simpler form, when using the set of hyperreals, instead the set of real numbers, to define time. Additionally, the component concept provides a simple description of dynamic network models. Given that base model semantics rely on process semantics, we refer here informally to the latter, postponing the formal description to the next section.

For each base model there is an associated component that is responsible for its simulation. The component provides base model operational semantics enabling πHyFlow base model unambiguous interpretation and implementation. A component is the actual entity that is placed into simulation. As such, a component keeps its current state and defines a set of actions to compute its next state, and component output, for example. Actions are described using base model definition.

In component definition a variable v is represented by $\langle v \rangle$. In action definition, the assignment to variable v of a value x is represented by $v \leftarrow x$, and The "forall" operator represents an iteration over a set or sequence. An action can also behave like a function and *return* a value (using the keyword "return"). A base component associated with model $M_B = (X, Y, P, P_0, \zeta, \Pi, \pi, \sigma, \{\Lambda_p\})$, is defined by:

$$\mathcal{C}_B = (\langle v, p \rangle, N, \Omega, \Delta),$$

where:

$v \in Y^{\varnothing}$, is the output value variable,

$p \in P$, is the current p-state variable,

$N :: \longrightarrow \mathbb{H}$, is the component next time transition action, defined by:

$$\mathrm{N}() \triangleq \text{return } \min_{i \in \pi(p)} \{\mathrm{N}_i()\}$$

$\Omega :: \mathbb{H}$, is the output action, defined by:

$$\Omega(t) \triangleq v \leftarrow \Lambda_p\Big(\bigtimes_{i \in (\sigma \circ \pi)(p)} \Lambda_i(t, p)\Big),$$

$V :: \longrightarrow Y^{\varnothing}$, is the output value action, defined by:

$$V() \triangleq \text{return } v$$

$\Delta :: \mathbb{H} \times X^{\varnothing}$, is the transition action, defined by:

$$\Delta(t, (x_c, x_d)) \triangleq$$

```
        if (t ≠ N() ∧ x_d = ∅) return
12      p ← ζ(p, (x_c, x_d))
        M ← {i | i ∈ π(p) ∧ N_i() = t}
14      forall (i ∈ σ(M)) p ← Δ_i(p)
        do {
16      Z ← {i | i ∈ π(p) ∧ K_i(p))}
          if (Z ≠ φ) {
18          i ← head(σ(Z))
            p ← Δ_i(t, p)
20          M ← M + i
          }
22      } while (Z ≠ φ)
        forall (i ∈ M) U_i(t)
```

A base component associated with model M_B keeps its output in variable v (line 1). The current p-state p is stored in line 2. Next time transition action N (line 3), computes the minimum re-activation time (next time) of the inner/child processes (line 4). Base component output action (line 5) sets variable v according to the outputs of the inner processes (line 6). Variable v value can be accessed through the output value action (lines 7-8). The transition action (line 9) defines base model next state based on the current time and the input value. The condition of line 11 simplifies network component transition definition (described in Section 2.8). Line 12 computes component the new p-state based on the input value. Line 13 finds the set of processes that are scheduled to undergo a transition at time t. These processes are ranked and triggered at time t, possibly changing the base component p-state (line 14). Since the p-state has changed, we find (line 16), the processes that can be re-activated, i.e., the processes whose conditions functions evaluate to $\top$. Line 18 chooses the process with the highest rank, which is re-activated in line 19. This sequence is repeated until there is no more process that can undergo a conditional transition (line 22). A process can be triggered several times in this cycle, since the guard condition also depends on the shared p-state that, in general, is modified at each transition. All processes that have undergone a transition will execute an update action (line 23). This action is described in the next section.

2.5 πHyFlow process simulator

The process simulator is responsible for updating process p-state according to the corresponding model. A process simulator describes the coroutine-like (non-preemptive) semantics of each πHyFlow process. For processes we use the simulator concept since, contrarily to base components, a simulator is a non-modular entity that can only exist within a specific parent base component. In a practical implementation, coroutines representing processes can only exist within some context, an object for example, being able to share memory with the coroutines in the same context. We emphasize that a πHyFlow process provides just a set of operators letting process semantics largely undefined and open to different interpretations. The simulator is required to precisely define process dynamic-behavior/operational-semantics. A simulator keeps the output value, the current p-state, and the time when the last transition has occurred. A simulator for process ϖ with model $M^B_\varpi = (Y, I, P, P_0, \kappa, \{\rho_i\}, \{\omega_i\}, \{\varkappa_i\}, \{\delta_i\}, \{\Lambda^c_i\}, \{\lambda^d_i\})$ and associated with base model $M_B = (X, Y, P, P_0, \zeta, \Pi, \pi, \sigma, \{\Lambda_p\})_B$ is defined by:

$$\mathcal{S}^B_\varpi = (\langle v, (p, t_L)\rangle, N, \Omega, V, K, \Delta, U),$$

where:

$v \in Y^{\varnothing}$, is the output value,

2 $(p,\ t_L)$ is the simulator current state, with $p \in P$ the simulator p-state and $t_L \in \mathbb{H}_0^{+\infty}$ the transition time to current p-state p,

4 $N :: \longrightarrow \mathbb{H}$, is the next transition time, defined by:

$$\mathrm{N}() \triangleq \text{return } t_L + \min\{\rho_{\kappa(p)}(p), \omega_{\kappa(p)}(p)\}$$

6 $\Omega :: \mathbb{H} \times P_B$, is the output action, defined by:

$$\begin{aligned} &\Omega(t,\ p_B) \triangleq \\ 8 \quad &\quad e \leftarrow t - t_L \\ &\quad v \leftarrow (\Lambda^{\mathrm{c}}_{\kappa(p)}((p,\ e),\ p_B),\ \Lambda^{\mathrm{d}}_{\kappa(p)}((p,\ e),\ p_B)) \end{aligned}$$

10 $V ::\longrightarrow Y^{\varnothing}$, is the output value, defined by:

$$V() \triangleq \text{return } v$$

12 $K :: P_B \longrightarrow \{\top,\ \bot\}$, is the condition verification, defined by:

$$K(q) \triangleq \text{return } \varkappa_{\kappa(p)}(p,\ q)$$

14 $\Delta :: \mathbb{H} \times P_B \longrightarrow P_B$, is the transition action, defined by:

$$\begin{aligned} &\Delta(t,\ q) \triangleq \\ 16 \quad &\quad (p,\ q') \leftarrow \delta_{\kappa(p)}((p,\ t - t_L),\ q) \\ &\quad \text{return } q' \end{aligned}$$

18 $U :: \mathbb{H}$, is the time update defined by:

$$U(t) \triangleq t_L \leftarrow t + \varepsilon$$

Lines 4-5 define simulator next transition time. The output action stores the current output value in variable v (lines 6-9). This value can be read by action V (lines 10-11). The condition verification action (lines 12-13) checks whether the process has conditions to run. The transition action (lines 14-17) computes the new simulator and parent base component p-states. The time update action (line 18) sets process simulator t_L (line 19).

2.6 πHyFlow network model

πHyFlow networks are composed by base or other network models. Additionally, each network has a special component, named as the executive, that is responsible for defining network topology (composition and coupling). A πHyFlow network model associated with name N is defined by:

$$M_N = (X, Y, \eta),$$

where:

$X = X^{\mathrm{c}} \times X^{\mathrm{d}}$ is the set of network input flows,
$Y = Y^{\mathrm{c}} \times Y^{\mathrm{d}}$ is the set of network output flows,
η is the name of the dynamic topology network executive.

The executive model is a πHyFlow base model extended with topology related operators. This model is defined by:

$$M_\eta = (X,\ Y,\ P,\ P_0,\ \zeta,\ \Pi,\ \pi,\ \{\Lambda_p\},\ \Sigma^*,\ \gamma),$$

where:
Σ^* is the set of network topologies,
$\gamma : P \longrightarrow \Sigma^*$ is the topology function.

