POWVER

Technical Report 2019-07

Title: **Concurrent Programming from pseuCo to Petri**

Authors: Felix Freiberger, Holger Hermanns

Report Number: 2019-07

ERC Project: Power to the People. Verified.

ERC Project ID: 695614

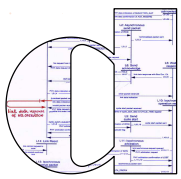
Funded Under: H2020-EU.1.1. – EXCELLENT SCIENCE

Host Institution: Universität des Saarlandes, Dependable Systems and Software
Saarland Informatics Campus

Published In: PETRI NETS 2019

Please cite this publication as follows:

Felix Freiberger, Holger Hermanns.
Concurrent Programming from pseuCo to Petri.
Application and Theory of Petri Nets and Concurrency - 40th International Conference, PETRI NETS 2019, Aachen, Germany, June 23-28, 2019, Proceedings. Lecture Notes in Computer Science 11522, Springer 2019, ISBN 978-3-030-21570-5: 279-297.



POWER TO THE PEOPLE.
VERIFIED.
erc

Concurrent Programming from pseuCo to Petri

Felix Freiberger[1,2] and Holger Hermanns[1]

[1] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
[2] Saarbrücken Graduate School of Computer Science, Saarland Informatics Campus, Saarbrücken, Germany

Abstract. The growing importance of concurrent programming has made practical concurrent software development become a cornerstone of many computer science curricula. Since a few years, a sound bridge from concurrency theory to concurrence practice is available in the form of pseuCo, a light-weight programming language featuring both message passing and shared memory concurrency. That language is at the core of an award-winning lecture at Saarland Informatics Campus. This paper presents a novel two-step semantic mapping from pseuCo programs to colored Petri nets, developed for the sake of further strengthening the educational concept behind pseuCo. The approach is fully integrated in pseuCo.com, our open-source teaching tool for pseuCo, empowering students to interact with the Petri-net-based semantics of pseuCo. In addition, we present a source-level exploration tool for pseuCo, also based on this semantics, that gives users an IDE-like debugging experience while enabling full control over the nondeterminism inherent in their programs. The debugger is also part of pseuCo.com, allowing students to access it without any set-up.

Keywords: Concurrency · Education · Colored Petri nets · Programming · Semantics

1 Introduction

Over the past decades, concurrent computation has grown tremendously in importance within computer science. This concerns both the theoretical modeling of concurrent systems with formalisms like Petri nets as well as the practical development of concurrent programs in real-world programming languages. The latter has made modern concurrent programming an integral part of computer science education. The ACM curricula recommendations [1] advocate a strong educational component on "Parallel and Distributed Computing" and stipulate that "Communication and coordination among processes is rooted in the message passing and shared memory models of computing and such algorithmic concepts as atomicity, consensus, and conditional waiting".

At Saarland University in Saarbrücken, Germany, this is addressed since 2005 in the mandatory Bachelor-level *Concurrent Programming* lecture [7], which in 2013 was awarded with the German "Preis des Fakultätentages Informatik" for its

Listing 1. A message passing PseuCo program. Expressions having the form `c <! x` (lines 11, 18 and 21) send the value of `x` on channel `c`. Expressions having the form `<? c` (lines 4, 19 and 22) receive a value from channel `c`.

```
1  void factorial(intchan c) {
2      int z, j, n;
3      while (true) {
4          z = <? c; // receive input
5
6          n = 1;
7          for (j = z; j > 0 ; j--) {
8              n= n*j;
9          }
10
11         c <! n; // send result
12     };
13 }
14
15 mainAgent {
16     intchan cc;
17     agent a = start(factorial(cc));
18     cc <! 3;
19     int mid = <? cc;
20     println("3! evaluates to " + mid + ".");
21     cc <! mid;
22     println("(3!)! evaluates to " + (<? cc) + ".");
23 }
```

"innovative concept combining classical process calculi with practical programming challenges". At its core, the lecture revolves around PseuCo, a light-weight programming language featuring both message passing and shared memory concurrency concepts, with its message passing syntax being inspired by Go [10]. It includes support for data structures with condition synchronization. Listing 1 shows a small message passing PseuCo program that computes the factorial of the factorial of 3.

PseuCo is overarching the lecture topics and bridges from a theoretical part – introducing process calculi in the form of Milner's CCS [16] – to a practical part dealing with Go and Java. The latter is effectively accomplished by providing a transpiler from PseuCo to Java source code which can either be output for inspection or immediately be compiled to Java byte code and executed.

To facilitate foundational reasoning about and analysis of concurrent programs, PseuCo also has a formal semantics mapping to value-passing CCS. A corresponding PseuCo-to-CCS-compiler, plus tools to facilitate analysis of the resulting transition system, are provided to students as part of an IDE called PseuCo.com [4]. It is based on web technologies to ensure it is easy to use for students. It requires no setup, updates automatically and works in all modern

