Proceedings of the 2023 Winter Simulation Conference
C. G. Corlu, S. R. Hunter, H. Lam, B. S. Onggo, J. Shortle, and B. Biller, eds.

πHYFLOW: A MODULAR PROCESS INTERACTION WORLDVIEW

Fernando J. Barros

Department of Informatics Engineering/CISUC
University of Coimbra
3030 Coimbra, PORTUGAL

ABSTRACT

Worldviews play a central role in M&S providing the basic constructs to describe simulation models. Three main worldviews have been defined: event scheduling, activity scanning, and process interaction (PI). The latter has been described in two flavors, one centered in the network of resources and other in the transitory transactions that flow in the network. In this paper we present a new M&S approach based on the πHYFLOW formalism that combines network and transaction PI, while keeping the support for modular and hierarchical models. We demonstrate πHYFLOW expressiveness by representing a hybrid production unit with a variable number of machines subjected to breakdowns. The hybrid model combines a fluid queue describing the work-in-progress, with discrete events modeling machines arrivals, departures, and breakdowns. Arrivals and departures of machines are achieved through modular communication, enabling model composition with other πHYFLOW components.

1 INTRODUCTION

Modeling perspectives or worldviews establish the basic constructs to define simulation models. The choice of the worldview impacts on how easy is to represent the relevant features of a system. Three main worldviews have been defined: event scheduling, activity scanning, and *process interaction* (PI). This last one has been used in two flavors, one centered in the network of resources and other in the temporary transactions that flow in the network. The PI enables a simple and intuitive description of simulation models based on the life cycle of each entity. Contrarily to the event scheduling approach that offers an unstructured perspective of the systems based on a set of events, commonly represented by event graphs (Law 2015; Schruben 1983), PI organizes events by entity and their temporal order of occurrence. The result is a script that is commonly easier to understand and verify than the corresponding event graph.

PI has its origins on the SIMULA language (Dahl et al. 1966), and later supported by other languages, including GPSS (Gordon 1978), and SIMSCRIPT (Russel 1999). Recently, PI has been supported in Java (Healy and Kilgore 1997), Python (Liu 2020), and C++ (Lomow and Baezner 1991; Marzolla 2004). Some simulation languages supporting PI favors the active transaction approach (Gordon 1978; Russel 1999). In this view, transitory entities, like clients, are represented by processes, while permanence entities, like servers, are represented as passive data structures. In the active network view, processes model the permanent resources of the system, like machines, while transactions are represented as passive data that is passed among processes (Henriksen 1981). Although non-modular languages enable modelers to use both PI perspectives (Henriksen 1981), the active network view has been considered a requirement for a modular representation of systems, and it is supported by formalisms like DEVS (Zeigler 1976) for describing discrete event systems, and HYFLOW (Barros 2016), to represent hybrid systems. The support for both PI perspectives while keeping the ability to represent modular and hierarchical models, is a research challenge addressed in this paper.

Although modularity enables 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 modular representations can only describe one event (Barros 2016), forcing, the use of network models to represent systems with two or more events. Systems that involve the coordination of several entities become also more complex requiring, for example, a centralized mechanism to assign competing resources to one entity (Barros 2015).

We present πHYFLOW, a new formalism to represent hierarchical, modular hybrid models that enables, at the base level, the ability to represent several processes, keeping the advantages of PI and providing modular models. πHYFLOW hybrid models combine dense outputs, generalized sampling, and discrete events. Our goal is to combine the simplicity of PI for describing small systems like a server with reneging customers, with hierarchical, and modular constructs that enable a systems-of-systems representation for addressing complexity (Ender et al. 2010). πHYFLOW introduces a new hierarchical and modular worldview, where models can support both network and transactions process interaction perspectives.

To demonstrate πHYFLOW ability for representing hybrid systems we model a production unit with a variable numbers of machines subjected to random breakdowns (Barros 2015). The set of machines is dynamic, and the representation relies on πHYFLOW ability to create/destroy processes at runtime. Machines are hired/fired by a controller and exchanged through πHYFLOW model modular interface. The work-in-process is represented by a fluid queue, being the overall system hybrid combining continuous and discrete flows. A brief overview of πHYFLOW++, a C++ modeling and simulation framework based on the formalism is provided. πHYFLOW++ implementation of the controller process is also described.

This paper is organized as follows. Section 2 introduces the πHYFLOW formalism base and process models. Section 3 describes a modular hybrid production unit, with mobile machines that are subjected to random failures. Section 4 gives a brief description of πHYFLOW++, a C++ implementation of πHYFLOW, and simulation results for the production unit of the previous section. Related work is discussed in Section 5.

2 THE πHYFLOW FORMALISM

The πHYFLOW formalism 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 $\pi_1, ..., \pi_n$. 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. Base model formal description is made in the next section.