The network topology $\gamma(p_\alpha) \in \Sigma$, corresponding to the p-state $p_\alpha \in P$, is given by:

$$\gamma(p_\alpha) = (C_\alpha, \{I_{i,\alpha}\} \cup \{I_{\eta,\alpha}, I_{N,\alpha}\}, \{F_{i,\alpha}\} \cup \{F_{\eta,\alpha}, F_{N,\alpha}\}),$$

where:

C_α is the set of names associated with the executive p-state p_α,
for all $i \in C_\alpha + \eta$:
$I_{i,\alpha}$ is the sequence of influencers of i,
$F_{i,\alpha}$ is the input function of i,
$I_{N,\alpha}$ is the sequence of network influencers,
$F_{N,\alpha}$ is the network output function,

for all $i \in C_\alpha$
$M_i = (X, Y, P, P_0, \zeta, \Pi, \pi, \sigma, \{\Lambda_p\})$, for base models,
$M_i = (X, Y, \eta)$, for network models.

Variables are subjected to the following constraints for all $p_\alpha \in P_\alpha$:

$N \notin C_\alpha$,
$\eta \notin C_\alpha$,
$N \notin I_{N,\alpha}$,
$F_{N,\alpha} : \underset{k \in I_{N,\alpha}}{\times} Y_k \longrightarrow Y^{\varnothing}$,
$F_{i,\alpha} : \underset{k \in I_{i,\alpha}}{\times} V_k \longrightarrow X_i^{\varnothing}$,
where $V_k = \begin{cases} Y_k^{\varnothing} & \text{if } k \neq N \\ X^{\varnothing} & \text{if } k = N \end{cases}$
$F_{N,\alpha}((v_{c,k_1}, \varnothing), (v_{c,k_2}, \varnothing), ...) = (y_{c,N}, \varnothing)$,
$F_{i,\alpha}((v_{c,k_1}, \varnothing), (v_{c,k_2}, \varnothing), ...) = (x_{c,i}, \varnothing)$.

Constraints 1 and 2 impose that the executive cannot remove neither the network nor itself. Constraint 3 enforces causality (Section 2.3). Constraints 7 and 8 are a characteristic of discrete systems and impose that a non-null discrete flow cannot be created from a sequence composed exclusively of null discrete flows.

The topology of a network is defined by its executive through the topology function γ, which maps the executive p-state into network composition and coupling. Topology adaption can thus be achieved by changing executive p-state. A πHyFlow network model is simulated by a πHyFlow network component that performs the orchestration of network inner components. Network simulation is achieved by a general communication protocol that relies only on the component interface. This protocol is independent from model details, enabling the composition of components that are handled as black boxes.

2.7 πHyFlow executive component

Before describing the network component, we define first the executive component, an extension to the base component, that introduces the topology function required to estab-

lish network topology. A HyFlow executive component $\mathcal{C}_\eta$ associated with the executive model $M_\eta = (X,\ Y,\ P,\ P_0,\ \zeta,\ \Pi,\ \pi,\ \{\Lambda_p\},\ \Sigma^*,\ \gamma)$ is defined by:

$$\mathcal{C}_\eta = (\langle v, (p, t_L)\rangle, N, \Omega, \Delta, \Gamma),$$

where:

$\Gamma :: \longrightarrow \Sigma^*$, is the executive topology action defined by:
$\Gamma() \triangleq$ return $\gamma(p)$

The Γ action returns network topology at the current time, based on executive current p-state. Since the executive model is an extension of the base model, the executive component inherits the actions previously defined for the base component.

2.8 πHyFlow network component

A network component is composed by one executive component and a set of other components. As mentioned before, components and their interconnections can change according to executive p-state. An exception being the executive that cannot be removed. Components can be base or other πHyFlow network components, making it possible to define networks hierarchically. A πHyFlow network component Ξ_N associated with the network model $M_N = (X, Y, \eta)$, executive $M_\eta = (X,\ Y,\ P,\ P_0,\ \zeta,\ \Pi,\ \pi,\ \{\Lambda_p\},\ \Sigma^*,\ \gamma)$, and current topology $\Gamma_\eta() = (C, \{I_i\} \cup \{I_\eta, I_N\}, \{F_i\} \cup \{F_\eta, F_N\})$, is defined by:

$$\mathcal{C}_N = (\langle v\rangle, N, \Omega, \Delta),$$

where:

$v \in Y$, is network component output,
2 $\mathrm{N} :: \longrightarrow \mathbb{H}$, is the next transition time, defined by:
$\mathrm{N}() \triangleq$ return $\min\{\mathrm{N}_i()|\ i\ \in\ C + \eta\}$
4 $\Omega :: \mathbb{H}$, is the network output action, defined by:
$\Omega(t) \triangleq v \leftarrow F_N(\underset{i\ \in\ I_N}{\times}\ \Lambda_i(t))$
6 $V :: \longrightarrow Y^\varnothing$, is the network output value action, defined by:
$V() \triangleq$ return v
8 $\Delta :: \mathbb{H} \times X^\varnothing$, is the network component transition, defined by:
$\Delta(t,\ x) \triangleq$
10 forall $(i \in C)\ \Delta_i(t, F_i(\underset{j\ \in\ I_i}{\times}\ v_j))$
$\Delta_\eta(t, F_i(\underset{j\ \in\ I_i}{\times}\ v_j))$
12 with $v_j = \begin{cases} \mathrm{V}_j() & \text{if } j \neq N \\ x & \text{if } j = N \end{cases}$

Network next transition time (lines 2-3) is defined as the minimum transition time of network components. Network output is defined based on the output of its inner components (lines 4-5). This value is stored in line 1 and can be accessed by the output action (lines 6-7). The transition action (line 8) is divided in two steps. In the first step, components, except the executive, compute their own transition (line 10). To simplify transition description, the decision to actually make the transition is made by each component, as mentioned in Section 2.4. The executive transition is only performed as the last one (line 11), since the new topology will only be used after transition time, at $t + \epsilon$.

2.9 πHyFlow component simulation

To perform a simulation, it is necessary to define a mechanism to execute component transitions according to their time advance specification. The simulation of a component, both base or network, is performed by the action:

$$\mathcal{S} :: \mathfrak{C} \times \mathbb{H},$$

where $\mathfrak{C} = \{c |\ M_c = (\{\} \times X^{\mathrm{d}}, Y, \ldots)\}$, is the set of names associated with πHyFlow models (base or network) defining a null continuous input flow interface. The simulation action is defined by:

$\mathcal{S}(c,\ end) \triangleq$
2 $clock \leftarrow N_c()$
while $(clock < end)$ {
4 $\Omega_c(clock)$
$\Delta_c(clock, (\varnothing,\ \varnothing))$
6 $clock \leftarrow N_c()$
}

The simulation loop involves a sequence of steps: compute current component output (line 4), trigger component transition at time *clock* (line 5) and compute the time of next transition that becomes the new *clock* (line 6). Line 4 enforces causality, since it computes component output before the transition is performed.

3 Applications

We demonstrate πHyFlow modeling ability through examples of hybrid and discrete base models. Some base models are used in subsequent sections for defining network models.

3.1 Continuous to piecewise-constant signal converter

We consider a model to convert a continuous signal into a piecewise constant representation based on sampling. A continuous to piecewise-constant signal converter (CPC) samples its continuous flow input at a fixed rate and produces a piecewise constant signal. Simultaneously, the CPC provides a sequence of discrete flows corresponding to the sampling times. A CPC model is described by:

$$M_{CPC} = (X, Y, P, P_0, \zeta, \Pi, \pi, \sigma, \{\Lambda_p\}),$$

where:

$X = \mathbb{R} \times \{\}$,
2 $Y = \mathbb{R} \times \mathbb{R}$,
$P = \mathbb{R}$,
4 $P_0 = \{v = 0\}$,
$\zeta(v, (v', \varnothing)) = v'$,
6 $\Pi = \{\tau\}$, the sampling process,
$\pi(v) = \Pi$,
8 $\sigma(\{\tau\}) = (\tau)$,
$\Lambda_v(\Lambda^{\mathrm{c}}_{i,\tau}(s_\tau, v), \Lambda^{\mathrm{d}}_{i,\tau}(s_\tau, v)) = (\Lambda^{\mathrm{c}}_{i,\tau}(s_\tau, v), \Lambda^{\mathrm{d}}_{i,\tau}(s_\tau, v))$,

10 with $i = \kappa_\tau(p_\tau)$.