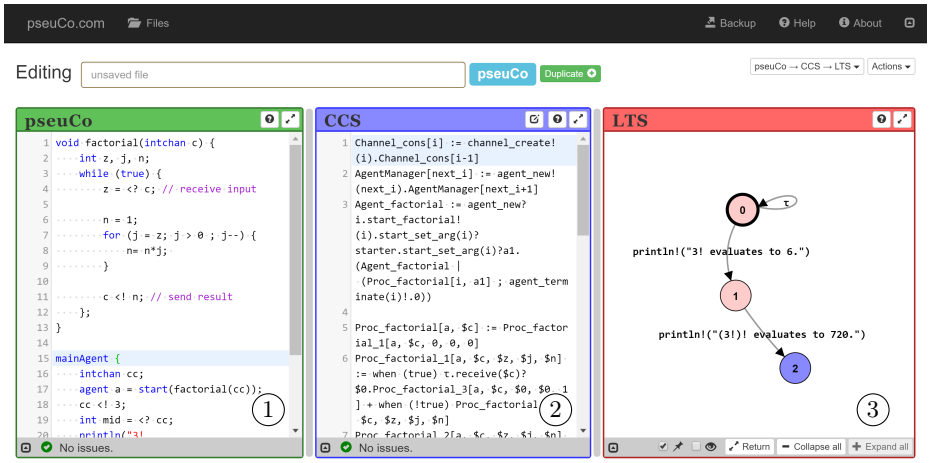


Fig. 1. PSEUCO.COM showing Listing 1 and the CCS-based semantics.

browsers including on tablets. All computations are performed on the client side (except for some advanced analysis tasks such as checks related to memory models) so the tool works offline after initial use. The Concurrent Programming lecture also uses PSEUCO.COM for the theoretical part as an IDE for CCS terms. Figure 1 shows a screenshot of PSEUCO.COM showing the program from Listing 1 ①, a fraction of its corresponding CCS term ② and the resulting LTS ③, minimized up to observational congruence.

Over the years, both PSEUCO and PSEUCO.COM have been subject to continuous improvement and have become an integral part of the annual lecture editions. While being an overall success, room for disruptive improvement has become apparent. Pragmatically speaking, this concerns readability of the compiler output and debugging. Conceptually speaking, it concerns a structured and concurrency-preserving and hence more faithful formal semantics.

- Students typically have a very hard time understanding the CCS terms produced by the compiler because the latter uses several low-level hacks. Among them:
  - Control flow is resolved into goto-style [5] spaghetti-code in CCS syntax.
  - The CCS terms generated contain many static helper constructs, such as an `AgentManager`, responsible for assigning unique ids to agents.
  - Synchronous (handshaking) and asynchronous (buffered) communication channels are internally distinguished by mapping them to negative, respectively positive integer identifiers. Every time a channel is used, the CCS term branches depending on the sign of the channel identifier.
- PSEUCO.COM lacks debugging support. It can show the LTS induced by the CCS term (induced by the PSEUCO program), but this does not provide an adequate debugging experience. The LTS is notoriously large, very often too large to grasp – Listing 1 already induces an LTS with 48 states.

– The students lack a feeling of the true concurrency inherent to a pseuCo program since the CCS translation comes with an interleaving semantics.

All in all, the lesson the students learn is thus typically restricted to "OK, there is a formal – but messy – semantics", instead of "There is a natural way of giving a formal concurrency semantics to a concurrent programming language".

A deeper analysis of the problems led to the insight that all these problems can be overcome by instead providing a Petri-net-based semantics for pseuCo. This is indeed what the paper develops. It describes a novel extension to pseuCo.com that aims at (i) providing easy-to-understand compiler output, (ii) providing a debugging experience that matches the usability and feature set of classic IDEs while being based on a complete semantics preserving nondeterminism, allowing full exploration of all nondeterministic possibilities during debugging, (iii) exposing students to Petri nets as a natural true concurrency formalism, and (iv) laying the basis for analysis of pseuCo programs using Petri net techniques. At the core of this work is a two-level formalization of the semantics in terms of colored Petri nets.

Related Work. Higher-level Petri nets are an attractive base for the formal semantics of programming languages or process calculi. Among the pioneering works, $B(PN)^2$ [3] has been proposed as a concurrent programming notation geared towards Petri nets. A compositional semantics maps $B(PN)^2$ to M-Nets, a Petri net dialect specifically designed as a vehicle for giving semantics to concurrent programming languages [2]. Just like our approach, M-Nets are based on *colored* Petri nets. They support CCS-style composition which is coherent with their unfolding operation. This is orthogonal to the approach presented here, which does not provide composition operations, but focuses on providing a Petri-net-based semantics for a programming language closely resembling traditional imperative programming, together with tool support for use in teaching. Petruchio [15] is a tool-supported approach that focuses on dynamically changing communication structures in Petri nets, especially rooted in the π-calculus [17]. It comes with a translation of the latter into Petri nets, so as to enable analysis with net verification tools. More recently, Nested-Unit Petri Nets [9] have been proposed as an extension of (uncolored) Petri nets, to be used when mapping compositional, process calculi-inspired programs to Petri nets. The addition of units allows more efficient storage of markings, speeding up analysis in the presence of appropriately-defined units.

Organization of the Paper. Section 2 reviews the main features of the pseuCo programming language. Section 3 introduces colored Petri nets and a JavaScript library for handling them. Section 4 introduces $C^{P}_{P}N$, a higher-level Petri net notation that is used as an intermediate step, its implementation and a translation from pseuCo to $C^{P}_{P}N$. Section 5 details how this translation and a debugger based on it are included in the pseuCo.com web application. Finally, Section 6 concludes this paper.

2 PSEUCO in a Nutshell

To set the stage for what follows, this section reviews the most important aspects of the PSEUCO language design, closely following its presentation in [4].