Networks enable the composition of base models, or other network models to represent more complex systems. Since both types of models share the same modular interface, they are indistinguishable under composition and coupling operations. πHYFLOW network models are defined in (Barros 2022). In this paper we focus on πHYFLOW base models and on their modular interface.

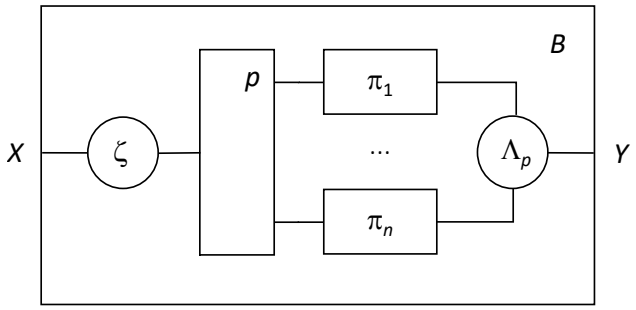

Figure 1: Base model internal structure.

2.1 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, avoiding the complexity of the synchronization problems generally associated with (preemptive) threads (Lee 2006). 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:

A1 $X = X^{\mathrm{c}} \times X^{\mathrm{d}}$ is the set of input flow values, with
A2 X^{c} is the set of continuous input flow values,
A3 X^{d} is the set of discrete input flow values, and
A4 $X^{\varnothing} = X^{\mathrm{c}} \times (X^{\mathrm{d}} \cup \{\varnothing\})$, with $\varnothing$ the null/absence-of value,
A5 $Y = Y^{\mathrm{c}} \times Y^{\mathrm{d}}$ is the set of output flow values, with
A6 Y^{c} is the set of continuous output flow values,
A7 Y^{d} is the set of discrete output flow values, and
A8 $Y^{\varnothing} = Y^{\mathrm{c}} \times (Y^{\mathrm{d}} \cup \{\varnothing\})$,
A9 P is the set of partial shared states (p-states),
A10 P_0 is the set of (valid) initial p-states,
A11 $\zeta : P \times X^{\varnothing} \longrightarrow P$ is the input function,
A12 Π is a set of processes,
A13 $\pi : P \longrightarrow \mathscr{P}(\Pi)$ is the current-processes function, where $\mathscr{P}$ is the power set operator,
A14 $\sigma : \mathscr{P}(\Pi) \longrightarrow \mathscr{P}^*(\Pi)$ is the ranking function, and $\mathscr{P}^*$ is the power sequence operator, constrained to: $\sigma(C | C \subseteq \Pi) = (c_1, \ldots, c_n) \Rightarrow \{c_1, \ldots, c_n\} = C \wedge |\sigma(C)| = |C|$,
A15 for all $p \in P$:
A16 $\Lambda_p : \bigtimes_{i \in (\sigma \circ \pi)(p)} Y_i^{\varnothing} \longrightarrow Y^{\varnothing}$ is the output function associated with p-state p.

Base model modular interface supports hybrid signals with both continuous and discrete flows (A1-A8). P is the set of partial states (p-states) that are shared among all processes (A9), and P_0 represents the set of valid initial p-states (A10). The input function ζ (A11) is responsible for updating the current p-state when the model receives a value either through sampling or event communication. The model can use the set Π of processes (A12). The current set of processes is dynamic, being the current set given by function π (A13). Since processes can access the shared p-state only one can be active at any time. Processes have no entry points, and the representation of model input needs to be stored in the shared p-state so it can be accessed by all processes. The ranking function (A14) σ decides process re-activation order. The output function $\lambda = \{\lambda_p\}$ (A17) 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.

2.2 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 described before. 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. 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 $\varpi \in \Pi$ located in B 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:

B1 Y is the set of output flow values,
B2 Y^{c} is the set of continuous output flow values,
B3 Y^{d} is the set of discrete output flow values,
B4 I is the set of indexes,
B5 P is the set of p-states,
B6 P_0 is the set of (valid) initial p-states,
B7 $\kappa : P \longrightarrow I$ is the index function,
B8 for all $i \in I$:
B9 $\rho_i : P \longrightarrow \mathbb{H}_0^{+\infty}$ is the time-to-input function,
B10 $\omega_i : P \longrightarrow \mathbb{H}_0^{+\infty}$ is the time-to-output function,
B11 $\varkappa_i : S \times P_B \longrightarrow \{\top, \bot\}$ is the condition function,
B12 $\delta_i : S \times P_B \longrightarrow P \times P_B$ is the transition function,
B13 $\Lambda_i^{\mathrm{c}} : S \times P_B \longrightarrow Y^{\mathrm{c}}$ is the continuous flow output function,
B14 $\lambda_i^{\mathrm{d}} : P \times P_B \longrightarrow Y^{\mathrm{d}}$ is the partial discrete flow output function,
B15 $\Lambda_i^{\mathrm{d}} : S \times P_B \longrightarrow Y^{\mathrm{d}} \cup \{\varnothing\}$ is the discrete flow output function defined by:
B16 $$\Lambda_i^{\mathrm{d}}((p,e), p_B) = \begin{cases} \lambda_i^{\mathrm{d}}(p, p_B) & \text{if } e == \omega_i(p), \\ \varnothing & \text{otherwise.} \end{cases}$$
B17 with $S = \{(p, e) | \, p \in P, 0 \leq e \leq \nu_{\kappa(p)}(p)\}$, the state set,
B18 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 (Barros 2016).