Base model shared p-state stores the current sample in variable v (lines 3-4). The input function updates the sampled value (line 5). Only one process for sampling is considered (line 6). This process is defined by:

$$M_\tau^{CPC} = (Y,\ I,\ P,\ P_0,\ \kappa,\ \{\rho_i\},\ \{\omega_i\},\ \{\nsucc_i\},\ \{\delta_i\},\ \{\Lambda_i^{\mathrm{c}}\},\ \{\lambda_i^{\mathrm{d}}\}),$$

where

$Y = \mathbb{R} \times \mathbb{R}$
2 $I = \{0, 1, 2\}$,
$P = I \times \mathbb{R}_0^+$,
4 $P_0 = \{(i = 0, sTime | \ sTime \in \mathbb{R}_0^+)\}$,
$\kappa(i, sTime) = i$,
6 $\rho_0(i, sTime) = 0$
$\rho_1(i, sTime) = \infty$
8 $\rho_2(i, sTime) = sTime$,
$\omega_{0,2}(i, sTime) = \infty$,
10 $\omega_1(i, sTime) = 0$,
$\nsucc_{0,1,2}(i, sTime) = \bot$,
12 $\delta_{0,2}(((i, sTime), e), v) = ((1, sTime), v)$,
$\delta_1(((i, sTime), e), v) = ((2, sTime), v)$,
14 $\Lambda^{\mathrm{c}}_{0,1,2}(((i, sTime), e), v) = v$,
$\lambda^{\mathrm{d}}_{0,2}((i, sTime), v) = \varnothing$,
16 $\lambda^{\mathrm{d}}_3 1((i, sTime), v) = v$.

Process p-state includes the current index i, and the sampling interval $sTime$ (lines 3-4). These intervals are defined by function ρ (lines 6-8). The first sample occurs at time 0 (line 6). After this time, sampling is made at regular intervals of length $sTime$ (line 8). Discrete outputs are created immediately after sampling (line 10). The transition function is responsible for updating the current index (lines 12-13). Converter continuous output value is described in line 14. The discrete flow output is non-null for index $i = 1$ (line 16).

Figure 3 gives an overview of the CPC base model. At index 0 the CPS starts by immediately sampling the input and assigns the read value to variable v ($[0, >> v]$), changing then to index 1. At index 1, it produces a discrete flow with the current sample v ($[0, << v]$), and changes to index 2. At this index, the model waits $sTime$, and samples a new value v. After sampling the CPS goes back to index 1, where the loop repeats.
For simulation purposes we have used the CPS with the input function $5\cos(t)$, and a sampling interval of 0.5 s. CPC continuous flow is represented in Fig. 4, and the discrete flow is depicted in Fig. 5.

3.2 Geometric integrator

πHyFlow support for sampling can be used to define numerical integrators. We consider the representation of a geometric integrator commonly used to simulate energy conservation systems (Swope et al. 1982). Given a 2nd-order Ordinary Differential Equation (ODE):

$$\ddot{y} = f(x(t), y(t)), \text{ with } y(0) = y_0, \dot{y}(0) = \dot{y}_0,$$

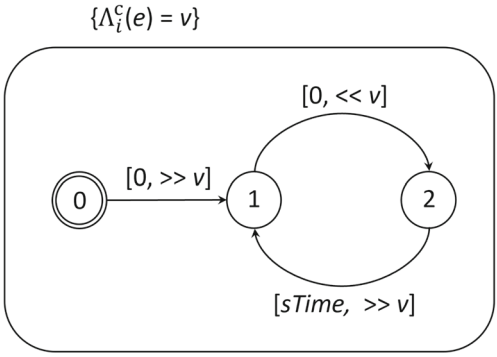

Fig. 3 Converter state diagram

and using the variable $v(t) = \dot{y}(t)$, a fixed step size h, 2nd-order ODE, 2nd-degree polynomial approximation, geometric integrator is described by the equations (Swope et al. 1982):

$$y_{n+1} = y_n + hv_n + \frac{1}{2}h^2 f_n, \tag{1}$$

$$v_{n+1} = v_n + \frac{h}{2}(f_n + f_{n+1}). \tag{2}$$

We exploit here the πHyFlow ability to define continuous flows for providing the interpolation function that can be associated with Eq. 1. As a result, we can calculate the position, not only at the sampling instants, but at any point in time. A geometric integrator can be described by the πHyFlow base model:

$$M_G = (X, Y, P, P_0, \zeta, \Pi, \pi, \sigma, \{\Lambda_p\}),$$

where:

$X = \mathbb{R}^3 \times \{\}$,
$Y = \mathbb{R}^3 \times \{\}$,
$P = \mathbb{R}^3$,
$P_0 = \{accel \in \mathbb{R}^3\}$,
$\zeta((accel), (x_c, x_d)) = x_c$,
$\Pi = \{\gamma\}$, the geometric integrator process,
$\pi(accel) = \{\gamma\}$,
$\sigma(\{\gamma\}) = (\gamma)$,

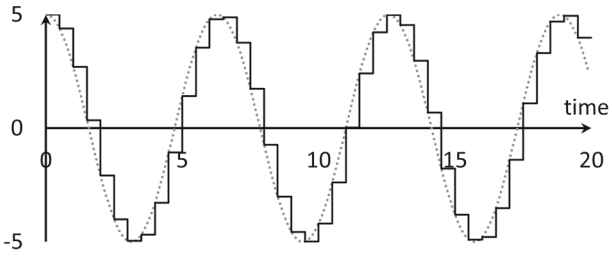

Fig. 4 Piecewise constant continuous flow generated by the CPC for the input function $5\cos(t)$

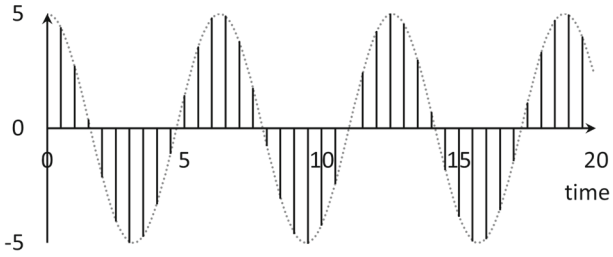

Fig. 5 CPC discrete output flow for 5 cos(t)

$\Lambda_{(accel)}(\Lambda^{c}_{i,\tau}(s_\gamma), \Lambda^{d}_{i,\tau}(s_\gamma))) = (\Lambda^{c}_{i,\tau}(s_\gamma), \varnothing)$,
10 with $i = \kappa_\gamma(p_\gamma)$.

The integrator samples acceleration values (line 1) and produces a continuous output flow corresponding to its position (line 2). The acceleration is stored in variable *accel* (line 4). The integrator has one process defined by:

$$M^G_\gamma = (Y,\ I,\ P,\ P_0,\ \kappa,\ \{\rho_i\},\ \{\omega_i\},\ \{\varkappa_i\},\ \{\delta_i\},\ \{\Lambda^{c}_i\},\ \{\lambda^{d}_i\}),$$

where:

$Y = \mathbb{R}^3 \times \{\}$
2 $I = \{0, 1\}$,
$P = I \times \mathbb{R}^+_0 \times \mathbb{R}^3 \times \mathbb{R}^3 \times \mathbb{R}^3$,
4 $P_0 = \{(i = 1, sTime, pos, vel, accel) |\ sTime \in (0, \infty), pos \in \mathbb{R}^3, vel \in \mathbb{R}^3, accel \in \mathbb{R}^3\}$,
$\kappa(i, sTime, pos, vel, accel) = i$,
6 $\rho_0(i, sTime, pos, vel, accel) = 0$,
$\rho_1(i, sTime, pos, vel, accel) = sTime$,
8 $\omega_{0,1}(i, sTime, pos, vel, accel) = \infty$
$\varkappa_{0,1}((i, sTime, pos, vel, accel), accel') = \perp$,
10 $\delta_{0,1}(((i, sTime, pos, vel, accel), e), accel') =$
$((1, sTime, pos + vel \cdot e + \frac{1}{2} accel \cdot e^2, vel + \frac{1}{2}(accel + accel') \cdot e, accel'), accel')$,
12 $\Lambda^{c}_{0,1}(((i, sTime, pos, vel, accel), e), accel') = pos + vel \cdot e + \frac{1}{2} accel \cdot e^2$,
$\lambda^{d}_{0,1}((i, sTime, pos, vel, accel), accel') = \varnothing$.