Mainstream programming is nowadays dominated by imperative programming languages. PSEUCO is an imperative language featuring a heavily simplified Java-like look and feel paired with language concepts inspired by the Go programming language [10]. It also has similarities with Holzmann's Promela language [11].

A very simplistic PSEUCO example is depicted in Listing 2. This program implements concurrent counting. A shared integer, `n`, is initialized to `10`. The procedure `countdown()` decrements this counter five times. The `mainAgent`, which is run when the program is started, starts a second agent that runs `countdown()` before calling `countdown()` itself. After both agents have executed this procedure, the `mainAgent` prints the final value of `n`. To ensure mutually exclusive access to the shared variable, a globally defined `lock` named `guard_n` is used within the `countdown()` procedure.

PSEUCO also provides native support for message passing concurrency. An example is presented in Listing 1. An agent running the procedure `factorial` interacts via a channel with the `mainAgent`. In a nutshell, `factorial` computes the factorial of a number received from channel `c` and reports the result on the same channel `c`. This channel is declared locally in line 16 and passed as a parameter of `factorial`. Its type `intchan` indicates that it accepts integers and is unbuffered, meaning that it induces a handshake between the agents sending to (via `<!`) and receiving from (via `<?`) it. PSEUCO also has channels that can hold

Listing 2. Shared memory concurrent counting in PSEUCO.

```
1  int n = 10;
2  lock guard_n;
3
4  void countdown() {
5      for (int i = 5; i >= 1; i--) {
6          lock(guard_n);
7          n--;
8          unlock(guard_n);
9      }
10 }
11
12 mainAgent {
13     agent a = start(countdown());
14     countdown();
15     join(a);
16     println("The␣value␣is␣" + n);
17 }
```

Listing 3. Replacement for lines 4 to 11 in Listing 1.

```
1  select {
2      case <? t: {
3          return;
4      }
5      case z = <? c: { // lines 6 to 11 identical to Listing 1
6          n = 1;
7          for (j = z; j > 0 ; j--) {
8              n= n*j;
9          }
10
11         c <! n; // send result
12     }
13 };
```

strings or Booleans. After starting the agent, the `mainAgent` feeds the number 3 into the channel `cc` and then waits for results to be sent back to him. The result is returned back to the `factorial` agent. After the second round, the main agent prints the result.

This program does not terminate the `factorial` agent. Explicit termination can be achieved by applying three changes. First, the expansion uses a new channel declared by inserting `boolchan2 t;` before line 1. This channel is a FIFO buffer which can hold up to 2 Booleans. Second, the main agent is instructed to send a message on that channel at the end of its execution by inserting `t <! true;` after line 22. Finally, the `factorial` agent may now receive a message on two different channels (`t` and `c`) and therefore a `select-case` statement is used to specify dedicated reactions by replacing lines 4 to 11 with Listing 3. In case any message on `t` is received, the agent immediately terminates. Otherwise, it proceeds as previously. PseuCo has borrowed the `select-case` concept from Go [10]. A `select` statement consist of several `case`s. Except for `default` cases, each case has a guard and a statement. The guard contains exactly one send (`<!`) or receive operation (`<?`). At runtime, a case can be selected only if the message passing operation of the guard is possible, i.e. if the channel can be read or be written to, respectively. One of those cases is selected nondeterministically and its guard and statement are processed. A `default` case can always be selected. If there are multiple cases that can be selected, one of them is selected nondeterministically.

These examples give an impression of the features provided by PseuCo, all of which are given semantics by translation to CCS. In addition, PseuCo supports arrays, `struct`s and `monitor`s with condition synchronization, however, these can be viewed as syntactic sugar and are not considered in this paper.

3 A Library for Colored Petri Nets

PSEUCO is an imperative programming language, and as such any PSEUCO program operates on variables. In this context, colored Petri nets [12] offer a clear advantage over basic Petri nets as a semantic model for the language.

Colored Petri Nets. We generally follow the definitions from [13] and assume any syntax for expressions where *Exp* is the set of expressions, *Types* is a set of types and *Vars* is a set of variables. Let $\mathbb{B} \in \mathit{Types}$ be the set of Booleans. Let $\mathit{Values} \coloneqq \bigcup_{t \in \mathit{Types}} t$ be the set of all values. We use $\mathit{Type} : \mathit{Vars} \to \mathit{Types}$ to express the type of a variable and $\mathit{Type} : \mathit{Exp} \to \mathit{Types}$ for the type of an expression. For a set of variables *Vars*, let $\mathit{Type}(\mathit{Vars})$ denote the set of types $\{\mathit{Type}(v) \mid v \in \mathit{Vars}\}$. We assume a function $\mathit{Var} : \mathit{Exp} \to \mathit{Vars}$ that returns the variables in an expression. A *binding* b is a function $b : \mathit{Vars} \to \mathit{Values}$ such that $\forall v \in \mathit{Vars} : b(v) \in \mathit{Type}(v)$. Let *Bindings* be the set of bindings. Lastly, we assume an evaluation function $\mathit{eval} : \mathit{Exp} \times \mathit{Bindings} \to \mathit{Values}$ such that $\forall e \in \mathit{Exp} : \forall b \in \mathit{Bindings} : \mathit{eval}(e, b) \in \mathit{Type}(e)$. When evaluating closed expressions, we omit the second argument to *eval*. Let X_{MS} be the set of all multisets over X.