A process defines only its output Y (B1-B3), while the *input* is *inferred* from base model p-state, as mentioned before. Processes define a sequence of steps whose indexes are given by set I (B4). A process defines its own (private) p-state, reducing inter process dependency (B5). A process dynamic behavior is ruled by six structured/split functions, being the segments currently active determined by the index function κ (B7). 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 (B9). 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 (B10). The condition function $\{\varkappa_i\}$, checks whether the process has conditions to run (B11). 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 (B12). Function $\{\Lambda_i^{\mathrm{c}}\}$ specifies process continuous output flow (B13), and $\{\Lambda_i^{\mathrm{d}}\}$ specifies process discrete output flow (B15), which is based on the partial discrete flow output function (B14). In the next section we provide the πHYFLOW base model of a hybrid production unit that has a variable number of machines subjected to breakdowns.

3 PRODUCTION UNIT

We consider a production unit (PU) with several machines subjected to breakdowns (Horton et al. 1998). The work in process (WIP) is represented by a fluid queue, with input rate a and nominal output rate given by $b|working|$, where b is the nominal machine processing rate, and *working* is the number of machines currently working. Machines are subjected to random breakdowns. We analyze here a variation to this system by considering that machines can be dynamically added/removed. The decision is based on the WIP value, and it is made by a controller process. Machines are exchanged through PU input/output interface, making it modular, enabling PU composition with other models.

3.1 Production Unit Base Model

The base model of a production unit D is defined by:

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

where:

C1 $X = \{\} \times \{\text{arrival}\}$,
C2 $Y = (\{\text{wip}\} \times \mathbb{R}_0^+) \times \{\text{hire}, \text{exit}\}$,
C3 $P = \{(arrival, machines, working) | \, arrival \in \{\top, \bot\}, machines \subset \mathbb{N}, working \subset \mathbb{N}\}$,
C4 $P_0 = \{(arrival = \bot, machines = \{0\}, working = \{\})\}$,
C5 $\zeta((arrival, machines, working), (\varnothing, \text{arrival})) = (\top, machines, working)$,
C6 $\Pi = \{\text{controller}, \text{wip}\} \cup \mathbb{N}$,
C7 $\pi(arrival, machines, working) = \{\text{controller}, \text{wip}\} \cup machines$,
C8 $\sigma(arrival, \{\ldots, m_{k-1}, m_k, m_{k+1}, \ldots\}, working) = (\text{controller}, \text{wip}, \ldots, m_{k-1}, m_k, m_{k+1}, \ldots)$, such that
C9 $\ldots < m_{k-1} < m_k < m_{k+1} < \ldots$,
C10 $\Lambda_p(\ldots, (\Lambda^{\text{c}}_{i,c_k}(s_{c_k}, p), \Lambda^{\text{d}}_{i,c_k}(s_{c_k}, p)), \ldots) =$
C11 $((y^{\text{c}} = \Lambda^{\text{c}}_{i,c_k}(s_{c_k}, p) | \, c_k \in C \wedge y^{\text{c}} \neq \varnothing), (y^{\text{d}} = \Lambda^{\text{d}}_{i,c_k}(s_{c_k}, p) | \, c_k \in C \wedge y^{\text{d}} \neq \varnothing))$,
C12 with $i = \kappa_{c_k}(p_{c_k})$, and $C = (\sigma \circ \pi)(arrival, machines, working) = (\ldots, c_k, \ldots)$.

The arrival of a machine is signaled at input port "arrival" (C1). The WIP can be sampled at continuous output port "wip". Requests for machines are made through discrete output port "hire", while machines are sent through port "exit" (C2). PU p-states include information on machine arrival, the machines in the PU, and the working (not broken) machines (C3). The PU starts with machine 0 (C4). Machine arrival is signaled in C5 that sets $arrival = \top$. The actual creation of a machine is made by the controller that is described below. The set of processes include "controller", "wip", and a dynamic set of machines (C7). Process ranking function is defined in C8. The output is defined in C10, and it gathers all non-null outputs from the current set of processes. As described next, the continuous output is produced by the "wip" process, while the discrete flow is produced by machines when they leave the PU.