The integrator stores position, velocity, and acceleration (lines 3-4), and defines the time step $sTime$ (line 7). The step is constant except when simulation starts, where it is 0 (line 6). The transition function implements Eqs. 1 and 2 (lines 10-11). Integrator continuous output flow corresponding to the position described by Eq. 1 is given in line 12. This model is used in Section 3.5 to define a two-mass pendulum.

3.3 Infinite servers (Delay)

As pointed in previous work, the PI active server approach can provide an efficient model for many systems, making it preferable to the *naive* PI active client alternative (Henriksen 1981), taken by languages like GPSS (Henriksen 1981), and Simscript (Russel 1999). Some systems, however, can be more efficiently represented by the active client view. If we take,

for example the representation of a single delay, the active server will require an infinite number of processes, many of them possibly inactive and waiting for a customer. However, the active client approach requires only one process for each client, removing the requirement for an infinite number of processes. πHyFlow makes it possible to combine both the active server and active client approaches in the same base model, due to the support for dynamic creation/destruction of processes. An infinite sever model, can be represented using the active client approach, given that we can create a process for each arriving client, and destroy it when the client finishes service. An infinite server (delay) model can be described by:

$$M_D = (X,\ Y,\ P,\ P_0,\ \zeta,\ \Pi,\ \pi,\ \sigma,\ \{\Lambda_p\}),$$

where:

$X = \{\} \times \mathcal{P}(\mathbb{N})$,
$Y = \{\} \times \mathcal{P}(\mathbb{N})$,
$P = \mathcal{P}(\mathbb{N})$,
$P_0 = \{A = \{\}\}$,
$\zeta(A, (\varnothing, x)) = A \cup x$,
$\Pi = \mathcal{P}(\mathbb{N})$,
$\pi(A) = A$,
$\sigma(A) = (\ldots, a_{k-1}, a_k, a_{k+1}, \ldots)$, such that
$\ldots < a_{k-1} < a_k < a_{k+1} < \ldots$, and
$\{\ldots, a_{k-1}, a_k, a_{k+1}, \ldots\} = A$,
$\Lambda_A(\ldots, (\Lambda^{\mathrm{c}}_{i,a_k}(s_{a_k}, A), \Lambda^{\mathrm{d}}_{i,a_k}(s_{a_k}, A)), \ldots) =$
$(\varnothing, \{y = \Lambda^{\mathrm{d}}_{i,a_k}(s_{a_k}, A) |\ a_k \in A' \wedge y \neq \varnothing\})$,
with $i = \kappa_{a_k}(p_{a_k})$, and $A' = \sigma(A) = (\ldots, a_k, \ldots)$.

The delay receives a set of integer identifiers (IDs) corresponding to the arriving clients (line 1). The IDs of processes finishing service are sent out as a discrete flow (line 2). Each arriving ID triggers the creation of the corresponding customer processes (line 5). Customers are ranked by their ID (lines 8-10). Server discrete flow output just collects the IDs of leaving clients (lines 11-13). A client process associated with identifier id is defined by:

$$M^D_C = (Y,\ I,\ P,\ P_0,\ \kappa,\ \{\rho_i\},\ \{\omega_i\},\ \{\varkappa_i\},\ \{\delta_i\},\ \{\Lambda^{\mathrm{c}}_i\},\ \{\lambda^{\mathrm{d}}_i\}),$$

where:

$Y = \{\} \times \mathbb{N}$,
$I = \{1, 2\}$,
$P = \mathbb{N} \times I \times \mathbb{R}^+_0$,
$P_0 = \{(id, i = 1, dTime = r.v.) |\ id \in \mathbb{N} \wedge dTime \in \mathbb{R}^+_0\}$,
$\kappa(id, i, dTime) = i$,
$\rho_{1,2}(id, i, dTime) = \infty$,
$\omega_1(id, i, dTime) = dTime$
$\omega_2(id, i, dTime) = \infty$
$\varkappa_{1,2}(id, i, dTime) = \perp$,
$\delta_1(((id, i, dTime), e), A) = ((id, 2, dTime), A - \{id\})$,
$\Lambda^{\mathrm{c}}_{1,2}(((id, i, dTime), e), A) = \varnothing$,
$\lambda^{\mathrm{d}}_1((id, i, dTime), A) = id$,
$\lambda^{\mathrm{d}}_2((id, i, dTime), A) = \varnothing$.

Client id is set at process creation being stored in the initial p-state (line 4). The delay time is set (line 4) to a random variate (not detailed here). A process waits for $dTime$ (line 7) and sends its id as a discrete flow value (line 12). Before leaving, the process removes itself from base model set of processes (line 10). πHyFlow ability to dynamically modify the set of processes provides a framework that enables the combination of the active client and the active server PI, while keeping the support for modular and hierarchical models.

3.4 Tank model

We consider the model of tank that can receive a piecewise constant flow of a liquid. Contrarily to previous models, a tank allows continuous and discrete input and output flows. The tank has minimum (zero) and maximum volume limits. When reaching these limits, the volume remains constant until current conditions are modified. For simplification, we do not model tank overflow. A tank is described by:

$$M_T = (X,\ Y,\ P,\ P_0,\ \zeta,\ \Pi,\ \pi,\ \sigma,\ \{\Lambda_p\}),$$

where:

$X = \mathbb{R} \times \{\text{event}\}$,
$Y = [0, \max] \times \{\text{event}\}$,
$P = \mathbb{R} \times \{\bot, \top\}$,
$P_0 = \{(rate = 0,\ flag = \bot)\}$,
$\zeta((rate,\ flag), (x_\text{c}, x_\text{d})) = (x_\text{c}, x_\text{d} = \text{event})$
$\Pi = \{\text{v}\}$, the (v)olume process,
$\pi(rate,\ flag) = \text{v}$,
$\sigma(\{\text{v}\}) = (\text{v})$,
$\Lambda_{(rate,\,flag)}(\Lambda^\text{c}_{i,\text{v}}(s_\text{v}, (rate,\ flag)), \Lambda^\text{d}_{i,\text{v}}(s_\text{v}, (rate,\ flag))) =$
$(\Lambda^\text{c}_{i,\text{v}}(s_\text{v}, (rate,\ flag)), \Lambda^\text{d}_{i,\text{v}}(s_\text{v}, (rate,\ flag)))$,
with $i = \kappa_\text{v}(p_\text{v})$.

The tank receives continuous and discrete flows describing a piecewise continuous input rate (line 1), making the output flow piecewise linear. The p-state stores the input rate and a flag to signal an input event (lines 3-4). When the input rate is modified, the model receives the discrete flow value "event", and the flag variable is set to $\top$ (line 5). There is only one process (v) in the tank (line 6). Tank output is computed by process v (line 9). This process is defined by:

$$M^T_\text{v} = (Y,\ I,\ P,\ P_0,\ \kappa,\ \{\rho_i\},\ \{\omega_i\},\ \{\varkappa_i\},\ \{\delta_i\},\ \{\Lambda^\text{c}_i\},\ \{\lambda^\text{d}_i\}),$$

where:

$Y = [0, \max] \times \text{event}$,
$I = \{1, 2\}$,
$P = [0, \max] \times \mathbb{R} \times \mathbb{R}^+_0$,
$P_0 = \{(volume, rate = 0, interval = \infty, i = 1) |\ volume \in [0, \max]\}$,
$\kappa(volume, rate, interval, i) = i$,
$\rho_{1,2}(volume, rate, interval, i) = \infty$,
$\omega_1(volume, rate, interval, i) = interval$,
$\omega_2(volume, rate, interval, i) = 0$,
$\varkappa_1((volume, rate, interval, i), (rate', flag)) = flag$,
$\varkappa_2((volume, rate, interval, i), (rate', flag)) = \bot$,

$$\delta_1(((volume, rate, interval, i), e_T), (rate', flag)) =$$
$$((volume', rate'', interval', 2), (rate', \perp)),$$
$$\delta_2(((volume, rate, interval, i), e_T), (rate', flag)) =$$
$$((volume', rate'', interval', 1), (rate', \perp)),$$
with
$$volume' = volume + e_T \cdot rate,$$
$$rate'' = \begin{cases} 0 & \text{if} \\ rate' & \text{otherwise,} \end{cases}$$
$$(rate' = 0) \vee (volume' = 0 \wedge rate' < 0) \vee (volume' = \max \wedge rate' > 0),$$
$$interval' = \begin{cases} \dfrac{\max - volume'}{rate''} & \text{if } rate'' > 0, \\ -\dfrac{volume'}{rate''} & \text{if } rate'' < 0, \\ \infty & \text{otherwise,} \end{cases}$$
$$\Lambda^{c}_{1,2}((volume, rate, interval, i), e_T), (rate', flag))) = volume + e_T \cdot rate,$$
$$\lambda^{d}_{1}((volume, rate, interval, i), (rate', flag)) = \varnothing,$$
$$\lambda^{d}_{2}((volume, rate, interval, i), (rate', flag)) = \text{event}.$$

Process v keeps tank volume, the current input rate, the time interval for the next event, and the current index (lines 3-4). The process does not sample its input (line 6) since it is driven by discrete flows. The transition function computes the time to reach one of the tank limits, given the current input/output rate (lines 11-12). When a transition occurs, the current volume is computed (line 14). In index 1, the transition can be triggered by variable $flag$ (line 9) that represents a discontinuity in the input flow. A transition can also be triggered when tank volume reaches a limit (lines 18-21). In these cases, tank rate is set to 0 to guarantee that volume stays within the admissible limits (line 16). The new rate is set to the current rate if volume constraints are guaranteed (line 17). When the new rate is zero, the tank remains in the current p-state until conditions are changed (line 21). Tank volume is piecewise linear (line 22) since tank psychical input/output rates are assumed as piecewise constant. (Formally, the tank has only one input signal, representing the difference between tank input and output rates.) In index 2, the process produces the discrete output flow "event" for signaling a discontinuity in tank volume (line 24). Tank model is used in Section 3.6 to describe πHyFlow dynamic topology networks.

3.5 Two-mass pendulum

We define now a πHyFlow network with a static topology, for representing the two-mass pendulum depicted in Fig. 6. Mass M_1 is constrained to move along the x-direction. For simplicity, we assume that M_1 motion on the x-direction has no bounds. M_2 can move along the 3-axis. M_1 and M_2 are connected through a spring with unstretched length L and stiffness k. We assume there is no friction.

Pendulum network topology is shown in Fig. 7, where the model of both M_1 and M_2 is the geometric integrator defined in Section 3.2. Pendulum model is given by:

$$M_{\mathrm{P}} = (X, Y, \mathrm{p}),$$

where:

$$X = \{\} \times \{\},$$

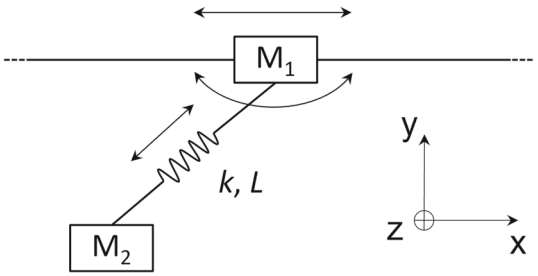

Fig. 6 Two-mass pendulum

$Y = \{\} \times \{\}$,
p is the executive name.

The executive model is defined by:

$$M_{\mathrm{p}} = (X,\ Y,\ P,\ P_0,\ \zeta,\ \Pi,\ \pi,\ \{\Lambda_p\},\ \Sigma^*,\ \gamma),$$

where:

$X = \{\} \times \{\}$,
2 $Y = \{\} \times \{\}$,
$P = P_0 = \{\}$,
4 $\zeta() = \varnothing$,
$\Pi = \{\}$,
6 $\sigma(\{\}) = ()$,
$\Lambda() = (\varnothing, \varnothing)$,
8 $\Sigma^* = \{\Sigma\}$,
$\gamma() = \Sigma$.

We consider a network without input/output values (lines 1-2), and with the static topology Σ (line 8), given by:

$C = \{\mathrm{M}_1, \mathrm{M}_2\}$,
2 $I_{\mathrm{M}_1} = (\mathrm{M}_1, \mathrm{M}_2)$,
$I_{\mathrm{M}_2} = (\mathrm{M}_2, \mathrm{M}_1)$,
4 $I_{\mathrm{P}} = I_{\mathrm{p}} = ()$,
$F_{\mathrm{M}_1}((p_1, \varnothing), (p_2, \varnothing)) =$

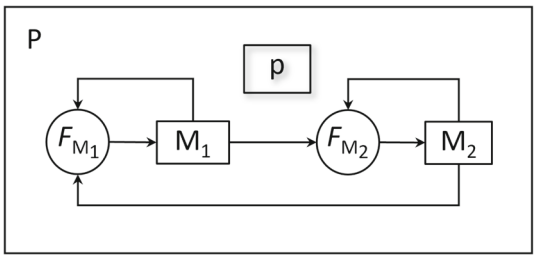

Fig. 7 Network model of the two-mass pendulum

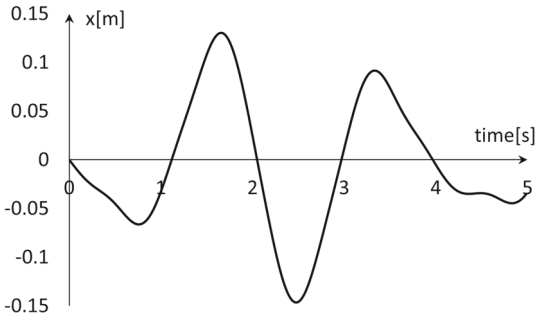

Fig. 8 Plot x(time) of mass M1

6 $\big(-\frac{k \cdot p_{1,2}}{m_1}\frac{(||p_{1,2}|| - L)}{||p_{1,2}||}\begin{pmatrix}1&0&0\\0&0&0\\0&0&0\end{pmatrix}, \varnothing\big)$, with $p_{1,2} = p_1 - p_2$,
$F_{\mathrm{M}_2}((p_2, \varnothing), (p_1, \varnothing)) =$
8 $\big(-\frac{k \cdot p_{2,1}}{m_2}\frac{(||p_{2,1}|| - L)}{||p_{2,1}||} - (0, g, 0), \varnothing\big)$, with $p_{2,1} = p_2 - p_1$,
$F_{\mathrm{P}}() = F_{\mathrm{p}}() = (\varnothing, \varnothing)$,
10 $M_{\mathrm{M}_1} = M_{\mathrm{M}_2} = M_\gamma$, where M_γ is the geometric integrator model defined in Section 3.2.

The acceleration at M_1 is computed in lines 5-6, where only the x-direction is considered. Lines 7-8 compute the acceleration at M_2. Here there is no constraint in the acceleration, and M_2 can move along the 3-directions.

Simulation results

For simulation we use the following initial parameters: $p_{0,\mathrm{M}_1} = (id = 1, sTime = 10^{-3}, pos = (0, 0, 0), vel = (-0.125, 0, 0))$; $p_{0,\mathrm{M}_2} = (id = 1, sTime = 10^{-3}, pos = (0, -1.5, 0), vel = (0.5, 0, 0))$. Spring has parameters $k = 5.0\ \mathrm{Nm}^{-1}$, and $L = 1.0$ m. M_1 has mass $m_1 = 0.4$ kg, M_2 has mass $m_2 = 0.1$ kg, and $g = 9.8\ \mathrm{ms}^{-2}$. For simplifying results, M_2 initial conditions were chosen so there is only motion in the x and y directions. M_1 position is depicted in Fig. 8, for a simulation time of 5 s.
For M_2, the x-y plot is depicted in Fig. 9. Given that the geometric integrators M_1 and M_2 have continuous output flows, the sampling interval for generating the plots can be arbitrarily

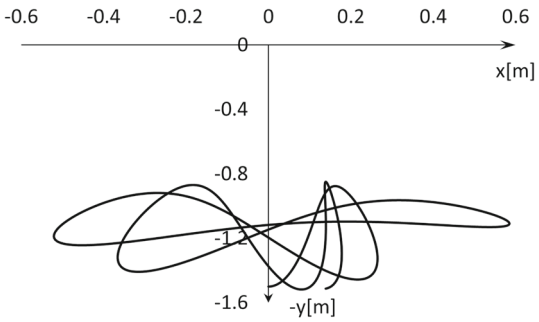

Fig. 9 Plot (x, y) of mass M2

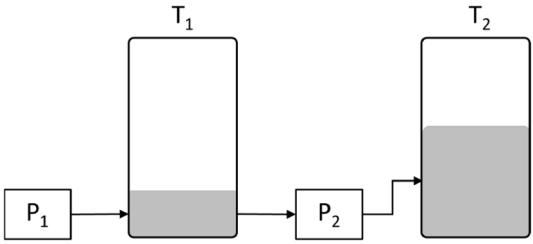

Fig. 10 Two-tank system

set for producing a smooth curve. For simplicity the sampler model for generating the results was omitted.