Definition 1 (Colored Petri net). *A colored Petri net is a tuple* $\mathit{CPN} = (\Sigma, P, T, A, N, C, G, E, I)$ *satisfying the requirements below:*

(i) Σ *is a finite set of non-empty types, called* color sets.
(ii) P, T *and* A *are pairwise disjoint sets of* places, transitions *and* arcs.
(iii) $N : A \to P \times T \cup T \times P$ *is a* node *function.*
(iv) $C : P \to \Sigma$ *is a* color *function.*
(v) $G : T \to \mathit{Exp}$ *is a* guard *function such that* $\forall t \in T : \mathit{Type}(G(t)) = \mathbb{B} \wedge \mathit{Type}(\mathit{Var}(G(t))) \subseteq \Sigma$.
(vi) $E : A \to \mathit{Exp}$ *is an* arc expression *function such that* $\forall a \in A : \mathit{Type}(E(a)) = C(p(a))_{MS} \wedge \mathit{Type}(\mathit{Var}(E(a))) \subseteq \Sigma$ *where* $p(a)$ *is the place of* $N(a)$.
(vii) $I : P \to \mathit{Exp}$ *is an* initialization *function such that* $\forall p \in P : \mathit{Type}(I(p)) = C(p)_{MS}$ *and* $\forall p \in P : \mathit{Var}(I(p)) = \emptyset$, *i.e. all expressions returned by* I *are closed.*

CPN in JavaScript. Colored Petri nets will serve as a semantic model for PSEUCO, and tool support for experiencing and exploring this semantics is at the core of our educational approach. For this purpose, we provide a JavaScript library to express and evaluate colored Petri nets, which we will call `colored-petri-nets`. The library implements support for the concepts needed to materialize Definition 1 and its semantic underpinning by introducing a syntax for arc expressions, implementing a data structure for Petri nets and providing an algorithm for finding enabled steps. In addition, for nets where only a finite number of values are reachable, it allows converting the colored Petri net into a basic Petri net. For simplicity, the library enforces some restrictions and simplifications:

1. In Definition 1, arc expressions evaluate to a multiset of colors. Our expression syntax only allows expressing single colors, which are always treated as singleton multisets. Multiple tokens can be consumed or produced by the use of multiple arcs.
2. Usually, colored Petri nets are defined over an arbitrary set Σ of color sets (or types). Our implementation, however, only supports a single type, $\Sigma = \{\mathbb{V}\}$, with $\mathbb{V}$ containing numbers, booleans, arrays and objects (where keys are strings and values are valid colors).
3. In full generality, both arcs from places to transitions ("incoming" arcs) and arcs from transitions to places ("outgoing" arcs) are inscribed with the same kind of expression. However, allowing arbitrary expressions on incoming arcs complicates computing the set of enabled markings because the binding are to be guessed or deduced. CPN tools [13] solve this problem by using a sophisticated algorithm [14] to compute the values of variables that can be deduced and by restricting unbounded variables to *small color sets*. For reasons of simplicity, we instead use a restricted pattern syntax on incoming arcs. When given token colors to read, these patterns evaluate to partial bindings. If they are compatible, they combine to the single binding under which the guard and the outgoing arcs' expressions are evaluated, assuming no unbound variables remain. While this restricts a single token read to return only one specific binding, nondeterminism can be retrieved by allowing expressions with inherent nondeterminism or by duplicating transitions.

The restricted pattern syntax for incoming arcs is inspired by JavaScript [6], especially by valid left-hand sides of JavaScript assignments. Various extensions and modifications aim to provide a more complete set of matching capabilities. The constructs in Table 1 can be used in patterns.

For outgoing arcs, the syntax and semantics of expressions are also based on a fragment of JavaScript, but feature some additions. These mostly serve the purpose of increasing the expressiveness without needing to allow procedural code. Just as in traditional JavaScript, we support (i) conditionals (`x ? 42 : 1337`); (ii) logical *or* (`||`) and *and* (`&&`); (iii) equality (`==`) and inequality (`!=`) checks; (iv) numerical comparison (`>`, `<`, `>=` and `<=`); (v) division-free basic arithmetic (`+`, `-` and `*`); (vi) boolean negation (`!`); (vii) property access (`point.coordinates.x`), including the `length` property of arrays; (viii) grouping with `(` and `)`; (ix) integer and boolean literals; (x) variables; (xi) array literals (`[4, 5, 6]`); and (xii) object literals (`{ a: 1, b: 2 }`), including ES6-style shorthand notation (`{ a, b }`). In addition, support is provided for (xiii) array concatenation via the new `@` operator; (xiv) spreads in object literals (`{ a: 1, ...x, b: 2, ...y, a: 2 }`) which copy in all keys and values from another object, using the rightmost value if a key is duplicated; and (xv) an evaluation function (`eval(a + b, vars)`) that evaluates a subexpression in a separate environment that is passed in object form as the second argument.

The JavaScript library `colored-petri-nets` is freely available at https://dgit.cs.uni-saarland.de/pseuco/colored-petri-nets.

Table 1. Constructs allowed in `colored-petri-nets` patterns.