3.2 The Machine Process

Machines process WIP at a (piecewise) constant rate. They are subjected to random breakdowns at rate up^{-1} and repaired at rate $down^{-1}$, both exponential random distributions. The model of a machine process m located in PU D is defined by:

$$M_m^D = (Y, I, P, P_0, \kappa, \{\rho_i\}, \{\omega_i\}, \{\varkappa_i\}, \{\delta_i\}, \{\Lambda_i^{\text{c}}\}, \{\lambda_i^{\text{d}}\}),$$

where:

D1 $Y = \{\} \times \{\text{exit}\}$,
D2 $I = \{0,..,4\}$,
D3 $P = \mathbb{N} \times \mathbb{R}_0^+$,
D4 $P_0 = \{(ix = 0, time = 0)\}$,
D5 $\kappa(ix, time) = ix$,
D6 $\rho_i(ix, time) = \infty$,
D7 $\omega_i(ix, time) = time$,
D8 $\varkappa_{0,2,3,4}(((ix, time).e), (arrival, machines, working)) = \perp$,
D9 $\delta_0(((ix, time), e), (arrival, machines, working)) =$
D10 $\quad((1, \exp(up)), (arrival, machines, working \cup \{m\}))$,
D11 $\varkappa_1(((ix, time), e), (arrival, machines, working)) = m \notin working$,
D12 $\delta_1(((ix, time), e), (arrival, machines, working)) =$
D13 $\quad((2, \exp(down)), (arrival, machines, working)),$ if $m \in working$,
D14 $\quad((3, 0), (arrival, machines, working)),$ otherwise,
D15 $\delta_2(((ix, time), e), (arrival, machines, working)) =$
D16 $\quad((1, \exp(up), (arrival, machines, working))$,
D17 $\delta_3(((ix, time), e), (arrival, machines, working)) =$
D18 $\quad(4, \infty), (arrival, machines \setminus \{m\}, working))$,
D19 $\Lambda_i^{\text{c}}(((ix, time), e), (arrival, machines, working)) = \varnothing$,
D20 $\lambda_{0,1,2,4}^{\text{d}}((ix, time), (arrival, machines, working)) = \varnothing$,
D21 $\lambda_3^{\text{d}}((ix, time), (arrival, machines, working)) = \text{exit}$.

Machines produce the value "exit" (D1), when they leave the PU. A machine has an index ix and time to next transition $time$ (D3). When a machine starts working it places its identifier m in the set of *working* machines (D9-D10). When working, the machine checks whether it has been removed from the working-set (D11). In this case, it exits the PU (D14, D21), and it deactivates removing itself from the set of machines (D18). After finishing work the machine breaks and undergoes a repair (D13). The cycle repeats when the machine goes back to $ix = 1$ (D16). We present next the model of a fluid queue process.

3.3 The Fluid Queue Process

Fluid queues (FQs) can represent continuous values, like WIP, that are processed at a piecewise constant rate (PCR). We consider a FQ with constant input rate IN and nominal PCR output rate $out = OUT \cdot |working|$, where OUT is the nominal processing rate of one machine, and *working* is the set of currently working machines. FQ content cannot be negative, limiting the actual FQ filling rate. When the FQ value is zero the filling rate remains zero while $out > IN$. The model of a fluid queue process φ located in PU D is defined by:

$$M_\varphi^D = (Y, I, P, P_0, \kappa, \{\rho_i\}, \{\omega_i\}, \{\varkappa_i\}, \{\delta_i\}, \{\Lambda_i^{\text{c}}\}, \{\lambda_i^{\text{d}}\}),$$

where:

E1 $Y = (\{\text{wip}\} \times \mathbb{R}_0^+) \times \{\}$,
E2 $I = \{0\}$,
E3 $P = \mathbb{R} \times \mathbb{R}_0^+ \times \mathbb{R}_0^+$,
E4 $P_0 = \{(flow = 0, vol = V_0, time = 0)\}$,
E5 $\kappa(flow, vol, time) = 0$,
E6 $\rho_0(flow, vol, time) = \infty$,

E7 $\omega_0(flow, vol, time) = time,$

E8 $\varkappa_0(((flow, vol, time), e), (arrival, machines, working)) =$

E9 $\quad flow \neq rate(vol + e_{\text{std}} \cdot flow \cdot |working|, IN, OUT)$, with

E10 $\quad rate(v, w, in, out) =$

E11 $\quad\quad 0, \qquad$ if $v == 0 \wedge in - w \cdot out < 0,$

E12 $\quad\quad in - w \cdot out, \qquad$ otherwise,

E13 $\delta_0(((flow, vol, time), e), (arrival, machines, working)) =$

E14 $\quad ((flow', vol', \infty), (arrival, machines, working)), \qquad$ if $flow' \geq 0,$

E15 $\quad ((flow', vol', -\frac{vol'}{flow'}), (arrival, machines, working)), \qquad$ otherwise,

E16 $\quad$ with $vol' = vol + e_{\text{std}} \cdot flow \cdot |working|$, and $flow' = rate(vol', |working|, IN, OUT)$

