Distributed Colored Petri Net Simulation – Synchronized Transitions Based on Event Streaming*

Laif-Oke Clasen[1], Justus Middendorf[1], Leo Grimme[1], Leven Wichelmann[1], Finn Bolinius[1], Sam Knoop[1] and Daniel Moldt[1]

[1]*University of Hamburg, Faculty of Mathematics, Informatics and Natural Sciences, Department of Informatics,* *http://www.paose.de*

Abstract

Simulation and verification are key to analyzing systems and assessing behavioral correctness. Coloured Petri nets unify concurrency, communication, and resource constraints; growing size and fidelity demand scalable, robust simulation.

Single-machine simulation is constrained by fixed memory and processing, limiting model size and detail in practice. This work investigates distributing simulation across platforms to overcome these limits while preserving coloured Petri net formal semantics via a suitable synchronization concept.

This work introduces a cross-platform, rendezvous-based synchronization concept that preserves formal firing semantics by coordinating cross-partition interactions. Using a constructivist prototyping methodology, it realizes the concept in Renew as a proof-of-concept distributed event-streaming simulator for Distributed Colored Petri Nets (Dcpns). Models are distributable; firing the local part of a transition is synchronized with the distributed firings for a specific binding.

The concept enables scalable, robust distributed simulation beyond single-machine execution while maintaining coherent firing semantics across partitions. This work extends Renew with distributed simulation capabilities and provides a practical, formally rigorous basis for analyzing increasingly complex coloured Petri net models beyond the limits of single-machine simulation.

Keywords

Distributed Colored Petri Nets, Coloured Petri Nets, Synchronous Channels, Rendezvous Synchronization, Partially Globally Synchronous Locally Synchronous, Distributed Simulation, Distributed Systems, Event Streaming

1. Introduction

Petri nets provide a rigorous formal basis for modeling and analyzing concurrent and distributed behavior, including P/T nets [1] and Coloured Petri Nets (CPNs) [2, 3]. Their graphical yet mathematically defined structure explicitly represents causality, concurrency, and resource constraints, with applications in organizational process modeling [4] and the engineering of distributed and intelligent systems [5]. Simulation is central because it enables executable validation, exploratory what-if studies, and performance-oriented experimentation prior to deployment.

As modeled-system complexity increases, CPN models grow in structural size (places, transitions, arc inscriptions) and behavioral complexity (token multiplicities, color domains, binding combinations). In realistic settings, single-machine simulation is constrained by finite resources (memory, processing capacity), motivating distributed simulation that exploits multiple computing resources while retaining a well-defined semantic interpretation. [6, 7, 8]

This work investigates distributing CPN simulation across multiple computing nodes without weakening firing semantics. We propose a cross-partition synchronization concept based on rendezvous-style coordination for Distributed Colored Petri Nets (Dcpns) to handle interactions spanning partition boundaries: firings with non-local effects are jointly coordinated, blocking interactions so distributed execution does not introduce behaviors disallowed under the reference semantics of the original unpartitioned model. The scope of cross-partition interaction and the semantic equivalence target (e.g., with

PNSE'26, International Workshop on Petri Nets and Software Engineering, 2026

* Supported by participants of our teaching project classes and student theses.

0009-0008-9653-0650 (L. Clasen); 0009-0009-7299-5354 (J. Middendorf); 0009-0000-5237-9110 (D. Moldt)

respect to firing traces and bindings under the chosen CPN execution semantics) are stated explicitly in subsequent sections.

Following a constructivist prototyping methodology [9, 10, 11, 12], we implement a proof-of-concept distributed simulator for DCPNs in RENEW. The prototype operationalizes rendezvous coordination and provides an executable basis for validating semantic faithfulness against the reference execution model under explicitly stated partitioning and communication assumptions.

This work begins with the foundations (Section 2) and the problem statement (Section 3). Next, the conceptual solution is introduced (Section 4), after which DISTRIBUTED COLORED PETRI NETS (DCPNS) are addressed (Section 5). The results are then presented (Section 6), followed by a discussion (Section 7) and consideration of related prior work (Section 8). Finally, the work concludes in Section 9.

2. Foundations

2.1. Synchronization

Distributed systems require explicit coordination to relate the progress of independently executing components across platform boundaries. [13] Synchronization defines agreed interaction points and ensures consistent state transitions across boundaries; three paradigms coordinate concurrent and distributed components: memory, message, and rendezvous synchronization. [14]

Memory-based synchronization [15] relies on shared storage, from shared variables and data structures to distributed shared memory; reads and writes are decoupled—writes do not require concurrent readers, and readers may observe updates later—so coordination depends on the memory model's visibility and ordering guarantees, while mutual exclusion, atomic read–modify–write behavior, and barrier-style agreement require explicit synchronization primitives (e.g., locks, barriers, atomic operations) with stated guarantees.

Message-based synchronization [16] uses explicit message exchange; transmission and processing are temporally decoupled, so at send time the sender has no immediate guarantee that the receiver has received or processed a message, and confirming reception, aligning control points, or enforcing a joint step typically requires protocol-level extensions such as acknowledgements, handshakes, or barrier protocols.

Rendezvous synchronization [17, 18, 19] provides bidirectional, synchronous interaction via an atomic joint step that occurs only when both partners are simultaneously ready to participate. [14] This tight temporal coupling supports coordinated actions when the intended semantics requires an indivisible interaction step rather than eventual observation via shared state or delayed processing via asynchronous messages.

2.2. Synchronous Channels

Synchronous Channels in the context of Petri nets enable rendezvous synchronisation 2.1 via channels. Concretely, a transition is equipped by inscribing it with a channel signature, which allows it to communicate and synchronize with other transitions carrying a matching signature; if inscribed with a synchronous channel, it must find a matching partner and can only fire jointly with that partner.

A channel signature comprises type, identifier, and parameters; all three must match for successful synchronization. Exactly two types exist, and matching requires the types to differ: *downlink* and *uplink*. The *downlink* is active and initiates synchronization, whereas the *uplink* is passive. Notation varies: Christensen and Hansen [20] use *!?* and *?!* for channels for coloured Petri nets, while for reference nets Kummer [21] uses net references for the downlink (e.g., *this* for a single net) and a lack thereof for the uplink. The identifier is generally the channel name, which must be equal for transitions to match. Parameters exchange information; e.g., an unassigned variable in one synchronizing transition is assigned via a parameter of the other.

For example, a *downlink* signature is `this:ch(1,2,3)`, with matching *uplink* signature `:ch(x,y,z)`; upon synchronization, `x,y,z` are bound to `1,2,3` respectively.

2.3. Coloured Petri Nets

Coloured Petri nets (CPN) are a high-level Petri net formalism that extends classical Petri nets. By allowing tokens to carry data values, called colors, CPNs enable the modelling of complex systems with a more compact representation, compared to P/T nets, where tokens are indistinguishable. A CPN has a set of color sets which define the types of colors that tokens can carry. Each place in a CPN is mapped to a specific color set, meaning that the tokens in that place can be colors from that set, while the marking (of that place) is a multiset over the color set of that place. The set of arcs is also mapped to a color set, specifically the color set of the place the arc is connected to. Coloured Petri nets also have a set a variables which are typed with color sets and can be used in expressions on arcs or transitions. Additionally, transitions can now have guards, which are boolean expressions over the transition's free variables, that must evaluate to true for the transition to be enabled. This introduction follows the definition of coloured Petri nets by Jensen et al. [2, 3].

Unlike interleaving models, where concurrent events are represented as occuring in some sequential order, coloured Petri nets inherently support true concurrency. As such, we use a partial order semantics [22, 23]. This semantics portrays the causal relationships between events, while also allowing for concurrent events to be represented as such.

2.4. Event Streaming

Event streaming is a log-based abstraction for communication and data management in distributed systems: components exchange time-ordered events as continuous streams rather than via synchronous calls or shared state[24]; state changes are externalized into persistent, append-only event logs, and system state is derived deterministically by processing these logs, aligning with event-driven architectures emphasizing asynchronous communication and loose coupling [25]. Producers append events describing facts or state transitions to streams without depending on consumers; consumers subscribe and process events in stream order while maintaining their position, typically as an offset—this ordering enforces deterministic processing semantics, while event handling is determined by the consumer implementation. The resulting decoupling improves modularity, allows independent evolution, enables concurrent consumption, recovery by resuming from the last recorded position, and replay of historical events for analysis or debugging; log persistence thus contributes to robustness and fault tolerance. Topics structure event streams and form the primary producer–consumer interface; continuous ordered streaming decouples components temporally and spatially, reducing direct dependencies.[25]

Apache Kafka [26] realizes these abstractions and offers scalability, high availability, and high data throughput; unlike traditional message queues, Kafka does not delete message-records after receipt and processing, enabling replay and reprocessing [27]. Kafka implements streams as partitioned and replicated commit logs and runs as a cluster of broker nodes serving as the central contact point: brokers receive and store events (e.g., key-value tuples in formats such as JSON); topics may be partitioned across brokers for horizontal scalability and partitions replicated across nodes for fault tolerance and high availability. Consumers subscribe to topics, track offsets, and may form consumer groups for parallel processing and automatic load balancing.[26, 27]

2.5. Renew

Renew [28] is an advanced open-source tool for modelling, simulation, and analysis of various Petri net formalisms, developed by the Algorithms, Randomization, and Theory (ART) research group at the University of Hamburg. Originally focused on Reference Nets, Renew has evolved to support additional approaches, including Agent Interaction Protocol Diagrams (Agent Interaction Protocol Diagrams), Business Process Modeling Notation (BPMN), and distributed place/transition (P/T) nets with synchronous channels [29, 30]. Implemented in Java with a plugin-based architecture, Renew has been enhanced with the Java Platform Module System (JPMS) to improve modularity and maintainability [31].