3.6 Two-tank system

For demonstrating πHyFlow dynamic topologies we interconnect two tanks described in Section 3.4 in a time-varying network that depends on tank volumes and rates. The system is depicted in Fig. 10. The model has tanks T_1 and T_2, and pumps P_1 and P_2, with rates r_1 and r_2, respectively. We do not provide the details of the pumps, but we assume they produce piecewise constant signals, similar to those generated by the CPC converter model described in Section 3.1.

The network needs to modify its topology to avoid chattering that would occur when T_1 is empty and $r_2 > r_1$. Under these conditions P_2 can only be operated at rate r_1 and the system becomes effectively represented by Fig. 12 that includes T_2, P_1 and P_2. The network keeps track of pumps rates, so it can switch back to the model represented by Fig. 10.

The initial tank network model is depicted in Fig. 11, and is given by:

$$M_{\mathrm{K}} = (X, Y, \mathrm{k}),$$

where:

$X = \{\} \times \{\}$,
$Y = \{\} \times \{\}$,
k is the executive name.

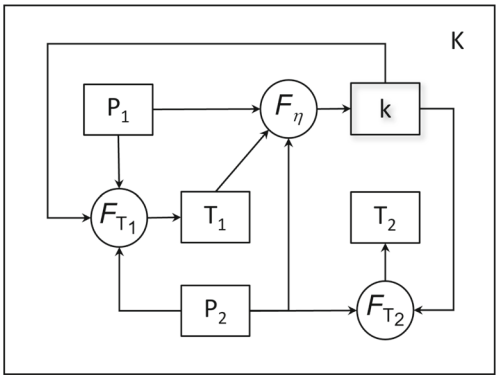

Fig. 11 Two-tank network model

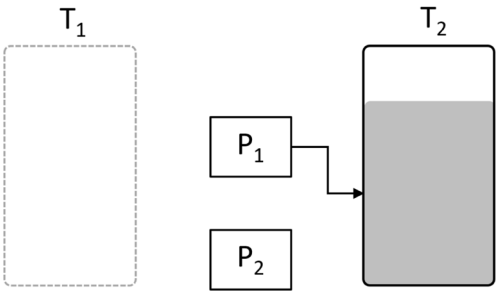

Fig. 12 Pump P_1 connected to T_2

The executive model is defined by:

$$M_{\mathrm{k}} = (X,\ Y,\ P,\ P_0,\ \zeta,\ \Pi,\ \pi,\ \{\Lambda_p\},\ \Sigma^*,\ \gamma),$$

where:

$X = \mathbb{R}^2 \times \{\text{event}\}$,
$Y = \{\} \times \{\text{event}\}$,
$P = \{0, 1\} \times \{\top, \bot\}$,
$P_0 = \{(tpy = 0, flag = \bot)\}$,
$\zeta((0, flag), ((v_1, r_{1,2}), \text{event})) =$
  $(1, \top)$ if $v_1 = 0 \wedge r_{1,2} \leq 0$,
  $(0, \bot)$ otherwise,
$\zeta((1, flag), ((v_1, r_{1,2}), \text{event})) =$
  $(0, \top)$ if $r_{1,2} > 0$,
  $(1, \bot)$ otherwise,
$\Pi = \{\alpha\}$,
$\sigma(\{\alpha\}) = (\alpha)$,
$\Lambda_{(tpy, flag)}(\Lambda^{\mathrm{c}}_{i,\alpha}(s_\alpha, (tpy, flag)), \Lambda^{\mathrm{d}}_{i,\alpha}(s_\alpha, (tpy, flag))) =$
$(\varnothing, \Lambda^{\mathrm{d}}_{i,\alpha}(s_\alpha, (tpy, flag)))$,
  with $i = \kappa_\alpha(p_\alpha)$,
$\Sigma^* = \{\Sigma_0, \Sigma_1\}$,
$\gamma(0, flag) = \Sigma_0$,
$\gamma(1, flag) = \Sigma_1$.

For simplicity, we consider Σ_0 as the initial topology, omitting the procedure to find this topology from tanks and pumps initial conditions. The executive receives v_1 (the volume of T_1), the difference $r_1 - r_2$, and events from T_1, P_1, and P_2 (line 1). It produces discrete flows to add discontinuities to the input signals received by T_1 and T_2 (line 2). The executive stores the current topology identifier, and a flag signaling a request for a change in topology (lines 3-4). As mentioned before, there are two topologies (line 15). When in topology 0, the network will switch to topology 1, if $v_1 = 0$ and $r_{1,2} = r_1 - r_2 \leq 0$ (lines 5-6). When in topology 1, the network switches to topology 0, if $r_{1,2} > 0$ (lines 8-9). Topology Σ_0 is defined by:

$C_0 = \{\mathrm{P}_1, \mathrm{P}_2, \mathrm{T}_1, \mathrm{T}_2\}$,
$I_{\mathrm{P}_1,0} = I_{\mathrm{P}_2,0} = I_{\mathrm{K},0} = ()$,
$I_{\mathrm{T}_1,0} = (\mathrm{P}_1, \mathrm{P}_2, \mathrm{k})$,

$I_{T_2,0} = (P_2, k)$,
$I_{k,0} = (T_1, P_1, P_2)$,
$F_{T_1,0}((r_1, d_1), (r_2, d_2), (\varnothing, d_k)) = (r_1 - r_2, d)$, with
$d =$
event if event $\in \{d_1, d_2, d_k\}$,
$\varnothing$ otherwise,
$F_{T_2,0}((r_2, d_2), (\varnothing, d_k)) = (r_2, d)$, with
$d =$
event if event $\in \{d_2, d_k\}$,
$\varnothing$ otherwise,
$F_{k,0}((r_1, d_1), (v_1, d_v), (r_2, d_2)) = ((v_1, r_1 - r_2), d)$, with
$d =$
event if event $\in \{d_1, d_2, d_v\}$,
$\varnothing$ otherwise,
$F_{K,0}() = (\varnothing, \varnothing)$,
$M_{T_1} = M_{T_2} = M_T$, where M_T is the tank model defined in Section 3.4.

T_1 input function maps the values received from P_1, P_2, and k, into the pair$(r_1 - r_2, d)$ (line 6). T_2 input function maps the values received from P_2, and k, into the pair(r_2, d) (line 10). Executive input function maps the values received from P_1, T_1, and P_2 into $((v_1, r_1 - r_2), d)$ (line 14). These values are used by the executive to perform changes int the topology, as described above.

When T_1 is empty and $r_1 < r_2$ the system can be represented by Fig. 12, where T_1 is removed, the connection $P_2 \to T_2$ is removed, and the link $P_1 \to T_2$ is created.