E17 $\Lambda_0^{\text{c}}(((flow, vol, time), e), (arrival, machines, working)) = (\text{wip}, vol + e_{\text{std}} \cdot flow \cdot |working|),$

E18 $\lambda_0^{\text{d}}((flow, vol, time), (arrival, machines, working)) = \varnothing.$

The queue has a continuous flow output port "wip" where queue current value is available for sampling (E1). Queue p-state stores volume, flow rate and time to transition (E3). The queue process loops on a single transition that is reschedule when there is a change in the flow rate (E9). The flow rate is defined in E10-E12 that considers the constraint: $tank - volume \geq 0$. The time to transition depends on the flow rate. If the flow is larger or equal to zero, the process will never undergo a transition since there is no upper bound on queue volume (E14). Otherwise, the transition occurs when queue volume reaches zero (E15). Queue current volume is computed in E17 and can be obtained through sampling. This operation can be made by an external component or by any process belonging to the PU.

3.4 The Controller Process

The PU can hire/fire machines for trying to keep WIP within certain bounds. A simple control strategy, employed here, is to sample WIP level at a regular time interval *SI*, and take a decision based on the sampled value. A model of controller process c located in PU D is defined by:

$$M_c^D = (Y, I, P, P_0, \kappa, \{\rho_i\}, \{\omega_i\}, \{\varkappa_i\}, \{\delta_i\}, \{\Lambda_i^{\text{c}}\}, \{\lambda_i^{\text{d}}\}),$$

where:

F1 $Y = \{\} \times \{\text{hire}\},$

F2 $I = \{0, 1, 2\},$

F3 $P = I \times \mathbb{N} \times \mathbb{R}_0^+,$

F4 $P_0 = \{(ix = 0, ID = 1, time = SI)\},$

F5 $\kappa(ix, ID, time) = ix,$

F6 $\rho_i(ix, ID, time) = \infty,$

F7 $\omega_i(ix, ID, time) = time,$

F8 $\varkappa_{0,1}(((ix, ID, time), e), (arrival, machines, working)) = \perp,$

F9 $\varkappa_2(((ix, ID, time), e), (arrival, machines, working)) = arrival,$

F10 $\delta_0(((ix, ID, time), e), (arrival, machines, \{w_0, w_1, \ldots, w_n\}, ID)) =$

F11 $\quad ((1, ID, 0), (arrival, machines, \{w_0, w_1, \ldots, w_n\})), \qquad$ if $vol \geq HIGH,$

F12 $\quad ((0, ID, SI), (arrival, machines, \{w_1, \ldots, w_n\})), \qquad$ if $vol < LOW \wedge |working| > 1,$

F13 $\quad ((0, ID, SI), (arrival, machines, \{w_0, w_1, \ldots, w_n\})), \qquad$ otherwise,

F14 $\quad$ with $vol = \Lambda_{i,wip}^{\text{c}}(s_{wip}, (arrival, machines, \{w_0, w_1, \ldots, w_n\}))$, and $i = \kappa_{wip}(p_{wip}),$

F15 $\delta_1(((ix, ID, time), e), (arrival, machines, working)) =$

F16 $\quad((2, ID, \infty), (arrival, machines, working)),$

F17 $\quad\delta_2(((ix, ID, time), e), (arrival, machines, working)) =$

F18 $\quad((0, ID+1, SI), (\perp, machines \cup \{ID\}, working)),$

F19 $\quad\Lambda_i^c(((ix, ID, time), e), (arrival, machines, working)) = \varnothing,$

F20 $\quad\lambda_{0,2}^d((ix, ID, time), (arrival, machines, working)) = \varnothing,$

F21 $\quad\lambda_1^d((ix, ID, time), (arrival, machines, working)) = \text{hire}.$

After sampling, the controller checks queue volume (F10-F13). If the volume is above the threshold *HIGH* (F11), the controller sends a hire request (F21). A machine is fired (removed from the working-set) if the volume is below threshold *LOW*, and if there are more than one working machine (F12). Otherwise, the controller will wait for interval *SI* and samples again (F13). After sending a hire signal the controller waits for an ∞ interval (F16), until a machine arrives (F9). Upon arrival, a machine is created with the identifier *ID* (F18).

πHYFLOW ability to dynamically modify the set of active processes provides a framework that enables the combination of active transaction and the active network PI, while keeping the support for modular and hierarchical models. As shown in this model, the concepts of resource and transaction becomes blurred, being the flexibility of πHYFLOW enabled by the possibility to combine permanent and temporary processes in the same base model. Next section gives an overview of formalism implementation in the C++ language. It also presents the controller process, and simulation results.