The reference net formalism is the most prominent, combining nets-in-nets with reference semantics and Java as a labeling language. A cloud-native plugin [32] exposes HTTP endpoints through Java Spring, enabling remote simulation control across networks.

3. Problem Description

In computer science, simulations analyze real or planned systems under controlled conditions, without requiring production infrastructure or user interactions. They enable risk-free scenario testing, such as studying error behavior or peak loads, and systematic parameter variation, such as scheduling, routing, or resource allocation. This approach often reduces costs by avoiding the need for dedicated hardware, laboratories, or field studies. Furthermore, reproducibility is achievable when start states, inputs, and external influencing factors are controlled. In distributed execution environments, achieving reproducibility typically requires additional, well-defined execution and synchronization rules.

Beyond analysis, simulations in suitable architectures can be integrated into operational system management by (i) simulation results influencing real decisions or actions, or (ii) observations from reality continuously updating the simulation state. A common distinction is between a digital shadow, with one-way coupling, and a digital twin, with bidirectional coupling. [33] In the context of these definitions, the classification serves as motivation. The following synchronization issues are considered without real-time coupling.

A simulation is called discrete if its state changes only at discrete points in time. [34] A discrete event simulation occurs when explicit events trigger state changes. [35, 34]

Petri nets are a widely used modeling technique in this context and can be classified, depending on expressiveness and extensions, as place/transition nets (P/T nets), coloured Petri nets, timed Petri nets, and stochastic Petri nets. In a discrete-event simulation of coloured Petri nets, state changes are modeled by firing enabled transitions, each of which induces a discrete marking change. This contribution focuses on discrete (event-driven) simulation based on the fundamentals introduced in Section 2.3. Non-determinism arises because several transitions may be firable in a given state, and their firing sequence is not determined a priori (conflict/interleaving non-determinism). In the distributed setting, execution must reproduce the well-defined reference semantics of the non-distributed simulation. Distribution must not introduce additional non-determinism due to scheduling or communication beyond the model-inherent choice between concurrently firable transitions. Typical modeling objects include communication and network protocols [36, 37], distributed systems [38], concurrent software [39], workflow and business process models [40, 41], production and logistics systems [42, 43], system architectures [44, 5], and interface behavior [45, 46].

As real systems grow in lifetime and complexity, associated models often grow as well (e.g., model size, event density, or the complexity of color data). Two scaling approaches are typically considered: (i) vertical scaling through more powerful individual resources and (ii) horizontal scaling through distribution across multiple resources. Since vertical-scaling costs often increase disproportionately, especially at higher performance levels, it is not practical in all scenarios. [47, 48, 49] Given the growing complexity of the models and the costs associated with vertical scaling, this article focuses on horizontal scaling, examining models that are practically impossible to simulate on a single simulator. Therefore, a series of simulation runs can be distributed across different computational elements only if sufficient capacity is available for multiple independent, concurrent simulations.

Horizontal scaling is therefore particularly relevant for large, detailed models. It requires additional simulator instances on other physical machines and synchronization to ensure the overall execution remains a coherent simulation. Distributed simulations use multiple processes running asynchronously and in parallel. Communication typically occurs via messages [34], and components are often connected via networks [7]. Motivations include (i) geographically distributed information, (ii) integration of heterogeneous simulators, and (iii) potential performance gains through parallel execution. [7, 50] This paper focuses on feasibility, particularly partitioning and synchronization mechanisms for executing large, detailed models across multiple simulator instances without substantially increasing modeling

effort.

We study distributed simulation of coloured Petri nets (CPNs) (Section 2.3) where a single CPN is partitioned across a set of simulators. The resulting global model spanning all partitions is a Distributed Colored Petri Net (Dcpn). Therefore, a synchronization protocol is required that enforces a formally specified consistency criterion between the participating simulator instances.

As a consistency requirement, we demand correctness of distributed execution with respect to partial order semantics [22, 23] (PO semantics) as the reference semantics of non-distributed simulation: every distributed execution generated by the protocol must correspond to an event order permitted by PO semantics. Equivalently, every observed sequence of transition firings in the distributed simulation must be a linearization of a permissible partial order in the sense of the semantics defined in Section 2.3.

In addition to correctness, throughput and scalability are key requirements. Throughput denotes the rate at which synchronization processes or relevant events are processed in distributed execution (e.g., completed synchronization operations per unit of time). Scalability denotes the ability to increase this throughput as model size grows efficiently or the number of simulator instances increases (or to reduce resources as demand decreases) without coordination and communication efforts dominating. The central research question of this work is the following:

> **Central Research Question.** How could a synchronization mechanism for distributed simulation of coloured Petri nets be designed (i) so that correctness concerning PO reference semantics applies, and (ii) so that the synchronization mechanism can be implemented in a practically efficient, high-throughput, and scalable manner?

To answer this question, syntactic and semantic elements at the model level are required to express rendezvous synchronization. These elements should make synchronization easy to use for the modeler while keeping the additional modeling effort as low as possible. This leads to the first research question:

> **RQ1.** What syntactic and semantic elements are required to express synchronization without significantly increasing the modeling effort?

In addition to model-level concepts, distributed simulation requires a system architecture that serves as a proof of concept for the proposed synchronization mechanism. Such an architecture must provide distributed processes capable of performing rendezvous synchronization while preserving the required consistency properties. Therefore, the second research question is:

> **RQ2.** What distributed system architecture is needed to realize the consistency requirements for this distributed simulation?

Finally, the distributed processes require concrete protocols to coordinate rendezvous synchronization. These protocols must ensure consistency and correctness while remaining suitable for efficient and scalable execution. This leads to the third research question:

> **RQ3.** Which rendezvous synchronization protocols are suitable for our concept to ensure consistency and correctness?

4. Conceptual Solution

This contribution addresses the synchronization mechanism between two simulators for distributed simulation of coloured Petri nets. The goal is to bridge simulator boundaries such that distributed execution reproduces a well-defined reference semantics of non-distributed simulation, without implementation-related artifacts—specifically, without additional non-determinism due to scheduling or communication latencies. For this purpose, three realization principles can be distinguished: memory, message, and rendezvous synchronization (Section 2.1).

The central limitation of message and memory synchronization is that pure asynchrony provides no intrinsic mechanism to enforce a common, atomic synchronization point—i.e., an indivisible logical event/step across simulator boundaries. [51, 52] Ensuring such progression therefore requires additional protocol steps (e.g., handshakes, barriers, acknowledgments). Without these mechanisms, delays in observing and processing information may occur, so strictly coupled execution of individual actions across simulator boundaries cannot be guaranteed immediately.

A synchronous model with global time barriers or rounds/handshakes subsumes the asynchronous model as a special case (it can always ignore the clock), while providing a coordination resource unavailable in pure asynchrony. This resource increases expressiveness and problem-solving capabilities: deterministic consensus is impossible in completely asynchronous systems in the presence of crashes [53], whereas it can be constructed under full or partial synchronization. Synchronization thus builds on asynchronism by incorporating atomic, blocking, and time-bound coupling, reinforcing the underlying assumption. Synchronous communication permits coupling and consistency conditions to be expressed as joint, blocking interactions rather than being emulated via subsequent protocol constructions. In this paper, rendezvous synchronization (Section 2.1) is assumed to have a semantics in terms of partially ordered event spaces with explicitly defined synchronization events: communication partners may proceed independently, but are brought together in a joint step at explicitly defined synchronization events.

For Distributed Colored Petri Nets, rendezvous synchronization can be modeled using synchronous channels. We distinguish (i) local synchronous channels, whose communication partners are located within the same simulator, and (ii) distributed synchronous channels, whose communication partners exist in a different simulator. The main conceptual contribution of this work is the introduction of distributed synchronous channels as a new language and modeling element for coloured Petri nets, enabling bidirectional synchronous communication across simulator boundaries.

For representing control flow, Agent Unified Modeling Language (AUML) Sequence Diagrams, a graphical notation for modeling agent interactions, have been proposed by FIPA and others [54, 55]. AUML Sequence Diagrams are also referred to as Agent interaction protocol diagrams [56] (Aips). To realize these distributed synchronous channels, Aips are employed for the distributed processes. This foundation led to the introduction of Petri net semantics for these Aips in [57]. As a result, it is possible to generate a Petri net from the Aips for the distributed processes. The correctness of this Petri net can be verified using workflow properties [58, 59]. This approach ensures the control flows of the distributed processes are correct.