Name	Example	Description
variable	`x`	matches anything and binds the value
wildcard	`_`	matches anything and drops the value
array patterns	`[a, b, c]`	matches any array of matching length and recursively matches the components
empty slots in array patterns	`[a, , c]`	act as a wildcard
spread in array pattern	`[hd, ...tl]`	matches the remainder of the array, only allowed in the last slot
object pattern	`{ a, b: c }`	matches objects having all required keys and matches the values to the pattern, if one is specified, or binds it to a variable named like the key
spread in object pattern	`{ one, two, ...rest }`	matches all unmentioned keys into a new object, only allowed in the last slot

4 Augmenting CPN for Concurrent Programming

As discussed in Section 1, there are three problems impeding readability of the CCS terms produced by the current pseuCo to CCS compiler: (i) hard-to-follow program flow, (ii) an abundance of static helper constructs and (iii) the insertion of runtime logic into user code (e.g. for message passing). While the switch to Petri nets (and using a graphical representation for them) inherently improves on problem (i), without additional care, issues (ii) and (iii) would resurface.

For example, when considering message passing in pseuCo, Petri nets are obviously capable of expressing both synchronous and asynchronous message passing channels. However, since channel variables can be dynamically reassigned, static analysis cannot always determine whether a channel variable refers to a synchronous or an asynchronous channel, or whether two channel variables in different agents could refer to the same channel. Therefore, a naïve implementation would be bound to introducing a central storage place for the contents of all asynchronous channels. Each use of a channel would then have to perform a runtime check to determine the type of channel, and in the case of an asynchronous channel proceed by synchronizing with the central storage place. Such constructs blow up the resulting Petri net and impede readability. They also hinder graph layout by their introduction of highly interconnected places. In addition, these constructs are not specific to pseuCo, so we would like to make them reusable for compiling other programming languages to Petri nets.

To this end, we introduce an abstraction layer between pseuCo and colored Petri nets, called *colored program Petri nets* (*CPPN*). This is a high-level notation

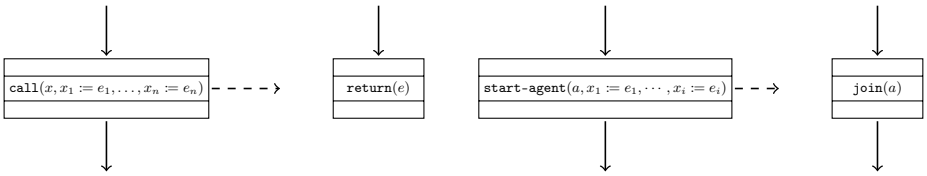


Fig. 2. Calling and agent management.

based on colored Petri nets tailored to concurrent programs using shared memory and/or message passing concurrency. As $C^{P}PN$ are meant as an easily visualizable and reusable intermediate form between an imperative concurrent programming language and Petri nets, we do not define an executable semantics for them, but instead provide a translational semantics to ordinary colored Petri nets.

4.1 $C^{P}PN$ Overview

At a glance, a $C^{P}PN$ is very similar to a colored Petri net. However, in $C^{P}PN$s, tokens are strictly associated with *agents* (or *threads*) on the program level, and token colors can be viewed as object valuations together representing the states of local variables of the agent. Agents have an identity that becomes relevant when waiting for an agent's termination or when handling reentrant locks. On the $C^{P}PN$ level, this is echoed by regular (i.e. CPN-typical) transitions being restricted to always consume and produce exactly one token, and the initial marking being constrained to contain a single token. In return, $C^{P}PN$ includes *command transitions* to handle (i) procedure calls, (ii) agent creation and management, (iii) message passing, (iv) global variables and (v) mutexes.

Calling and Agent Management (see Figure 2). For both `call` and `start-agent`, the x_i are local variables representing the arguments of the called procedure and the e_i are expressions representing their values. For `call`, x is the variable that captures the return value. Similarly, for `start-agent`, a captures the identity of the newly started agent. This allows waiting for the termination of the agent with a `join`(a) transition. For `return`, e is an expression representing the return value.

Note that `call` and `start-agent` have two outgoing arcs, one of which is dashed. The dashed arc represents the called procedure or the newly started agent, while the solid arc represents the caller.

Message Passing Support (see Figure 3). `init-chan` creates a new channel of capacity c, assigned to variable x. Sending and receiving messages is handled by `send`, `receive` and `default` transitions. Any place that has an outgoing arc to such a transition may not have an outgoing arc to other transitions. The `default` transition is always allowed and allows bailing out of a place that has message passing transitions.

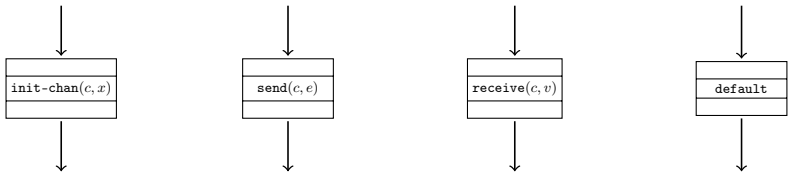


Fig. 3. Message passing support: Channel creation and sending/receiving messages.

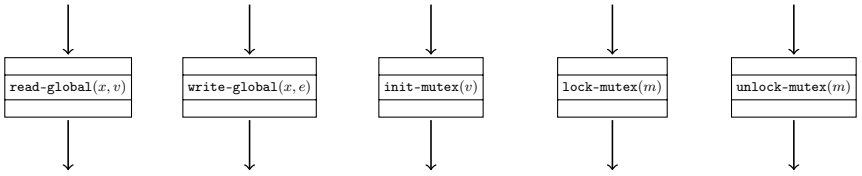


Fig. 4. Shared memory support: Global variables and mutexes.