4 πHYFLOW++ MODELING & SIMULATION ENVIRONMENT

The πHYFLOW++ is a modeling and simulation framework based on the πHYFLOW formalism. It was implemented in the C++20 language (Standard 2023) using the Visual Studio 2022 C++ compiler. πHYFLOW processes are represented by C++ coroutines that enable an efficient and scalable solution due to coroutines non-preemptive behavior. To simplify model development, the implementation defines implicitly same formalism operators. πHYFLOW input function is defined by considering a set of buffers that receive all input discrete input flows. The condition and transition functions are combined in the same operator. The index function is also implicit in C++ language sequence of operations (program counter).

4.1 πHYFLOW++ Controller

We give an example of πHYFLOW++ application by presenting the implementation, in Listing 1, of the controller processes described in the last section. The buffer containing machine arrivals is defined in line 4. The controller waits "SI" times units (line 6), and then it samples the "wip" process (line 7). The hire decision is made in line 8. The request for a machine is made in line 9. The controller waits for a machine arrival in line 10. The buffer is cleared in line 11, so the controller can receive new machines. A machine is created in line 12. The fire decision is taken in line 14, and the machine is removed from the working-set in line 15.

The implementation automatizes many formalism operators, and the result is a C++ script that is intuitive and easy to follow. The implementation also supports πHYFLOW network models enabling the composition of models (Barros 2022). The production unit can be connected, for example, to a machine center that is responsible to coordinate machines among production units in a larger organization. Simulation results are presented in the next section.

4.2 Simulation Results

For simulation we consider a production unit with the parameters given in Table 1. In addition, we model the response time for a machine request as an exponential distribution with a mean 5.

```
sim_void controller(const std::string_view name, double SI, double LOW, double HIGH) {
   sim_start(name);
   auto ID {1};
   auto& arrival = buffers("arrival");
   while (true) {
      sim_wait duration(SI);
      auto volume = sample<double>("wip");
      if (volume > HIGH) {
         sim_wait out("hire", vol);
         sim_wait until([&arrival] {return arrival.any();});
         arrival.clear();
         machine(ID++);
      }
      else if (volume < LOW && working.size() > 1)
         working.pop();
   }
   sim_end;
}
```

Listing 1: πHYFLOW++ controller process.

Table 1: Simulation parameters.

SI	V_0	*IN*	*OUT*	*LOW*	*HIGH*	*UP*	*DOWN*
5.	90.	30.	20.	45.	120.	6.	3.

The WIP is represented in Figure 2 for a simulation interval of 200 units. Given that queue fill rate is piecewise constant the queue volume is piecewise linear. Changes in queue volume depend on the number of working machines (Figure 4) that depend also on the hire/fire commands affecting the total number of machines (Figure 3).

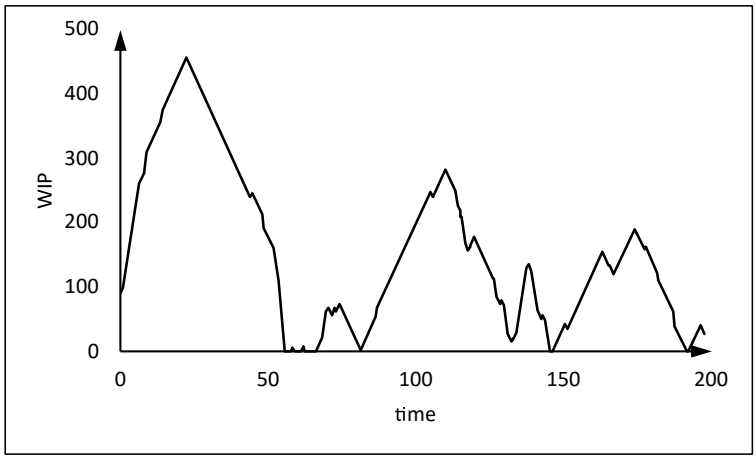

Figure 2: WIP in production unit.

As it can also be observed in Figure 3, hire/fire commands are made one at a time, and sequentially, after the last command being fulfilled. We can also observe than at least one machine is always present. Different control strategies could also be easily described in the framework. Given πHYFLOW ability to represent exactly continuous flow, simulation becomes mainly event driven, making it very efficient.

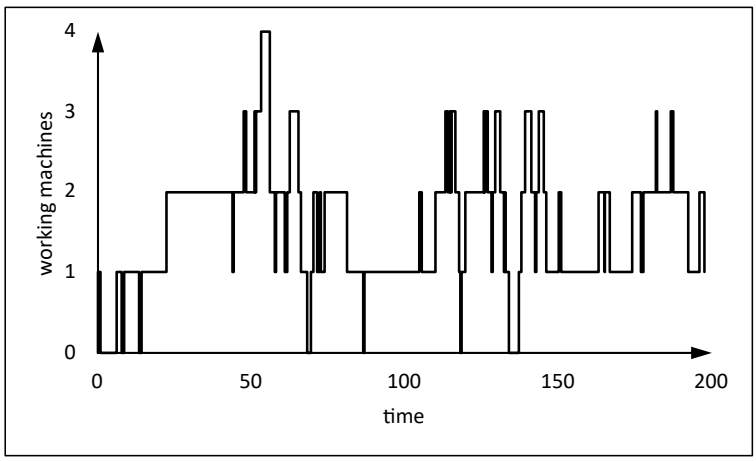

Figure 3: Number of working machines.