The Globally Asynchronous Locally Synchronous [60, 61, 62] (GALS) paradigm describes systems without a global clock that are coupled asynchronously, yielding global asynchrony with simultaneous local synchrony within each domain. Globally Synchronous Locally Synchronous [63, 52] (GSLS) extends this to global synchrony by providing a standard time reference via a system-wide clock or equivalent mechanism. However, for the distributed synchronous channel structure considered here, the Partially Globally Synchronous Locally Synchronous (PGSLS) paradigm applies: synchronous channels introduce global synchrony only partially for the considered synchronized transitions, namely within the transitions involved in the net modules. This synchrony is not total system-wide because overall execution follows a partial-order semantics. Outside the synchronization locality, causally independent transitions can continue to switch concurrently as long as there are no dependencies via synchronous channels or other causal relations. PGSLS thus models localized synchronization islands with strong coupling where semantically required, while maintaining maximum concurrency in the rest of the system.

5. Distributed Colored Petri Nets

5.1. Distributed Synchronous Channels

Distributed synchronous channels extend local synchronous channels 2.2 to enable distributed synchronization, i.e., synchronization of channels across distributed simulators. As with non-distributed

channels, a transition inscribed with a distributed synchronous channel must find another such transition with a matching signature.

Syntax The syntax mirrors local synchronous channels: downlinks and uplinks. Distributed channels are denoted *DD* and *DU* (i.e., *Distributed Downlink* and *Distributed Uplink*); a downlink must match an uplink. The remaining inscription is unchanged: channel name as identifier, and a tuple of parameters for information exchange. Transitions with a distributed synchronous channel are restricted to have no other synchronous channels, local or distributed; multiple channels are left for future work.

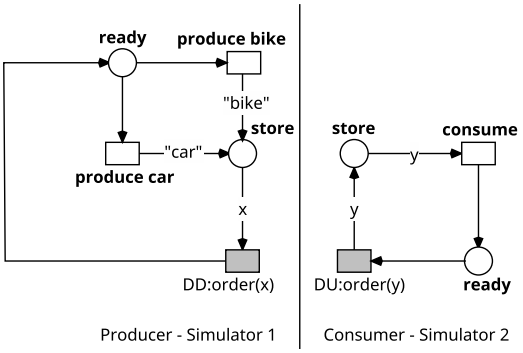

Figure 1: Example: Producer and Consumer

Figure 1 shows two nets (in distributed simulators), each with a transition (gray) inscribed with a distributed synchronous channel, in a Producer-Consumer setting: the Producer produces `"car"` or `"bike"` and synchronizes with the Consumer because one inscription is a downlink and the other an uplink with the same channel name and a matching parameter tuple; during synchronization, `x` (downlink) binds to `y` (uplink), and during firing the value taken by `x` moves from the Producer's `store` place to the Consumer's `store` place. For simplicity, we omit explicit color mappings.

Semantics The semantics can be demonstrated using workflow transitions [64], which transfer resettability [65] from classical workflow nets [66] to transitions and are used here to address distributed synchronization. Because communication can render preset markings obsolete, workflow transitions allow token reservation: synchronization concludes successfully only once each transition can assure it can fire (i.e., has reserved the necessary tokens), otherwise reserved tokens are reset and returned. Hence, either all synchronizing transitions fire or none do, so firing a distributed synchronous channel remains one atomic action.

Figure 2 shows the workflow transitions for Figure 1: a `request`-transition reserves tokens, a `cancel`-transition frees reserved tokens, and a `confirm`-transition completes a successful firing. State consistency is essential: the entire DCPN must remain consistent throughout every simulator, with the required step sequence defined by a protocol (Section 5.3). After synchronization is initiated, `request`-transitions attempt to fire and communicate their firing status; if any request fails, it notifies the others to trigger `cancel`-transition and stop synchronization. A `confirm`-transition must only fire if every participating transition reserved its required tokens via `request`-transition; in that case each `confirm`-transition must fire to keep a consistent state over the entire DCPN and conclude the distributed firing.

Firing any transition with a distributed synchronous channel is atomic and can occur concurrently with any other transition, including itself; thus distributed synchronous channels fit the established partial order semantics of coloured Petri nets 2.3. Moreover, they do not increase the computational power of coloured Petri nets, since unfolding the workflow transitions yields behavioral equivalence to regular coloured Petri nets without synchronous channels.

In the unfolding, all `confirm` transitions are merged into a single transition to represent atomicity, while the remaining elements stay separate. A `sync` transition initiates synchronization by checking,

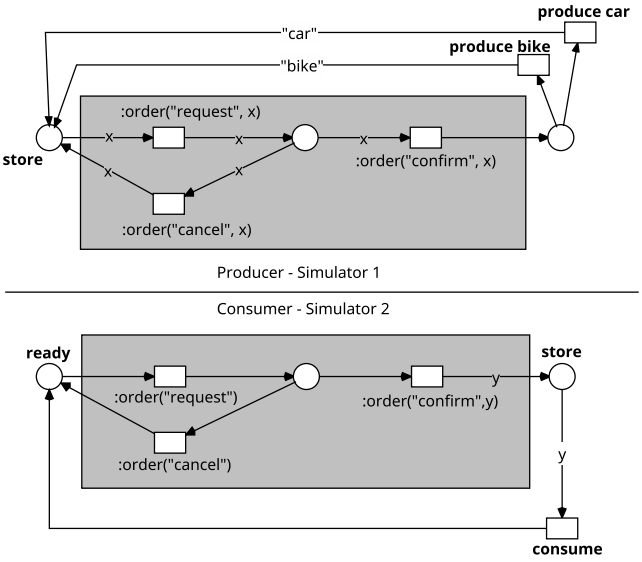

Figure 2: Workflow transitions for the Producer-Consumer example

for each preset place of either workflow transition, that the tokens required for synchronization are available. The *request*-transitions are enabled by adding two new preset places per transition that receive a token only when `sync`-transition fires; these tokens can be removed by `request`- or `cancel`-transitions, ensuring that after each successful or unsuccessful synchronization the unfolded workflow transition returns to its initial state (Figure 3).

Fairness This unfolding represents the behavior of Figure 2 but not the external influence required to facilitate synchronization: even if `request`-transitions are continuously (or repeatedly) activated via `sync`-transition, they may never fire; the same holds for `confirm`-transition. Thus, initiating synchronization via `sync`-transition does not ensure successful conclusion, which is a fairness problem. [67, 1] While one notion of fairness requires that a transition enabled infinitely often fires infinitely often, here we require a `Fair Selection Rule` [68] ensuring that any transition continuously enabled will fire. Then `request`-transitions and `confirm`-transition eventually fire when enabled, ensuring that a started synchronization concludes as long as markings in the preset places remain available. Therefore, we assume the unfolded net is fair w.r.t. the Fair Selection Rule for these transitions; under this assumption we accurately capture the behavior of the workflow transition.

5.2. Distributed System Architecture

The target architecture (Figure 4) distributes a coloured Petri net (Section 2.3) by partitioning it across simulators; the net spanning all simulators is a Distributed Colored Petri Net (Dcpn). It aims to (i) accelerate simulation for fixed model size and (ii) enable larger/more detailed models, scaling to many, potentially millions, of simulators.

Quality Attributes Latency is the interval between marking a distributed synchronous channel and system-wide completion of its firing; throughput is the number of successful distributed firings per second.

The reference scenario comprises three computer nodes: a Kafka broker (communication medium), one Renew as synchronization service, and two additional Renew's as simulators on different nodes. Each simulator hosts one net module (downlink vs. uplink); the producer-consumer example 1 is used.

Evaluation must cover strong and weak scaling. Strong scaling accelerates a fixed-size Dcpn by adding simulators, with speedup limited by serial components and protocol/synchronization coordination

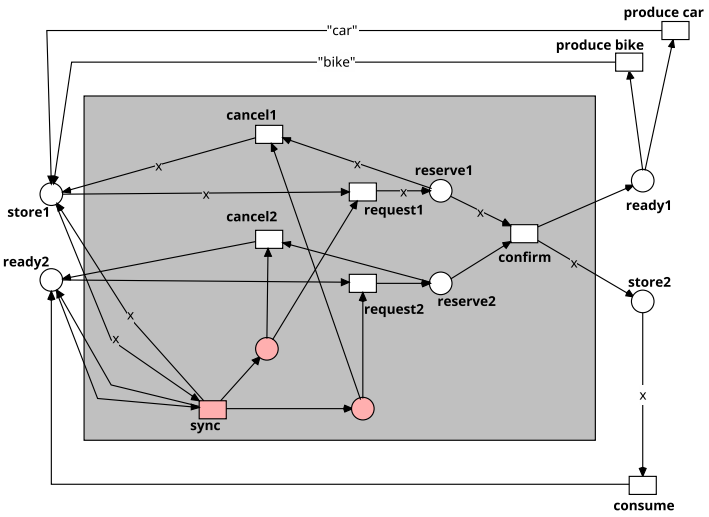

Figure 3: Unfolded Producer-Consumer net

costs (asymptotic marginal utility is usually discussed via Amdahl's Law). Weak scaling targets similar runtime for larger DCPNs by proportionally increasing simulators, keeping local work per simulator approximately constant.

For reproducibility, distributed execution must be semantically equivalent to the non-distributed reference under partial order semantics (Section 5.1); any additional non-determinism from scheduling, communication, or partitioning must be excluded, leaving only options explicitly permitted by the model semantics. Throughput is prioritized over latency provided latency SLOs are met, to maximize simultaneously executable firings and thus effective simulation rate.