The corresponding network topology Σ_1 is depicted in Fig. 13 and defined by:

$C_1 = \{P_1, P_2, T_2\}$,
$I_{P_1,1} = I_{P_2,1} = I_{K,1} = ()$,
$I_{T_2,1} = (P_1, k)$,
$I_{k,1} = (P_1, P_2)$,
$F_{T_2,1}((r_2, d_2), (\varnothing, d_k)) = F_{T_2,0}((r_2, d_2), (\varnothing, d_k))$,
$F_{k,1}((r_1, d_1), (r_2, d_2)) = (r_1 - r_2, d)$, with
$d =$
event if event $\in \{d_1, d_2\}$,

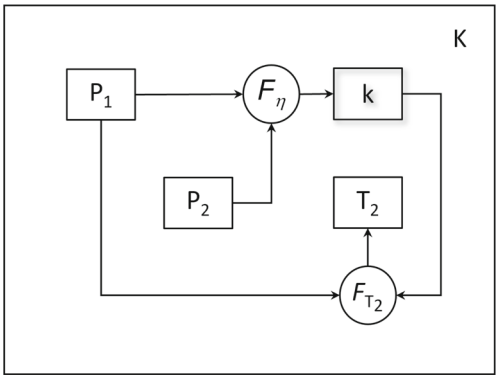

Fig. 13 One tank network model

$$\varnothing \qquad \text{otherwise,}$$

10 $F_{\mathrm{K},1}() = (\varnothing, \varnothing)$.

Σ_1 changes both network composition and coupling. T_1 is removed since it keeps the zero volume while topology remains in Σ_1. T_2 receives now its input from P_1, and not from P_2 (line 3). The executive receives inputs from P_1 and P_2 (line 4) since it needs the value $r_1 - r_2$ for deciding when to switch back to Σ_0.

Tanks are informed from structural changes, so they can receive updated input rates. The executive needs a process to signal the structural changes to tanks. This process is defined by:

$$M_\eta^{\mathrm{T}} = (Y,\ I,\ P,\ P_0,\ \kappa,\ \{\rho_i\},\ \{\omega_i\},\ \{\varkappa_i\},\ \{\delta_i\},\ \{\Lambda_i^{\mathrm{c}}\},\ \{\lambda_i^{\mathrm{d}}\}),$$

where:

$Y = \{\} \times \{\text{event}\}$,
2 $I = \{1, 2\}$,
$P = \mathbb{N}$,
4 $P_0 = \{i = 1\}$,
$\kappa(i) = i$,
6 $\rho_{1,2}(i) = \infty$,
$\omega_1(i) = \infty$,
8 $\omega_2(i) = 0$,
$\varkappa_{1,2}(i, (tpy, flag)) = flag$,
10 $\delta_1((i, e_T), (tpy, flag)) = (2, (tpy, \perp))$,
$\delta_2((i, e_T), (tpy, flag)) = (1, (tpy, \perp))$,
12 $\Lambda_{1,2}^{\mathrm{c}}((i, e_T), (tpy, flag)) = \varnothing$,
$\lambda_1^{\mathrm{d}}(i, (tpy, flag)) = \varnothing$,
14 $\lambda_2^{\mathrm{d}}(i, (tpy, flag)) = \text{event}$.

The executive process waits for the flag signaling a change in topology (line 9). It then sends a discrete flow to inform tanks that topology was changed (line 14). After this value is sent, the current index i is set to 1 (line 11).

Simulation results

For experiments we have considered T_1 with a maximum volume of 40 l. Pump rates r_1 and r_2, and the volume of T_1, v_1, are depicted in Fig. 14. From time 0 to 15 s, $r_1 > r_2$, and tank

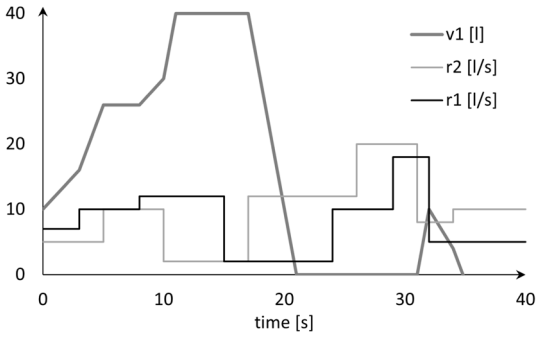

Fig. 14 Rates at pumps P_1 and P_2, and volume at tank T_1

volume increases. Under these conditions, at time 11 s, T_1 reaches it maximum volume that is kept until time 17 s, when $r_2 > r_1$. From this time on, the volume decreases until it reaches 0 at time 21 s. Here the network changes topology to Σ_1, and T_1 stays empty. At time 31 s, $r_1 > r_2$, and the topology is changed to Σ_0. The topology goes again to Σ_1 at time 34.8 s.

This example shows πHyFlow ability to represent hybrid systems with dynamic topology that includes support for modifying both composition and coupling.

4 πHyFlow++ framework overview

πHyFlow++ is an implementation of πHyFlow in MSVC++ 20. πHyFlow++ uses C++ support for modules, variants, lambdas, and coroutines. It uses the concept of port to segment continuous and discrete flows. For each discrete flow input πHyFlow++ assigns an input buffer that collects all values directed to that port. Each continuous output port is assigned to a process, that defines a function parameterized by the elapsed time since process last transition. Listing 1 provides πHyFlow++ implementation of the CPC converter described in Section 3.1. πHyFlow++ also provides the operators to support dynamic topology networks, like the one described in Section 3.6.

```
1  export class cpwc: public sim::component {
2     public:
3     std::vector<sim::port> in_ports_c() {return {"value"};}
4     std::vector<sim::port> out_ports_c() {return {"value"};}
5     std::vector<sim::port> out_ports_d() {return {"event"};}
6     cpwc(std::string_view const name, double sTime):
7     sim::component(name) {
8        init();
9        sampler("sampler", sTime);
10    }
11    sim_void sampler(std::string_view const name, double sTime) {
12       sim_start(name);
13       double value = 0;
14       output_c("value", [&](const double&)->sim::cvalue {
15          return value;
16       });
17       sim_wait sample(0, "value", value);
18       while (true) {
19          sim_wait out(0, "event", value);
20          sim_wait sample(sTime,"value", value);
21       }
22       sim_end;
23    }
24 };
```

Listing 1 πHyFlow++ continuous to piecewise-constant signal converter model.

The πHyFlow++ cpwc component defines the continuous input port "value" (line 3), the continuous output port "value" (line 4), and the discrete output port "event" (line 5). The sampler process is created in line 9 and is defined in line 11. The process sets the continuous flow associated with output port "value" (lines 14-16). The first sample is taken at time 0 (line 17). The sampler executes a loop where it sends the value through discrete output port "event" (line 19), and after waiting "sTime" units, it samples the continuous output port "value" (line 20). Given the declarative description of processes, we consider that πHyFlow++ leverages a simplified representation of complex systems.

5 Related work

The concepts of continuous flow and generalized sampling were introduced in the Continuous Flow System Specification (CFSS) formalism (Barros 2002). CFSS enables the exact representation of continuous signals in digital computers. These signals can be read using non-uniform sampling. Different components can also sample signals asynchronously. The support for multiple clocks was introduced in the Esterel language (Berry and Sentovich 2001), but continuous flows were limited to piecewise constant segments (Berry and Sentovich 2001).

The Hybrid Flow System Specification (HyFlow) formalism (Barros 2017) combines continuous flows (Barros 2002) and discrete events (Zeigler 1976). The process interaction worldview (PI) was introduced by SIMULA (Dahl et al. 1966). The formal specification of PI was introduced in (Zeigler 1976). This work, however, was limited to discrete event non-modular models, based on a static set of processes. This approach also provides a limited view on process conditional waiting. A more general approach to this problem was proposed in Cota and Sargent (1992). This work, however, did not add any support to model modularity.

Conditional waiting was also defined in Timed Process Algebras (TPAs) (Nicollin and Sifakis 1994), being time constrained to be discrete on earlier developments. A restriction of formal algebras like CSP is that they represent *resources*, like forks in the philosophers problem, as processes, requiring extra features, like an additional footman process (who only allows four philosophers to be seat simultaneously in a table with five seats), to avoid deadlock (Hoare, 1985, Chapter 2.5). On the contrary, πHyFlow processes access to shared variables supports Petri Net-like semantics where resources are only taken when they all become available. This semantics avoids introducing extrinsic causes of deadlock (Peterson 1981, Chapter 3.4.6).