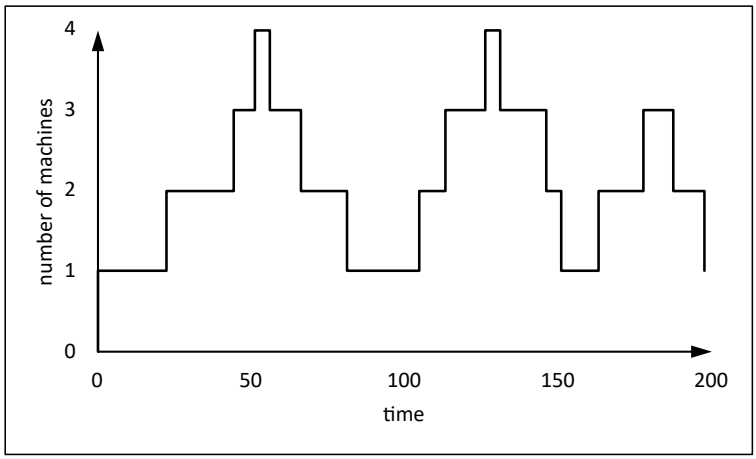

Figure 4: Number of machines.

5 RELATED WORK

The process interaction worldview (PI) was introduced by the SIMULA language (Dahl et al. 1966). Currently, most PI descriptions do not distinguish between transaction and network perspectives (Banks et al. 2010; Pidd 1997; Law 2015), and do not discuss the hierarchical and modular model representation (Banks et al. 2010; Pidd 1997). The identification of transaction and network PI perspectives was made by (Henriksen 1981). A formal specification of PI was proposed in (Zeigler 1976). However, this work does not support dynamic sets of processes, making it not possible to provide, for example, a solution to represent a simple delay, that would require an unbounded set of processes. This approach also provides a limited view on process conditional waiting. A general "wait until" operator was proposed in (Franta 1977), and later used by (Cota and Sargent 1992) and in the SLX language (Henriksen 1997). However, these approaches do not address the representation of hierarchical and modular models as defined in πHYFLOW.

The network view is supported by tools like ExtendSim (Krahl 2003) that also enables the representation of hierarchical and modular models, but limited to static network topologies. This framework, however, does not support processes, making it more similar to the HYFLOW formalism (Barros 2016).

The network view is also supported by the SSF framework (Nicol et al. 2003). However, SSF does not provide support for modularity as it can be found in the simple producer/consumer system (Banks et al.

2010, pg. 146-147). Channels are used to signal that a value has arrived at a shared queue. This queue, however, is represented as a global variable that can be accessed by any class, breaking modularity. Given SSF constraint to allow each class for only represent one process make it difficult to support complex models. In πHYFLOW processes can specialize in different ports, whereas a single process solution would require the single process to react to all possible inputs, yielding in general to more convoluted models.

Modular formalisms like HYFLOW (Barros 2016), or DEVS (Zeigler 1976), impose a partitioning into fine-grained components and implicitly supporting only PI network perspective. Since they do not have shared states, a coordination mechanism is also required for activities that require resources located in different models. Taking for example the representation of Fluid Stochastic Petri Nets (FSPNs) (Horton et al. 1998), the HYFLOW description requires a central manager to coordinate tokens and transitions (Barros 2015). On the contrary, πHYFLOW processes have a direct access to the shared set of tokens making it very simple to coordinate transitions using conditions. πHYFLOW enables modelers to take the advantages of shared memory operators while keeping the ability to more complex network models. We consider that the decision on what a base model is should be left to the modeler and not constrained by the formalism.

The ability to modify model topology supported by the HYFLOW formalism is kept in the πHYFLOW. However, changing the topology of a network of modular components is more complex than add/remove processes. Taking, for example, a single delay, the corresponding HYFLOW is a network where each arriving transaction is represented by a new atomic model that needs to be connected to the network. In a similar manner when the delay ends, the model needs to coordinate with the network executive that removes it from simulation (Barros 1998). On the contrary, dynamic creation and destruction of processes become very simple operations, as show in Sections 3-4.

6 CONCLUSION

The πHYFLOW formalism introduces a new modeling approach that combines process interaction on both active network and active transaction perspectives, while also supporting modular and hierarchical models. The dynamic creation/destruction of processes enables a flexible framework to represent models with an unbounded number of dynamic processes that act like transitions, while supporting permanent ones that act like servers. Processes can represent continuous flows, generalized sampling, and discrete events, supporting thus a representation of hybrid systems. πHYFLOW base models are modular, being communication achieved through a well-defined input/output interface. The C++ heap-based coroutines implementation of processes enables an efficient solution that can handle a large number of processes.