Components communicate orderly and reliably via TCP. The communication medium connects simulators via synchronous channels: per channel, messages are totally ordered by a unique linear sequence relation; no global order across channels is assumed. Durability is commit-based: a message is committed only after permanent securing per replication and quorum policy; with replication factor N, tolerable failures depend on the quorum policy. The error model is non-Byzantine; completely unpredictable node behavior (including sending correct, incorrect, or contradictory information to different parts of the system) is out of scope. Delivery is at-least-once: each committed message is delivered at least once and may be delivered multiple times, so processing must tolerate duplicates via idempotency, deduplication, or state-based consistency mechanisms. No global clock is provided; progress and order derive exclusively from causal dependencies. Millisecond metrics denote execution time and must be separated from the simulation's model time.

Components The communication medium must be scalable and robust; event streaming is suitable, with Apache Kafka [26] as a concrete implementation. Robustness derives from decoupling of partners, replication, and recoverability (retention, replay, log compaction); scalability from partitioning, consumer groups, and horizontal broker scaling. To ensure order across distributed synchronous channels, each channel is mapped to a Kafka topic.

Each simulator runs one or more DCPN modules with external interfaces given by distributed synchronous channels; each holds part of the system state, and all simulators jointly constitute the global simulation state and thus the DCPN. Simulators are implemented using RENEW (Section 2.5). For distributed firings, simulators stream static and dynamic parts of distributed synchronous channels to the synchronization service via the communication medium: static parts comprise net structure (edges, pre- and post-area places, guards) of the affected channel; dynamic parts comprise markings of places in the pre-area.

The synchronization service constructs an Artificial Transition from this static/dynamic information

and checks unifiability; if successful, it informs the involved simulators of the required binding so the distributed circuit fires consistently. It scales via sharding, with each instance responsible for a specific set of distributed synchronous channels, eliminating the synchronization service as a bottleneck; it is implemented with Renew (Section 2.5).

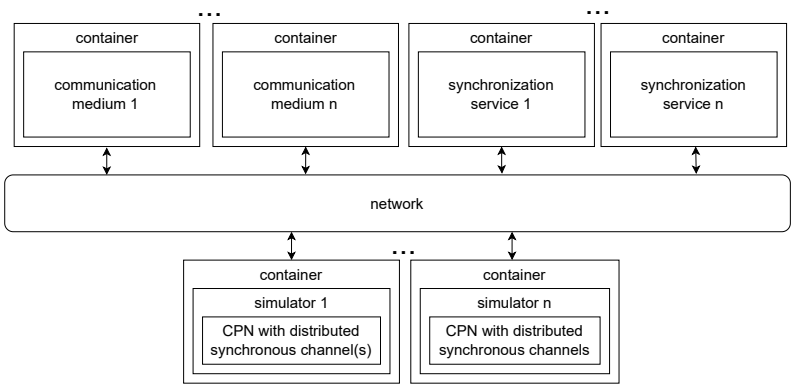

Figure 4: Distributed System Architecture

5.3. Protocols

This section introduces the event language and the protocols for registering transitions, updating a preset, and firing a distributed synchronous channel. The protocols are modelled using Agent interaction protocol diagrams (Aips) [56], based on the AUML Sequence Diagrams proposed by FIPA et al. [54, 55]. Aips extend UML sequence diagrams with case distinctions and concurrency. A Petri net–based formal semantics for Aips (reference nets) was proposed in [57], providing an operational semantics based on partial order semantics that supports the design of complex distributed systems.

Event Language Event streaming provides communication between simulators and Synchronization Service. The event language defines protocol messages and categorizes actions that require communication. All events share: the Universal Unique Identifier (UUID) [69] of the event; the UUID of the simulator; the UUID of the net instance; and the UUID of the synchronous channel. These UUIDs identify the relevant content, enabling administration and processing of events and reproduction of the firing sequence of a Dcpn. Use-case-specific content depends on the protocol phase. The event language comprises eleven events, grouped into registration, updating, and firing (Table 1).

Category	Event name	Description
Registration events	RegisterUplink/ RegisterDownlink	A downlink or uplink registers with the Synchronization Service.
	DeregisterUplink/ DeregisterDownlink	A downlink or uplink deregisters with the Synchronization Service.
Updating events	UpdateUplink/ UpdateDownlink	An uplink or downlink informs the Synchronization Service that its preset has changed and how it has changed.
Firing events	RequestFiring	Synchronization Service instructs the simulators to fire.
	ConfirmUplink/ ConfirmDownlink	The uplink or downlink confirm that they are ready to fire.
	CancelUplink/ CancelDownlink	The uplink or downlink report that they cannot fire.

Table 1
Protocol event categories, event names, and their functional roles

Registration events additionally include synchronous-channel and transition information, including guards and arc weights. Updating events provide supplementary information about the new preset of the transition. Firing events further specialize content for transitions with synchronous channels; the corresponding protocols are detailed below. Event contents follow the JSON specification [70], primarily to facilitate reading and writing by humans and machines.

Registration On startup, the registration protocol establishes communication between the simulators, Kafka, and the Synchronization Service (Figure 5). Each simulator evaluates whether it hosts transitions belonging to a distributed synchronous channel; if not, the protocol terminates immediately (*IStart*, false branch). Otherwise, for each such transition, the simulator sends a RegisterUplink or RegisterDownlink event to Kafka, carrying the transition UUID, channel name, parameter tuple, and preset structure. The Synchronization Service polls Kafka for new events; if none are available yet (false branch), it waits until events arrive (true branch), then stores the received structural information. A transition is considered for unification only after its information has been stored. At simulation end, DeregisterUplink and DeregisterDownlink events remove transitions from the Synchronization Service's registered set.

This design deliberately avoids a centralized bottleneck: simulators register independently and concurrently, while the Synchronization Service defers unification checks until all partners are known. Consequently, neither Kafka nor the Synchronization Service becomes a synchronization barrier during startup. [71, 72, 73]

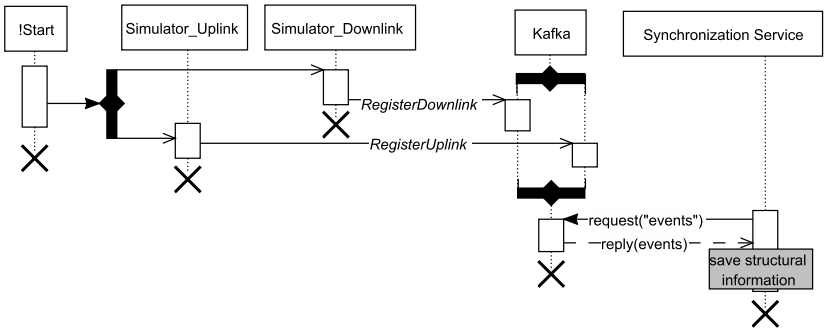

Figure 5: Registration protocol: simulators publish structural information to Kafka; the Synchronization Service polls and stores it before admitting transitions to unification

Updating the Preset Whenever the preset of an uplink or downlink changes, the owning simulator triggers the update protocol (Figure 6). It sends an UpdateUplink or UpdateDownlink event to Kafka, containing the transition UUID and the updated preset marking. The Synchronization Service polls Kafka; upon receiving the event, it stores the new marking and immediately re-evaluates whether firing on the affected channel is now possible. This continuous synchronization ensures that the Synchronization Service always operates on current preset information without requiring simulator-initiated firing requests. [71, 72, 73]

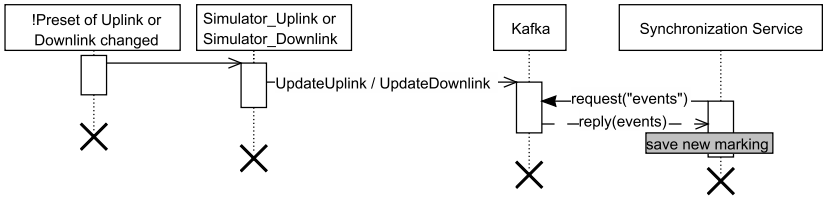

Figure 6: Update protocol: a simulator publishes a changed preset marking to Kafka; the Synchronization Service polls, stores, and re-evaluates firability

Firing The firing protocol coordinates distributed channel synchronization via a two-phase, peer-confirmed mechanism (Figure 7), triggered whenever the Synchronization Service stores new structural information. Both phases are designed to minimize Synchronization Service involvement and shift coordination load to the participating simulators as early as possible.

In the first phase, the Synchronization Service constructs an artificial transition (*CreateArtificialTransition*) from the stored presets of a candidate uplink–downlink pair and attempts unification (*Unification*). If unification fails, the protocol terminates silently (false branch). If it succeeds, the Synchronization Service publishes a ConfirmFireable event to Kafka, containing the firing UUID, the computed parameter allocation, and the identifiers of both participating transitions.