Shared Memory Support (see Figure 4). CP_PNs support the use of global variables that all agents can access. These variables cannot be used directly but can be accessed using two dedicated commands: `read-global`(x, v) copies the value of the global variable x to the local variable v, and `write-global`(x, e) writes the result of evaluating the expression e to x. Coordination is possible through the use of *mutexes* (or *locks*). They are initialized with `init-mutex`, which saves their identity in variable v. Then, they can be used with `lock-mutex` and `unlock-mutex`.

Example. Figure 5 depicts a CP_PN for the PSEUCO program listed in Listing 1. In the visual representation, the three lines labeling the transition bodies indicate the name of the transition, which kind of command transition it is and the guard (which defaults to *true*).

Syntax of CP_PN. The ensemble of structures enabled by CP_PN is as follows. Let *GlobVars* be a set of identifiers for *global variables*. We define a set *Cmds* of *commands* that can be associated with transitions to make them command transitions. For example, the transition `factorial-send` in Figure 5 has the command `send`(c, n) associated with it, indicating that the result of evaluating the expression n is sent over the channel returned by evaluating the expression c. Similarly, we define commands for all types of command transitions appearing in Figures 2 to 4. Let *MPCommCmds* $\subset$ *Cmds* be the set of possible message passing commands, i.e. sending, receiving or `default` transitions. Similarly, *StartAgentCmds*, *CallCmds* and *ReturnCmds* refer to the corresponding respective subsets of *Cmds*.

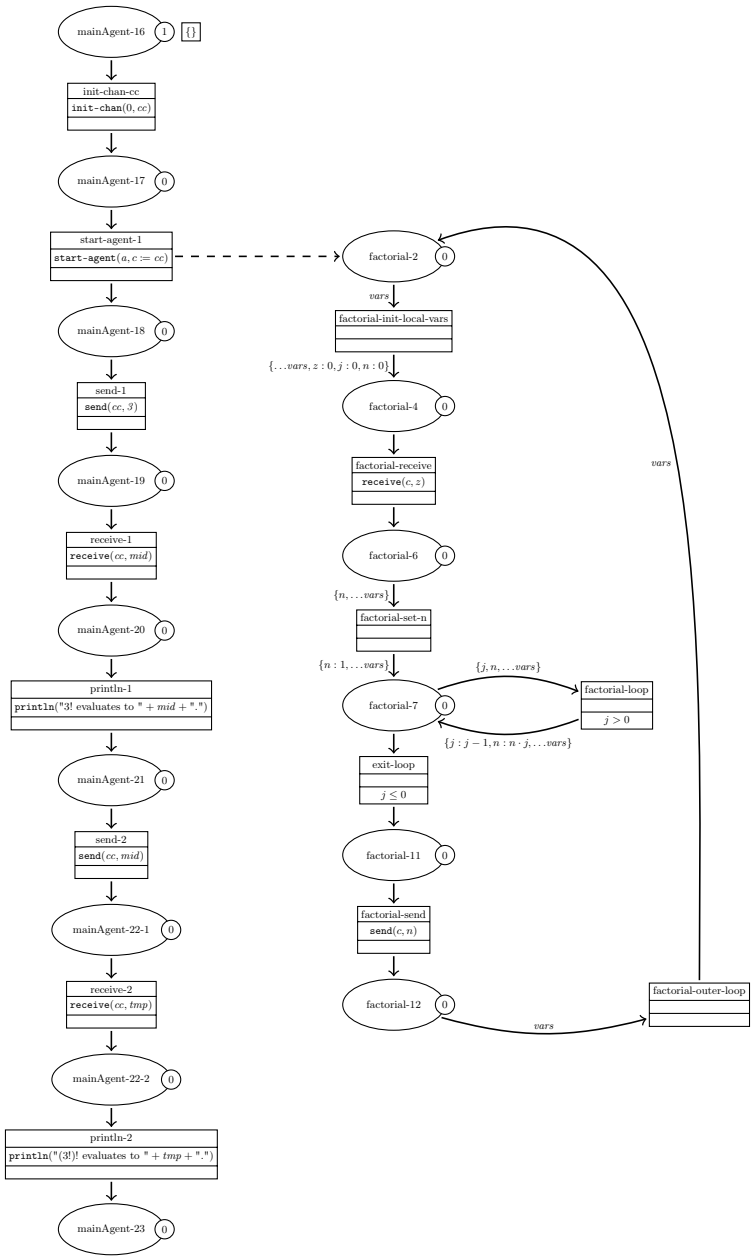


Fig. 5. A $C^{P}PN$ for Listing 1. The notation adheres to certain simplifications that are introduced in Section 4.2.

Definition 2 ($C^{P}_{P}N$). *A $C^{P}_{P}N$ is a tuple $CPN = (\Sigma, P, T, A, Cmd, S, N, C, G, E, I)$ satisfying the requirements below:*

(i) $(\Sigma, P, T, A, N, C, G, E, I)$ *is a colored Petri net.*

(ii) $Cmd : T \to Cmds \cup \{\bot\}$ *is a* command *function.*

(iii) $S : A \to \mathbb{B}$ *indicates for each arc whether it is starting a new procedure or agent.*

(iv) The arc expression function E only returns expressions that always evaluate to singleton multisets:

$$\forall a \in A : \forall b \in Bindings : |eval(E(a), b)| = 1.$$

(v) The initialization function specifies a single token:

$$\exists p \in P : |eval(I(p))| = 1 \wedge \forall p' \in P \setminus \{p\} : |eval(I(p))| = 0.$$