Traditional representations of ODEs rely on the analog computer paradigm and require no explicit representation of numerical methods (Henzinger 1996). In these approaches, the modeler only needs to describe the ODEs (Praehofer 1991), and in some cases to choose the numerical method for handling the overall set of ODEs Fritzson (2003). Although at a first glance this is a good solution, since it frees users from numerical details, it has several limitations. The co-simulation (Bastian et al. 2011) is not guaranteed since ODEs need to be transformed and converted into a set of first-order ODEs and collectively solved by a single numerical integrator. Another limitation of these approaches is the difficulty to introduce new numerical integrators since they are not explicitly represented. Given that these frameworks only offer ODEs as first-class constructs, solutions requiring the combination of different families of numerical integrators can also not be described.

Statecharts (SCs) provide similar semantics of processes when regarding discrete event systems (Harel and Politi 1998). The concept of *orthogonal components* can be considered analogous to πHyFlow concurrent processes. πHyFlow presents, however, several major differences. Given SCs graphical notation it semantics is limited to a small set of operators like conditions and switch statements. We consider that, in more complex cases, a programming language supporting general purpose statements may be preferable to a complex set of annotations that need to be added to SCs. While πHyFlow provides a direct access to the time elapsed since the last transition, useful when a preemption occurs, SCs do not offer a direct access to this value. Although SCs enable sampling operations, these are performed on physical devices. SCs do not actually support dense outputs to enable the representation of continuous variables, being mostly limited to represent piecewise constant values. A major limitation of SCs, in the perspective of M&S, is the inability to support the dynamic cre-

ation/destruction of processes, limiting SCs expressiveness, making it difficult to represent, for example, the simple delay model described in Section 3.3.

SCs extension has been used in Simflow but keeping most of its features and limitations (Tripakis et al. 2005). ODEs are discretized making the overall system discrete time, where components can only communicate at discrete time instants. Likewise Modelica (Fritzson 2003), Simulink, maps ODEs into a set of 1st-order equations, making it difficult to extend Modelica or Simulink with geometric integrators, for example.

Extensions to Petri Nets have introduced the representation of continuous signals (Allamigeon et al. 2017; Ciardo et al. 1999; David and Alla 2001; Júlvez and Oliver 2019). In general, Petri Nets extensions to represent hybrid systems do not have the ability to describe numerical integrators, exhibiting the limitations pointed before for this type of approach. Petri Net-like formalisms are designed for analysis and optimization but, generally, they impose limitations to modeling constructs. On the contrary, πHyFlow is focused on simulation/performance evaluation providing, in general, more expressive semantics to describe models supporting, for example, arbitrary transition pre&post-conditions. Preemptive behavior based on generic rules can also be described in πHyFlow. Nevertheless, modeling and simulation is often complementary to Petri net-based modeling, being the choice limited, for example, by model large number of states (Valmari 1998).

Mixed-logical-dynamical (MLD) approaches combine hybrid models with control strategies, and they were designed to optimize hybrid systems using model predictive control (Bemporad and Morari 1999; Yaakoubi and Haggège 2022. ODE representation can be based on hybrid automaton ideal description without any reference to numerical integrators (Yaakoubi and Haggège 2022). MLD has also been described using discrete event models where the optimization algorithm uses piecewise constant velocities for describing vehicles moving in a highway (Fabiani and Grammatico 2018). In this case, simulation was performed without any relation to a particular modeling approach, suggesting an unstructured implementation (Fabiani and Grammatico 2018) based on a discrete time model. This points that MLD can be mapped into a hybrid modular modeling formalism like πHyFlow.

6 Conclusion

The πHyFlow formalism provides a representation for hierarchical, and modular hybrid systems. This formalism is intended to support modeling & simulation for performance evaluation. πHyFlow uses the concepts of continuous flow and generalized sampling to describe continuous systems. πHyFlow networks have a time-varying topology leveraging a simple description of systems that undergo structural changes. πHyFlow introduces the support for processes into the original HyFlow formalism. πHyFlow ability to dynamically create/destroy processes provides a framework to combine the active client and the active server process interaction worldviews. To the best of our knowledge this paper introduces the first formal definition of hybrid models semantics based on the process interaction worldview.

Funding Open access funding provided by FCT|FCCN (b-on).

Declarations

Conflict of Interest The author declares that he has no conflict of interest.

References

Allamigeon X, Boeuf V, Gaubert S (2017) Stationary solutions of discrete and continuous Petri nets with priorities. Performance Eval 113:1–12

Barros F (2002) Towards a theory of continuous flow models. Int J General Syst 31(1):29–39

Barros F (2017) Chattering avoidance in hybrid simulation models: a modular approach based on the HyFlow formalism. In: Symposium on theory of modeling and simulation

Bastian J, ClaußC, Wolf S, Schneider P (2011) Master for co-simulation using FMI. In: Proceedings of the 8th modelica conference

Bemporad A, Morari M (1999) Control of systems integrating logic, dynamics, and constraints. Automatica 35:407–427

Berry G, Sentovich E (2001) Multiclock Esterel. Correct hardware design and verification methods, LNCS 2144:110–125

Ciardo G, Nicol D, Trivedi K (1999) Discrete-event simulation of Fluid Stochastic Petri Nets. IEEE Trans Softw Eng 25(2):207–217

Cota B, Sargent R (1992) A modification of the process interaction world view. ACM Trans Model Comput Simulat 2(2):109–129

Dahl O-J, Myhrhaug B, Nygaard K (1966) SIMULA - An ALGOL-based simulation language. Commun ACM 9(9):671–678

David R, Alla H (2001) On hybrid Petri nets. Discrete Event Dynamic Systems: Theory and Applications 11:9–40

Fabiani F, Grammatico S (2018) A mixed-logical-dynamical model for automated driving on highways. In: IEEE Conference on decision and control, pp 1011–1016

Fritzson P (2003) Principles of Object-Oriented Modeling and Simulation with Modelica 2.1. Wiley

Harel D, Politi M (1998) Modeling Reactive Systems with Statecharts. McGraw-Hill

Henriksen J (1981) GPSS - finding the apropriate world-view. In: Winter simulation conference, pp 505–516

Henzinger T (1996) The theory of hybrid automata. In: 11th Annual IEEE symposium on logic in computer science, pp 278–292

Hoare C (1985) Communicating Sequential Processes. Prentice-Hall

Júlvez J, Oliver S (2019) Flexible Nets: a modeling formalism for dynamic systems with uncertain parameters. Discrete Event Dynamic Syst 29:367–392

Lee E, Zheng H (2005) Operational semantics of hybrid systems. Hybrid systems computation and control, of LNCS 3414:392–406

Nicollin X, Sifakis J (1994) An overview and synthesis on timed process algebras. Inf Comput 114:131–178

Nielson C, Larsen P, Fitzgerald J, Woodcock J, Peleska J (2015) Systems of systems engineering: basic concepts, model-based techniques, and research directions. ACM Comput Surv 48(2):1–41

Peterson J (1981) Petri Net Theory and the Modeling of Systems. Prentice-Hall

Praehofer H (1991) Systems theoretic formalisms for combined discrete-continuous system simulation. Int J General Syst 19(3):219–240

Russel E (1999) Building Simulation Models with Simscript II.5. CACI, La Jolla

Schruben L (1983) Simulation modeling with event graphs. Commun ACM 26(11):957–963

Swope W, Andersen H, Berens P, Wilson K (1982) A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters. J Chemical Phys 76(1):637–649

Tripakis S, Sofronis C, Caspi P, Curic A (2005) Translating discrete-time Simulink to Lustre. ACM Trans Embedded Comput Syst 4(4):779–818

Valmari A (1998) The state explosion problem. Lectures on petri nets I: basic models, Springer 1491:429–528

Yaakoubi H, Haggège J (2022) Modeling of three-tank hybrid system using Mixed Logical Dynamical formalism. In: 5th International conference on advanced systems and emergent technologies, pp 55–60
Zeigler B (1976) Theory of Modelling and Simulation. Wiley

Fernando Barros is a professor at the University of Coimbra. His research interests include theory of modeling and simulation, hybrid systems, and dynamic topology models. Fernando Barros has published more than 80 papers in modeling and simulation. He is associate editor of SIMULATION and vice-president of the Society for Modeling and Simulation.