In the second phase, the uplink and downlink simulators act concurrently and symmetrically. Each polls Kafka, receives ConfirmFireable, and executes *EvaluateChannel* locally. A simulator that cannot fire rolls back any tentative changes and publishes CancelUplink or CancelDownlink (false branch); one that can fire reserves its marking and publishes ConfirmUplink or ConfirmDownlink (true branch). Each simulator then polls Kafka a second time to receive its *partner's* reply and evaluates it (*EvaluateLocalDownlinkAnswer* or *EvaluateLocalUplinkAnswer*): a positive partner reply triggers *FireTransition*; a negative reply triggers a local rollback. This symmetric, peer-to-peer confirmation guarantees that a transition fires on both sides if and only if both partners independently confirm, without any further involvement of the Synchronization Service. [71, 72, 73]

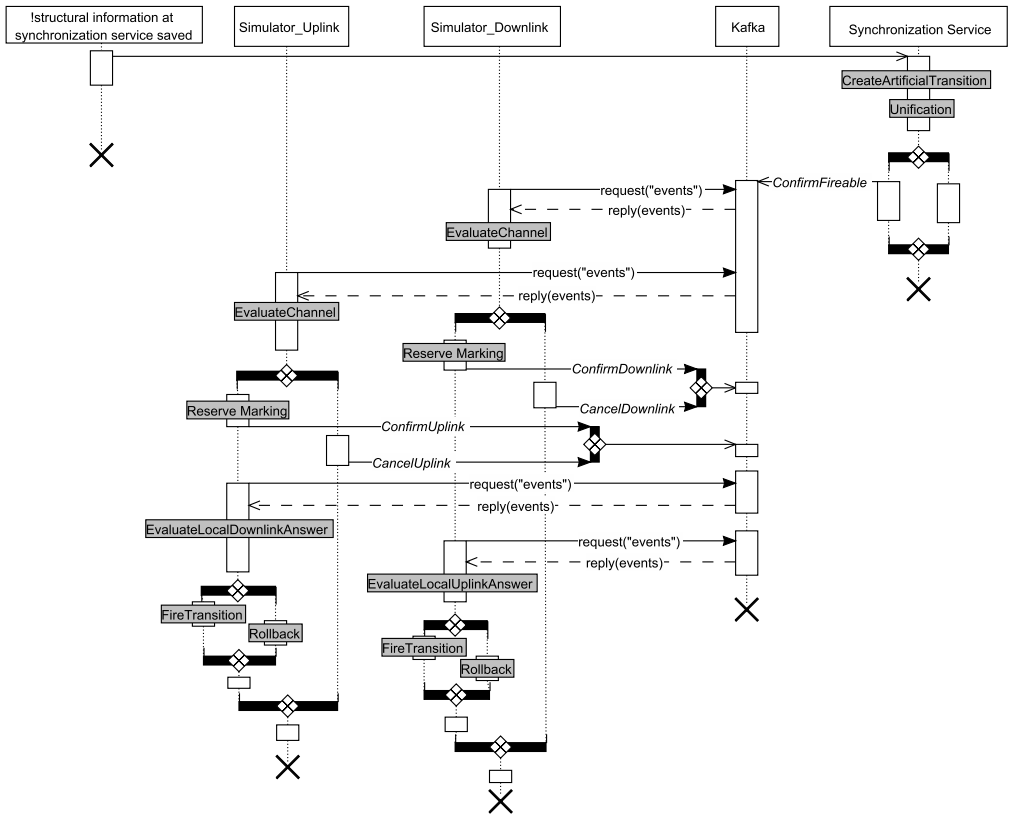

Figure 7: Firing protocol: two-phase peer-confirmed synchronization — the Synchronization Service verifies unification (Phase 1), then uplink and downlink simulators concurrently evaluate and cross-confirm via Kafka before firing (Phase 2)

6. Results

This section assesses the proposed bidirectional rendezvous synchronization concept for the distributed simulation of coloured Petri nets against the research questions in the problem description (Section 3).

RQ1: What syntactic and semantic elements are required to express bidirectional rendezvous synchronization without significantly increasing the modeling effort? This contribution introduces bidirectional rendezvous synchronization of transitions in Distributed Colored Petri Nets, using distributed synchronous channels (Section 5.1). Syntactically, an explicit notation (DD/DU) distinguishes local from distributed synchronous channels by uniquely identifying distributed uplinks and downlinks, making channel affiliation explicit without auxiliary nets. Semantically, each distributed synchronous transition is realized by a simulator-internal rendezvous workflow transition that checks local fireability, coordinates via the channel, and executes the global firing action only after the synchronized partner transitions have been consistently determined.

RQ2: What distributed system architecture is needed for this distributed simulation to implement the consistency requirements? The architecture comprises three central components—communication medium, synchronization service, and simulators (Section 5.2). The communication medium uses event streaming: components interact via event-based interfaces rather than point-to-point connections, yielding loose coupling between simulators so that errors in individual simulators cannot propagate to the overall simulation. Kafka serves as the communication medium, enabling high-availability provisioning and scalability under high communication loads; since Kafka is optimized for throughput rather than latency, it optimizes the number of concurrent distributed firings. The synchronization service centrally unifies distributed synchronous channels and is, in principle, scalable, further optimizing the number of concurrent distributed firings. The simulators simulate Dcpns.

RQ3: Which rendezvous synchronization protocols are suitable for our concept to ensure consistency and correctness? We use three logically separate protocols (Section 5.3). The registration protocol transmits the static structures of a distributed synchronous channel to the synchronization service—slots in the pre- and post-area and the incoming and outgoing edges of the uplink or downlink—establishing the basis for matching/unification decisions. The update protocol transmits dynamic state information, i.e., changes to the relevant markers in the pre-area. The firing protocol implements the distributed firing action as a coordinated commit between participating simulators: the synchronization service performs channel-related unification, and the simulators execute the rendezvous workflow transition, finalizing local token removal/production only after a successful global decision. Importantly, all of these protocols were modeled using so-called Agent Interaction Protocol Diagrams [56], which, according to [57], have Petri net semantics. With these Petri net semantics, the workflow properties [74] can be verified for the modeled protocols. Consequently, the control structures of the protocols introduced here are correct with respect to the workflow properties.

Central RQ: How could a synchronization mechanism for distributed simulation of coloured Petri nets be designed (i) so that correctness concerning PO reference semantics applies, and (ii) so that the synchronization mechanism can be implemented in a practically efficient, high-throughput, and scalable manner? This article introduces a bidirectional rendezvous synchronization mechanism for distributed simulation of coloured Petri nets. Relative to local simulation, the protocols allow all previously possible firing sequences and introduce no additional sequences, thereby preserving the PO reference semantics of the simulators. The distributed system architecture enables scaling of individual components in principle; event streaming via Kafka optimizes throughput and thereby optimizes the number of concurrent distributed firings. Moreover, partial synchrony in the locality of the distributed transitions of a distributed synchronous channel introduces the Partially Globally Synchronous Locally Synchronous paradigm by applying bidirectional rendezvous synchronization.

Following the constructivist prototyping methodology, we implemented a proof of concept to demonstrate basic feasibility: the simulators and synchronization service were implemented with Renew, and the communication medium with Kafka. The proof-of-concept currently supports the basic data types, integers, and strings, for distributed synchronous channels, thus demonstrating basic feasibility. It does not claim product quality, which has limited its use as a tool so far; this is part of future work.

The proposed concept could be suitable to teach students about distributed systems and software engineering, allowing them to work with concrete models. Furthermore it could be beneficial for formal analysis tasks where the decidability of certain properties is unknown, making simulation a practical method for assessing correctness. It could also support inherently simulation-heavy tasks such as

what-if analysis. Lastly, extending this idea further, the simulator could be used to generate logs for object-centric processes or multi-agent systems more efficiently.

7. Discussion

The concept enables simulation of a DCPN model across multiple simulator instances, assuming the net is pre-partitioned into modules and statically assigned. Under this assumption, compute and memory load can be distributed across nodes. Speedup depends strongly on the fraction of locally executable firings versus distributed, synchronously coupled firings and on coordination latency. Aggregating CPU and memory across simulators enables larger, more detailed models, but bottlenecks may shift to synchronous-channel coordination and to network bandwidth and latency, limiting scalability.

Under the specified coordination and communication assumptions, distributed simulation retains the intended partial-order semantics of the underlying DCPN, preserving the order of firing operations induced by causality and competition across partition boundaries. The protocol ensures semantic fidelity by permitting no firing combinations beyond the reference model and by neither creating nor destroying tokens, but only operationalizing specified token flows and marking changes.

An event-streaming approach implemented with Apache Kafka provides persistence, horizontal scalability, and high throughput. With appropriate persistence and replication, it can reduce event-loss probability under failures, although end-to-end guarantees depend on the error model and chosen delivery semantics. Scalability targets high communication volumes via partitioning and parallel consumer processing; because Kafka is throughput-optimized, the latency–throughput trade-off must be explicitly accounted for in synchronization-sensitive implementations. Kafka-based event streaming also decouples simulators at the communication layer, since producers and consumers do not communicate directly, reducing error propagation and increasing robustness.

The concept supports distributed modeling and simulation by enabling separate module development. This requires precise coordination of interfaces so the benefit is primarily organizational and depends on clear module and interface specifications.

Compared to a monolithic simulator, the architecture increases component count and heterogeneity (multiple simulator instances, a messaging infrastructure, and the synchronization service), increasing operating costs and generally requiring distributed runtime environments such as Kubernetes orchestration. Rendezvous firings across partition boundaries incur additional protocol rounds, serialization, and transport overhead, and thus typically have higher latency than purely local firings; overall simulation can be slower, especially with a high proportion of synchronously coupled transitions or hotspot coordination points. Predetermined simulator count and fixed module assignment can lead to poor partitioning, resulting in uneven performance across simulators and affecting performance. Without dynamic repartitioning or adaptive load balancing, this limitation persists, altering overall efficiency.