(vi) Let $post(x) \coloneqq \{y \mid \exists a \in A : N(a) = (x, y)\}$ denote the set of successors and $pre(x) \coloneqq \{y \mid \exists a \in A : N(a) = (y, x)\}$ denote the set of predecessors of a place or transition. For each transition, the number of incoming and outgoing arcs must be correct, in the following sense:

1. $\forall t \in T : |pre(t)| = 1$
2. $\forall t \in T : |post(t)| = \begin{cases} 2 & \text{if } Cmd(t) \in StartAgentCmds \cup CallCmds \\ 0 & \text{if } Cmd(t) \in ReturnCmds \\ 1 & \text{otherwise} \end{cases}$

For call and start transitions, exactly one arc must be marked as starting:

3. $\forall t \in T : |\{a \mid a \in post(t) \wedge S(a) = true\}| = 1$

(vii) Let $T_{MPComm} \coloneqq \{t \in T \mid Cmd(t) \in MPCommCmds\}$ be the set of message passing communication transitions. If a transition has an outgoing arc to any transition in T_{MPComm}, all outgoing arcs must lead to such transitions, i.e. $\forall p \in P : post(p) \cap T_{MPComm} \neq \emptyset \implies post(p) \subseteq T_{MPComm}$.

Relative to Definition 1, we can note the following differences:

- There is a command function (see (ii)) assigning commands to transitions. If $Cmd(T) = \bot$, we call T a *Petri transition*, otherwise, it is a *command transition*.
- The arc expression function must yield expressions that always return a single token. This, together with condition (vi), ensures that Petri transitions must consume and produce exactly one token at all times.
- The initial marking is now restricted to contain a single token.
- Condition (vi) and the new function S (see (iii)) ensure command transitions have the correct number of incoming and outgoing arcs, and `call` and `start-agent` transitions have one outgoing arc designated as the starting arc, represented by a dashed line in the graphical representation.
- An additional restriction, item (vii), ensures that any place that has an outgoing arc to a message passing command transition can only have arcs to such transitions.

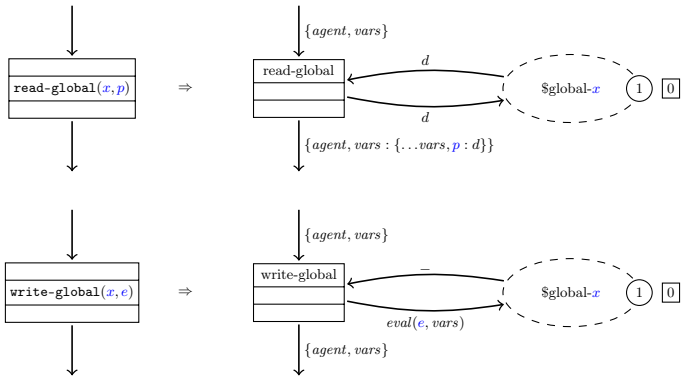


Fig. 6. Unfolding of shared memory command transitions. Each `read-global` and `write-global` command transition is replaced by the construct above, creating a new place global-x (initialized with a token of color 0) whenever a global variable named x is seen for the first time.

4.2 Translation from $C^{P}_{P}N$ to CPN

As mentioned previously, $C^{P}_{P}N$s do not posses an executable semantics, but instead are translated to regular colored Petri nets. At its core, the construction unfolds the command transitions into the structures corresponding to their intended functionality.

1. All arc expressions and the initial marking are updated by adding an `agent` property that contains an agent id and the current recursion depth, initially both 0. This enables e.g. `join` transitions to recognize agents and allows matching the token of a returning procedure to its caller. The color sets of all places are adjusted accordingly.
2. A fixed set of management places is added to handle id generation for locks, channels and agents, manage agent termination and store channel contents and lock states.
3. For each global variable, a management place is added to store its value.
4. Each command transition is replaced with a specific construct implementing the command functionality, typically by synchronizing with one or more management places. For example, Figure 6 shows the unfolding for shared memory command transitions. For message passing command transitions, these replacement constructs in addition can synchronize with each other to implement handshaking over synchronous channels. Similarly, each pair of `call` and `return` command transitions causes a linking transition to be inserted that handles returns from that specific `return` transition to that `call` transition.

The various details of the needed constructions are too verbose to include them in all detail here due to lack of space. A majority of these constructions result

in a linear increase in the size of the net, similar to Figure 6. However, overall, the size of the resulting CPN is quadratic in the number of `call`, `return`, `send` and `receive` transitions due to the additional transitions needed to handle handshaking and returns.

Implementation. The JavaScript library `colored-petri-nets` introduced in Section 3 has full support for C^PN and for the translation to colored Petri nets. On top of the restrictions and simplifications that apply to plain colored Petri nets (see Section 3), two additional simplifications are added: First, command transitions are not allowed to have a guard different from *true*. Second, arcs belonging to command transitions do not have arc inscriptions. Therefore, command transitions are not allowed to change the token color except in the way dictated by their command. For command transitions where arc inscriptions seem necessary, e.g. when passing arguments to a called procedure or newly started agent, the behavior is instead controlled by additional parameters within the command. The notation used in Figures 2 to 6 matches these simplifications.