REFERENCES

Banks, J., J. Carson, B. Nelson, and D. Nicol. 2010. *Discrete-Event System Simulation*. 5th ed. Pearson.

Barros, F. 1998. "Hierarchical Testing of Dynamic Structure Systems". *SIMULATION* 81(5):381–393.

Barros, F. 2015. "A Modular Representation of Fluid Stochastic Petri Nets". In *Symposium on Theory of Modeling and Simulation*.

Barros, F. 2016. "On the Representation of Time in Modeling & Simulation". In *Winter Simulation Conference*, edited by T. M. Roeder, P. I. Frazier, R. Szechtman, E. Zhou, T. Huschka, and S. E. Chick, 1571–1582.

Barros, F. 2022. "High-Fidelity Modeling & Co-Simulation with πHyFlow". In *Software Engineering Formal Methods Collocated Workshops*, Lecture Notes in Computer Science, Vol. 13765, 269–285.

Cota, B., and R. Sargent. 1992. "A Modification of the Process Interaction World View". *ACM Transactions on Modeling and Computer Simulation* 2(2):109–129.

Dahl, O.-J., B. Myhrhaug, and K. Nygaard. 1966. "SIMULA - An ALGOL-Based Simulation Language". *Communications of the ACM* 9(9):671–678.

Ender, T., R. Leurck, B. Weaver, P. Miceli, W. Blair, P. West, and D. Mavris. 2010. "Systems-of-Systems Analysis of Ballistic Missile Defense Architecture Effectiveness Through Surrogate Modeling and Simulation". *IEEE Systems Journal* 4(2):156–166.

Franta, W. 1977. *The Process View of Simulation*. North-Holland.

Gordon, G. 1978. *System Simulation*. 2nd ed. Prentice-Hall, Englewood Cliffs, New Jersey.

Healy, K., and R. Kilgore. 1997. "Silk: A Java-based Process Simulation Language". In *Winter Simulation Conference*, edited by S. Andradóttir, K. J. Healy, D. H. Withers, and B. L. Nelson, 475–482.

Henriksen, J. 1981. "GPSS - Finding the Apropriate World-View". In *Winter Simulation Conference*, edited by T. I. Ören, C. Delfosse, and C. Shu, 505–516.

Henriksen, J. 1997. "An Introduction to SLX". In *Winter Simulation Conference*, edited by S. Andradóttir, K. J. Healy, D. H. Withers, and B. L. Nelson, 559–565.

Horton, G., V. Kulkarni, D. Nicol, and K. Trivdedi. 1998. "Fluid Stochastic Petri Nets: Theory, Applications, and Solution Techniques". *European Journal of Operational Research* 105:184–201.

Krahl, D. 2003. "Extend: An Interactive and Simulation Tool". In *Winter Simulation Conference*, edited by S. Chick, P. Shnchez, D. Ferrin, and D. Morrice, 188–196.

Law, A. 2015. *Simulation Modeling and Analysis*. 5nd ed. McGraw-Hill.

Lee, E. 2006. "The Problem with Threads". *Computer* 39(5):33–42.

Liu, J. 2020. "Simulus: Easy Breezy Simulation in Python". In *Winter Simulation Conference*, edited by K.-H. G. Bae, B. Feng, S. Kim, S. Lazarova-Molnar, Z. Zheng, T. Roeder, and R. Thiesing, 2329–2340.

Lomow, G., and D. Baezner. 1991. "A Tutorial Introduction to Object-Oriented Simulation and SIM++". In *Winter Simulation Conference*, edited by B. Nelson, W. Kelton, and G. Clark, 157–163.

Marzolla, M. 2004. "libcppsim: A Simula-like, Portable Process-Oriented Simulation Library in C++". In *18th European Simulation Multiconference*, 222–227.

Nicol, D., J. Liu, M. Liljenstam, and G. Yan. 2003. "Simulation of Large Scale Networks using SSF". In *Winter Simulation Conference*, edited by S. Chick, P. Shnchez, D. Ferrin, and D. Morrice, 650–657.

Pidd, M. 1997. *Computer Simulation in Management Science*. Wiley.

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

Schruben, L. 1983. "Simulation Modeling with Event Graphs". *Communications of the ACM* 26(11):957–963.

C++20 Standard . 2023. "The Current Revision of the C++ Standard". https://en.cppreference.com/w/cpp/20. Accessed: June 16th, 2023.

Zeigler, B. 1976. *Theory of Modelling and Simulation*. Wiley.

AUTHOR BIOGRAPHY

FERNANDO J. 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. His email address is barros@dei.uc.pt.