The proof of concept currently supports only strings and integers, while the concept includes an extension to other elementary or complex data types. The concept currently requires a homogeneous simulator environment, i.e., identical definition and evaluation of data types, guards, and inscriptions; removing this assumption would require additional mechanisms. Even with highly available messaging, simulators lack well-defined recovery mechanisms (e.g., checkpointing, deterministic replay, rejoin protocols), so simulator failures can still interrupt the simulation or lead to inconsistent states. While the focus is on distributed simulation, basic mechanisms—such as partitioning and rendezvous-based synchronization—could plausibly serve as a basis for verification tasks, linking simulation concepts to wider applications.

This paper emphasizes concepts and architecture. Consequently, a performance analysis is not included in this paper. This omission sets the stage for upcoming research. Likewise, this paper does not include a performance comparison with other distributed simulators, further illustrating the focus on foundational concepts. Building on these omissions, these aspects—performance analysis of DCPNs and comparison with other distributed simulators—will be addressed in further research.

8. Related Work

In this section, we discuss related work on distributed net simulation and synchronisation mechanisms for Petri nets. Christensen et al. [20] introduce synchronous channels to coloured Petri nets; we extend this concept with distributed synchronous channels for distributed modelling and simulation. Other synchronization concepts for coloured Petri nets—place- and transition-fusion [75] and hierarchical coloured Petri nets [76, 77]—differ in execution and function from synchronous channels and do not consider distribution across multiple simulators.

Concurrent Object Oriented Petri Nets (CO-OPN) by Buchs [78] combine Petri nets with object-oriented concepts based on algebraic Petri nets [79]. CO-OPN and CO-OPN\2 [80] provide synchronous communication mechanisms supporting concurrent actions between distributed synchronizing components [81, 82]. However, components are modelled as objects; while each object can be seen as a Petri net, communication is not modelled through Petri nets, and distribution follows the object-oriented paradigm rather than distributed simulators.

Voss et al. [83] introduce synchronous channels for place/transition nets (P/T Nets). Clasen et al. [71], Bartelt [73], and Stahl [72] extend these to distributed P/T nets, accounting for simulator distribution and using rendevouz synchronization, but without colors and thus only black tokens. For Reference Nets, which allow net references using java as an inscription language [21], Simon et al. [84] introduce a distributed algorithm using remote procedure calls between distributed simulators, trading scalability and robustness for latency, distinctly different from our event based approach. Clasen et al. [85] build upon Simon and simulate Reference Nets using event streaming; beyond formalism, execution differs as we coordinate distributed simulators via the centralized Synchronization Service, whereas they use a decentralized approach where each simulator evaluates and processes events from each other simulator.

9. Future Work

The proof of concept provides a robust foundation for Dcpns equipped with strings and integers and motivates further research. Since this paper emphasizes concepts and architecture, it deliberately does not include a performance analysis or a performance comparison with other distributed simulators. These omissions further underline the basic focus of the present work and set the stage for future research on the runtime behavior, throughput, latency, scalability, and comparative performance of Dcpns.

Further research directions comprise: (i) expanding data type support beyond strings and integers (characters, floating-point numbers, particularly user-defined data types) to align Dcpns with classical CPNs; (ii) distributed synchronous channel architectures beyond one-to-one uplink–downlink correspondence (one uplink with multiple downlinks; multiple synchronous channels per transition; chained synchronizations forming multi-party rendezvous) supporting broadcast-style coordination, selective communication patterns, and hierarchical synchronization structures; and (iii) systematic performance analysis of Dcpns, including Kafka-level throughput and latency optimization as well as comparison with other distributed simulators.

Longer-term directions include generalizing the Synchronization Service-based architecture to other formalisms, such as reference nets [21], for multi-formalism simulations and for synchronized communication between reference nets, Distributed P/T Nets, and Dcpns. As demonstrated in [86] for locally synchronized multi-formalisms in Renew, this would foster integrated simulation environments under appropriate semantic constraints. Further directions are dynamic Synchronization Service scaling (not yet implemented), inspired by Distributed P/T Nets [87], and resilience mechanisms for state preservation and recovery as in Distributed P/T Nets [88], which are challenging for Dcpns but would improve fault tolerance for long-running simulations in unstable environments.

Declaration on Generative AI

During the preparation of this work, the authors used …

- …DeepL in order to: Translate Text.
- …ChatGPT in order to: Rephrasing.
- …Grammarly in order to: Grammar and spelling check, Rephrasing.
- …Claude in order to: Rephrasing.
- …Perplexity in order to: Rephrasing.

After using these tool(s)/service(s), the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] W. Reisig, Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies, Springer, 2013. URL: https://doi.org/10.1007/978-3-642-33278-4. doi:10.1007/978-3-642-33278-4.

[2] K. Jensen, Coloured Petri Nets: Volume 1; Basic Concepts, Analysis Methods and Practical Use, EATCS Monographs on Theoretical Computer Science, Berlin Heidelberg New York, 1992.

[3] K. Jensen, L. M. Kristensen, Coloured Petri Nets - Modelling and Validation of Concurrent Systems, Springer, 2009. URL: https://doi.org/10.1007/b95112. doi:10.1007/B95112.

[4] K. M. van Hee, N. Sidorova, J. M. van der Werf, Business Process Modeling Using Petri Nets, in: Transactions on Petri Nets and Other Models of Concurrency VII, Springer, 2013, pp. 116–161.

[5] S. Ebert, J. Mey, R. Schöne, S. Götz, U. Aßmann, Distributed Petri nets for model-driven verifiable robotic applications in ROS, Innovations in Systems and Software Engineering (2024) 1–27.

[6] R. Fujimoto, Parallel and distributed simulation, in: Proceedings of the 2015 Winter Simulation Conference, Huntington Beach, CA, USA, December 6-9, 2015, IEEE/ACM, 2015, pp. 45–59. URL: https://doi.org/10.1109/WSC.2015.7408152. doi:10.1109/WSC.2015.7408152.

[7] R. M. Fujimoto, Research challenges in parallel and distributed simulation, ACM Transactions on Modeling and Computer Simulation (TOMACS) 26 (2016) 1–29.

[8] R. M. Fujimoto, Development of the parallel and distributed simulation field, Simul. 100 (2024) 1197–1223. URL: https://doi.org/10.1177/00375497241261407. doi:10.1177/00375497241261407.

[9] R. Budde, K. Kautz, K. Kuhlenkamp, H. Züllighoven, What is prototyping?, Information Technology & People 6 (1990) 89–95.

[10] G. Pomberger, W. Pree, A. Stritzinger, Methoden und Werkzeuge für das Prototyping und ihre Integration, Inform., Forsch. Entwickl. 7 (1992) 49–61.

[11] T. Wilde, T. Hess, Forschungsmethoden der Wirtschaftsinformatik, Wirtschaftsinformatik 4 (2007) 280–287.

[12] P. Ralph, N. bin Ali, S. Baltes, D. Bianculli, J. Díaz, Y. Dittrich, N. Ernst, M. Felderer, R. Feldt, A. Filieri, B. B. N. de França, C. A. Furia, G. Gay, N. Gold, D. Graziotin, P. He, R. Hoda, N. Juristo, B. Kitchenham, V. Lenarduzzi, J. Martínez, J. Melegati, D. Mendez, T. Menzies, J. Molleri, D. Pfahl, R. Robbes, D. Russo, N. Saarimäki, F. Sarro, D. Taibi, J. Siegmund, D. Spinellis, M. Staron, K. Stol, M.-A. Storey, D. Tamburri, M. Torchiano, C. Treude, B. Turhan, X. Wang, S. Vegas, Empirical Standards for Software Engineering Research, Technical Report arXiv:2010.03525, arXiv, 2020. URL: https://arxiv.org/abs/2010.03525.

[13] Stankovic, A perspective on distributed computer systems, IEEE Transactions on Computers C-33 (1984) 1102–1115. doi:10.1109/TC.1984.1676389.

[14] A. Hamdani, A. Abdelli, Time petri net with rendezvous, in: 2017 4th International Conference on Control, Decision and Information Technologies (CoDIT), 2017, pp. 0126–0131. doi:10.1109/CoDIT.2017.8102578.

[15] R. Mohanty, D. P. Behera, A. Routray, Analysis and realization of relaxed consistency memory model for multi-core cpu or gpu, in: 2014 5th International Conference - Confluence The Next Generation Information Technology Summit (Confluence), 2014, pp. 866–870. doi:10.1109/CONFLUENCE.2014.6949247.

[16] Y. Timura, K. Tanaka, M. Takizawa, Object-based precedency of messages in object-based systems, in: Proceedings. Fifth International Workshop on Object-Oriented Real-Time Dependable Systems, 1999, pp. 141–147. doi:10.1109/WORDSF.1999.842345.

[17] C. A. R. Hoare, Communicating sequential processes, Communications of the ACM 21 (1978) 666–677. URL: https://doi.org/10.1145/359576.359585. doi:10.1145/359576.359585.

[18] S. D. Brookes, C. A. R. Hoare, A. W. Roscoe, A theory of communicating sequential processes, Journal of the ACM 31 (1984) 560–599. URL: https://doi.org/10.1145/828.833. doi:10.1145/828.833.

[19] A. W. Roscoe, The Theory and Practice of Concurrency, Prentice Hall International Series in Computer Science, Prentice Hall, 1997.

[20] S. Christensen, N. D. Hansen, Coloured Petri Nets Extended with Channels for Synchronous Communication, Technical Report DAIMI PB-390, Aarhus University, 1992. URL: https://tidsskrift.dk/daimipb/article/view/6625.

[21] O. Kummer, Referenznetze, Logos Verlag, Berlin, 2002. URL: http://www.logos-verlag.de/cgi-bin/engbuchmid?isbn=0035&lng=eng&id=.

[22] W. Brauer, W. Reisig, G. Rozenberg (Eds.), Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986, volume 254 of *Lecture Notes in Computer Science*, Springer, 1987. URL: https://doi.org/10.1007/978-3-540-47919-2. doi:10.1007/978-3-540-47919-2.

[23] W. Brauer, W. Reisig, G. Rozenberg (Eds.), Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986, volume 255 of *Lecture Notes in Computer Science*, Springer, 1987. URL: https://doi.org/10.1007/3-540-17906-2. doi:10.1007/3-540-17906-2.

[24] J. Kreps, I heart logs: Event data, stream processing, and data integration, " O'Reilly Media, Inc.", 2014.

[25] B. M. Michelson, Event-driven architecture overview, 2006.

[26] Apache Software Foundation, Introduction, 2026. URL: https://kafka.apache.org/41/getting-started/introduction/.

[27] J. Kreps, N. Narkhede, J. Rao, Kafka: A distributed messaging system for log processing, in: Proceedings of the 6th International Workshop on Networking Meets Databases, Athens, Greece, 2011, pp. 1–7.

[28] O. Kummer, F. Wienberg, M. Duvigneau, L. Cabac, M. Haustermann, D. Mosteller, M. Hansson, Renew – the Reference Net Workshop, 2026. URL: http://www.renew.de/, release 5.0.

[29] L. Clasen, D. Moldt, M. Hansson, S. Willrodt, L. Voß, Enhancement of Renew to version 4.0 using JPMS, in: M. Köhler-Bußmeier, D. Moldt, H. Rölke (Eds.), Proceedings of the International Workshop on Petri Nets and Software Engineering 2022 co-located with the 43rd International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2022), Bergen, Norway, June 20th, 2022, volume 3170 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2022, pp. 165–176. URL: https://ceur-ws.org/Vol-3170.

[30] D. Moldt, J. Johnsen, R. Streckenbach, L. Clasen, M. Haustermann, A. Heinze, M. Hansson, M. Feldmann, K. Ihlenfeldt, RENEW: modularized architecture and new features, in: L. Gomes, R. Lorenz (Eds.), Application and Theory of Petri Nets and Concurrency - 44th International Conference, PETRI NETS 2023, Lisbon, Portugal, June 25-30, 2023, Proceedings, volume 13929 of *Lecture Notes in Computer Science*, Springer Nature Switzerland AG, Cham, Switzerland, 2023, pp. 217–228. URL: https://doi.org/10.1007/978-3-031-33620-1_12.

[31] M. Duvigneau, Konzeptionelle Modellierung von Plugin-Systemen mit Petrinetzen, volume 4 of *Agent Technology – Theory and Applications*, Logos Verlag, Berlin, 2010. URL: http://www.logos-verlag.de/cgi-bin/engbuchmid?isbn=2561&lng=eng&id=.

[32] J. H. Röwekamp, M. Taube, P. Mohr, D. Moldt, Cloud native simulation of reference nets, in: M. Köhler-Bußmeier, E. Kindler, H. Rölke (Eds.), Proceedings of the International Workshop on Petri Nets and Software Engineering 2021 co-located with the 42nd International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2021), Paris, France, June 25th, 2021 (due to COVID-19: virtual conference), volume 2907 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2021, pp. 85–104. URL: http://ceur-ws.org/Vol-2907.

[33] T. Bergs, S. Gierlings, T. Auerbach, A. Klink, D. Schraknepper, T. Augspurger, The Concept of Digital Twin and Digital Shadow in Manufacturing, Procedia CIRP 101 (2021) 81–84.

[34] A. Ferscha, S. K. Tripathi, Parallel and distributed simulation of discrete event systems, Citeseer, 1998.

[35] A. Varga, Discrete event simulation system, in: Proc. of the European Simulation Multiconference (ESM'2001), volume 17, 2001, p. 7.

[36] F. Ahmad, M. T. Chaudhry, M. H. Jamal, M. A. Sohail, D. Gavilanes, M. Masias Vergara, I. Ashraf, Formal modeling and analysis of RPL protocol using colored Petri nets, PLOS ONE 18 (2023) e0285700. doi:10.1371/journal.pone.0285700.

[37] M. Moradi, A. H. Jahangir, S. Sharifian, A Petri net model for Time-Delay Attack detection in Precision Time Protocol-based networks, IET Cyber-Physical Systems: Theory & Applications 9 (2024) 407–423. doi:10.1049/cps2.12088.

[38] S. Yau, M. Caglayan, Distributed Software System Design Representation Using Modified Petri Nets , IEEE Transactions on Software Engineering 9 (1983) 733–745. URL: https://doi.ieeecomputersociety.org/10.1109/TSE.1983.235581. doi:10.1109/TSE.1983.235581.

[39] W. Zhong, J. Zhou, T. Sun, Concurrent software fine-coarse-grained automatic modelling by coloured Petri nets for model checking, IET Software 17 (2023) 55–75. doi:10.1049/sfw2.12084.

[40] W. M. P. van der Aalst, Business process management as the "killer app" for Petri nets, Software and Systems Modeling 14 (2015) 685–691. doi:10.1007/s10270-014-0424-2.

[41] S. Medina-García, E. J. Ortiz-Soria, L. A. Medina-Hernández, C. A. Muñoz-Galeano, L. J. Hernández-Zúñiga, A Petri net-based model for business process simulation, Applied Sciences 13 (2023) 11192. doi:10.3390/app132011192.

[42] I. Grobelna, A. Karatkevich, Challenges in application of Petri nets in manufacturing systems, Electronics 10 (2021) 2305. doi:10.3390/electronics10182305.

[43] M. Dotoli, G. Cavone, C. Seatzu, A survey on Petri nets models for logistics and transportation systems, IEEE CSS TC-DES Virtual Tutorial Series, 2021. URL: https://ieeecss.org/presentation/tcdes-tutorial/survey-petri-nets-models-logistics-and-transportation-systems, online tutorial page referencing the corresponding IEEE T-ITS survey.

[44] I. Grobelna, Model checking of reconfigurable fpga modules specified by petri nets, Journal of Systems Architecture 89 (2018) 1–9.

[45] T. Brant-Ribeiro, R. D. Araújo, I. E. Mendonça, M. S. Soares, R. G. Cattelan, Interactive web interfaces modeling, simulation and analysis using colored petri nets, Software & Systems Modeling 18 (2019) 721–737.

[46] A. Santos Filho, R. J. Rodríguez, E. L. Feitosa, Automated broken object-level authorization attack detection in rest apis through openapi to colored petri nets transformation, International Journal of Information Security 24 (2025) 83.

[47] U. Manne, R. Scholar Y, Horizontal vs. vertical scaling in modern database systems: A comparative analysis of performance and cost trade-offs, INTERNATIONAL JOURNAL OF COMPUTER ENGINEERING & TECHNOLOGY 15 (2024) 514–524. doi:10.5281/zenodo.13851050.

[48] A. Gandhi, P. Dube, A. Karve, A. Kochut, L. Zhang, Modeling the impact of workload on cloud resource scaling, in: 2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing, 2014, pp. 310–317. doi:10.1109/SBAC-PAD.2014.16.

[49] K. Razavi, M. Salmani, M. Mühlhäuser, B. Koldehofe, L. Wang, A tale of two scales: Reconciling horizontal and vertical scaling for inference serving systems, CoRR abs/2407.14843 (2024). URL: https://doi.org/10.48550/arXiv.2407.14843.

[50] I. Mitrani, Simulation techniques for discrete event systems, volume 14, Cambridge University

Press, 1982.
[51] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (1978) 558–565. URL: https://doi.org/10.1145/359545.359563. doi:10.1145/359545.359563.
[52] N. A. Lynch, Distributed Algorithms, Morgan Kaufmann, 1996.
[53] M. J. Fischer, N. A. Lynch, M. S. Paterson, Impossibility of distributed consensus with one faulty process, Journal of the ACM 32 (1985) 374–382. doi:10.1145/3149.214121.
[54] Foundation for Intelligent Physical Agents, FIPA interaction protocol library specification, 2000. URL: http://fipa.org/specs/fipa00025/PC00025C.html.
[55] J. J. Odell, H. Van Dyke Parunak, B. Bauer, Representing agent interaction protocols in uml, in: Agent-Oriented Software Engineering: 2000 Revised Papers, Springer, 2001, pp. 121–140.
[56] L. Cabac, Modeling Petri Net-Based Multi-Agent Applications, Dissertation, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, 2010. URL: https://ediss.sub.uni-hamburg.de/handle/ediss/3691.
[57] L. Cabac, D. Moldt, Formal semantics for AUML agent interaction protocol diagrams, in: J. Odell, P. Giorgini, J. P. Müller (Eds.), The Fifth International Workshop on Agent-Oriented Software Systems (AOSE-2004). Proceedings, Columbia University, New York, USA, 2004, pp. 97–111. URL: http://dx.doi.org/10.1007/978-3-540-30578-1_4.
[58] W. M. P. van der Aalst, Verification of workflow nets, in: P. Azéma, G. Balbo (Eds.), Application and Theory of Petri Nets 1997, 18th International Conference, ICATPN '97, Toulouse, France, June 23-27, 1997, Proceedings, Lecture Notes in Computer Science, Springer, 1997, pp. 407–426. URL: https://doi.org/10.1007/3-540-63139-9_48. doi:10.1007/3-540-63139-9_48.
[59] H. M. W. Verbeek, T. Basten, W. M. P. van der Aalst, Diagnosing workflow processes using Woflan, Comput. J. 44 (2001) 246–279.
[60] D. M. Chapiro, Globally-asynchronous locally-synchronous systems, Ph.D. thesis, Stanford University, USA, 1985. URL: https://searchworks.stanford.edu/view/1137794.
[61] P. Teehan, M. R. Greenstreet, G. G. Lemieux, A survey and taxonomy of GALS design styles, IEEE Des. Test Comput. 24 (2007) 418–428. URL: https://doi.org/10.1109/MDT.2007.151. doi:10.1109/MDT.2007.151.
[62] M. Krstic, E. Grass, F. K. Gürkaynak, P. Vivet, Globally asynchronous, locally synchronous circuits: Overview and outlook, IEEE Des. Test Comput. 24 (2007) 430–441. URL: https://doi.org/10.1109/MDT.2007.164. doi:10.1109/MDT.2007.164.
[63] G. A. Pratt, J. Nguyen, Distributed synchronous clocking, IEEE Trans. Parallel Distributed Syst. 6 (1995) 314–328. URL: https://doi.org/10.1109/71.372779. doi:10.1109/71.372779.
[64] T. Jacob, Implementierung einer sicheren und rollenbasierten Workflow"=Managementkomponente für ein Petrinetzwerkzeug, Diploma thesis, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, D-22527 Hamburg, 2002.
[65] W. M. P. van der Aalst, K. M. van Hee, A. H. M. ter Hofstede, N. Sidorova, H. M. W. Verbeek, M. Voorhoeve, M. T. Wynn, Soundness of workflow nets with reset arcs, Transactions on Petri Nets and Other Models of Concurrency III 3 (2009) 50–70. URL: https://doi.org/10.1007/978-3-642-04856-2_3.
[66] W. Aalst, The application of Petri nets to workflow management, Journal of Circuits, Systems, and Computers 8 (1998) 21–66. doi:10.1142/S0218126698000043.
[67] E. Kindler, W. M. van der Aalst, Liveness, fairness, and recurrence in petri nets., Inf. Process. Lett. 70 (1999) 269–27.
[68] H. Carstensen, R. Valk, Infinite behaviour and fairness in petri nets, in: G. Rozenberg (Ed.), Advances in Petri Nets 1984, Springer Berlin Heidelberg, Berlin, Heidelberg, 1985, pp. 83–100.
[69] P. Leach, M. Mealling, R. Salz, A universally unique identifier (uuid) urn namespace, 2005. URL: https://dn720004.ca.archive.org/0/items/rfc4122/rfc4122.txt.pdf.
[70] T. Bray, Rfc 8259: The JavaScript Object Notation (JSON) Data Interchange Format, 2017.
[71] L. Clasen, S. Bartelt, Y. Stahl, D. Moldt, Distributed P/T net simulation prototypes based on event streaming, in: M. Köhler-Bußmeier, D. Moldt, H. Rölke (Eds.), Proceedings of the International Workshop on Petri Nets and Software Engineering 2024 co-located with the 45th International

Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2024), June 24 - 25, 2024, Geneva, Switzerland, volume 3730 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2024, pp. 192–216. URL: https://ceur-ws.org/Vol-3730.

[72] Y. Stahl, Prototypische Umsetzung eines Kommunikationsprotokolls für die verteilte Simulation von P/T Netzen innerhalb des Petrinetz Simulators Renew, Bachelor thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, 2024.

[73] S. Bartelt, Prototypische Umsetzung eines Simulators als Komponente von verteilter Simulation von P/T Netzen in Renew, Bachelor thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, 2024.

[74] W. M. P. van der Aalst, Verification of workflow nets, in: P. Azéma, G. Balbo (Eds.), Application and Theory of Petri Nets 1997, 18th International Conference, ICATPN '97, Toulouse, France, June 23-27, 1997, Proceedings, Lecture Notes in Computer Science, Springer, 1997, pp. 407–426. URL: https://doi.org/10.1007/3-540-63139-9_48. doi:10.1007/3-540-63139-9_48.

[75] S. Christensen, L. Petrucci, Towards a modular analysis of coloured petri nets, in: K. Jensen (Ed.), Application and Theory of Petri Nets 1992, Springer Berlin Heidelberg, Berlin, Heidelberg, 1992, pp. 113–133.

[76] P. H. K. Jensen, R. M. Shapiro, Hierarchies in coloured petri nets, in: G. Rozenberg (Ed.), Advances in Petri Nets 1990, volume 383 of *Lecture Notes in Computer Science*, Springer-Verlag, 1990, pp. 313–341.

[77] K. Jensen, Coloured petri nets: A high level language for system design and analysis, in: G. Rozenberg (Ed.), Advances in Petri Nets 1990, volume 383 of *Lecture Notes in Computer Science*, Springer-Verlag, 1990, pp. 342–416.

[78] D. Buchs, Co-opn : A concurrent object oriented approach, LNCS 524 (1991) 432–454. URL: https://cir.nii.ac.jp/crid/1573950400279264256.

[79] J. Vautherin, Parallel systems specifications with coloured petri nets and algebraic specifications, in: G. Rozenberg (Ed.), Advances in Petri Nets 1987, Springer Berlin Heidelberg, Berlin, Heidelberg, 1987, pp. 293–308.

[80] O. Biberstein, D. Buchs, N. Guelfi, Object-Oriented Nets with Algebraic Specifications: The CO-OPN/2 Formalism, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, pp. 73–130. URL: https://doi.org/10.1007/3-540-45397-0_3. doi:10.1007/3-540-45397-0_3.

[81] D. Buchs, N. Guelfi, A formal specification framework for object-oriented distributed systems, IEEE Transactions on Software Engineering 26 (2000) 635–652. doi:10.1109/32.859532.

[82] S. Chachkov, D. Buchs, From formal specifications to ready-to-use software components: the concurrent object oriented petri net approach, in: Proceedings Second International Conference on Application of Concurrency to System Design, 2001, pp. 99–110. doi:10.1109/CSD.2001.981768.

[83] L. Voß, S. Willrodt, D. Moldt, M. Haustermann, Between expressiveness and verifiability: P/T-nets with synchronous channels and modular structure, in: M. Köhler-Bußmeier, D. Moldt, H. Rölke (Eds.), Proceedings of the International Workshop on Petri Nets and Software Engineering 2022 co-located with the 43rd International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2022), Bergen, Norway, June 20th, 2022, volume 3170 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2022, pp. 40–59. URL: https://ceur-ws.org/Vol-3170.

[84] M. Simon, Concept and Implementation of Distributed Simulations in Renew, Bachelor thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, 2014.

[85] L. Clasen, C. Nayci, D. Moldt, Distributed reference net simulation based on event streaming, in: E. G. Amparore, L. Mikulski (Eds.), Application and Theory of Petri Nets and Concurrency - 46th International Conference, PETRI NETS 2025, Paris, France, June 22-27, 2025, Proceedings, volume 15714 of *Lecture Notes in Computer Science*, Springer, 2025, pp. 130–154. URL: https://doi.org/10.1007/978-3-031-94634-9_7. doi:10.1007/978-3-031-94634-9_7.

[86] M. Haustermann, D. Mosteller, D. Moldt, Model checking of synchronized domain-specific multi-formalism models using high-level Petri nets, in: D. Buchs, J. Carmona (Eds.), Application and Theory of Petri Nets and Concurrency - 42nd International Conference, PETRI NETS 2021, Virtual

Event, June 23-25, 2021, Proceedings, volume 12734 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 230–249. URL: https://doi.org/10.1007/978-3-030-76983-3_12.

[87] L. Clasen, C. Nayci, E. Nayci, J. Middendorf, T. Mack, Investigations towards dynamic scaling of distributed P/T nets, in: M. Köhler-Bußmeier, D. Moldt, H. Rölke (Eds.), Proceedings of the International Workshop on Petri Nets and Software Engineering 2025 co-located with the 46th International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2025), June 22 - 27, 2025, Paris, France, volume 3998 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2025, pp. 124–144. URL: https://ceur-ws.org/Vol-3998/paper08.pdf.

[88] L. Clasen, P. Leonhardt, L. Wichelmann, Resilient distributed P/T net simulators, in: M. Köhler-Bußmeier, D. Moldt, H. Rölke (Eds.), Proceedings of the International Workshop on Petri Nets and Software Engineering 2025 co-located with the 46th International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2025), June 22 - 27, 2025, Paris, France, volume 3998 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2025, pp. 60–80. URL: https://ceur-ws.org/Vol-3998/paper04.pdf.