4.3 From PSEUCO to C^PN

As per the design goals of C^PN, compiling PSEUCO programs to C^PN is rather straightforward. This task is taken care of by `pseuco-cpn-compiler`, a JavaScript-based implementation of such a compiler. The compiler starts with a net consisting of a single place, then simply traverses the abstract syntax tree of the input program, processing children in reverse order while building up the net from bottom to top. This direction is advantageous because it simplifies building a *source map*, indicating which program statement each place belongs to, as the compiler creates the place representing the program state *before* a certain statement while processing that statement. The JavaScript library `pseuco-cpn-compiler` is freely available at https://dgit.cs.uni-saarland.de/pseuco/pseuco-cpn-compiler. It currently supports the array- and structure-free subset of PSEUCO. In conjunction with the `colored-petri-nets` library, it allows compiling PSEUCO programs into regular colored Petri nets or, for PSEUCO programs with a bounded state space, basic Petri nets.

5 PSEUCO.COM: An Educational Tool Backed by Petri Nets

As previously discussed, the main motivation of this work has been to enhance PSEUCO.COM by providing a more easily digestible semantics of PSEUCO programs and by providing IDE-like debugging capabilities. This section details the result of these efforts.

Colored Petri Nets in PSEUCO.COM. To integrate the `colored-petri-nets` library and `pseuco-cpn-compiler` into the educational tool PSEUCO.COM, an appealing way is needed to visualize Petri nets. For the purpose of working with labeled transition systems, PSEUCO.COM does already employ a force-directed

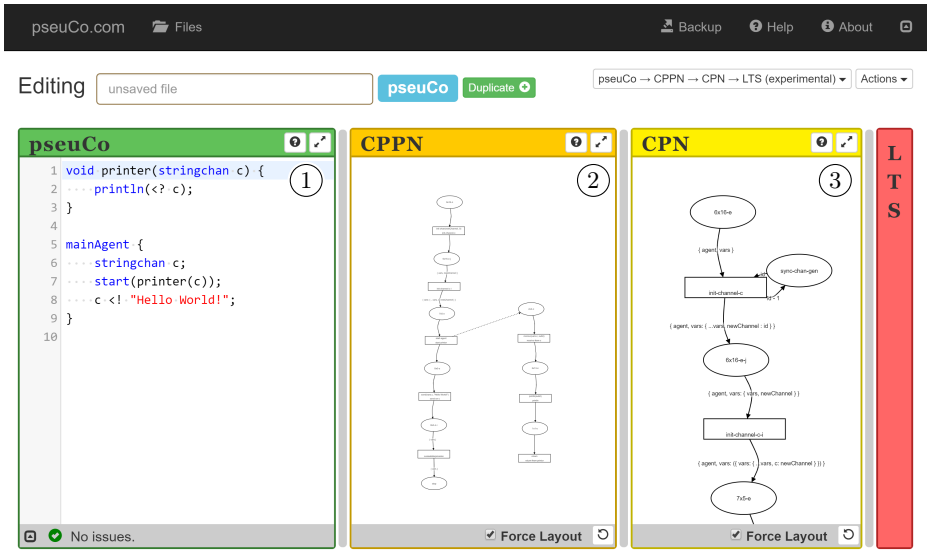


Fig. 7. PseuCo.com showing a sample pseuCo program and its Petri net semantics.

graph layout system allowing the user to interactively explore a transition system by expanding or collapsing states (i.e. showing and hiding their successors). Petri nets incur less need for interactive exploration as they tend to stay small even for pseuCo programs resulting in thousands of LTS states using the existing CCS-based compiler. Still, the integration of Petri nets into pseuCo.com uses a force-based graph layout, so as to allow users to influence the layout of the net by dragging and dropping, similarly to what is supported for LTS. In addition to standard forces like electrical charges and spring forces along arcs, pseuCo.com employs custom forces to orient graphs. They ensure that regular arcs typically point downwards, while called agents and procedures are positioned horizontally. This is demonstrated in Figure 7, showing a sample pseuCo program (1) and two graphs describing the corresponding $C^{P}PN$ (2) and a fragment of its unfolding (3). To speed up convergence, the force layout is initialized by node positions precomputed with `dagre` which employs an algorithm similar to Graphviz [8]. The force layout can be disabled to fall back to the static layout provided by `dagre`.

Debugging pseuCo *Programs in* pseuCo.com. The CCS-based semantics employed by pseuCo.com so far has been of little help for students seeking to understand the behavior of their programs. At its core, this problem is rooted in the difficulty of mapping a state of the resulting LTS to the state of the underlying pseuCo program, a process that requires parsing convoluted CCS terms and an understanding of the internal low-level hacks used by the pseuCo to CCS compiler. While switching to the Petri-net-based semantics alleviates this by replacing CCS terms with markings in a fixed net, significantly improving read-

ability, a partial understanding of the compiler's internals is still required. To resolve this, we present the pseuCo.com debugging feature. While designed to be usable even without an understanding of Petri nets, it is based upon the semantics described above and backed by the $C^{P}PN$-based pseuCo compiler included in pseuCo.com.

In its core, the debugger is a tool to explore the marking graph of the underlying Petri net. However, instead of showing the current marking directly, the debugger translates the marking into pseuCo terminology. Figure 8 shows the debugger in action, demonstrating its main features: It allows the user to

- inspect the console output (1),
- see running agents and their local variables and call stack (2),
- identify the statement an agent is currently executing (3),
- see global variables and the state of asynchronous channels and locks (not present in the example),
- see which agents are currently waiting for message passing synchronization to happen (4),
- single-step agents (5), manually resolving nondeterminism if present,
- automatically execute single agents (6) as long as their behavior is deterministic,
- automatically execute the whole program (7), resolving nondeterminism randomly,
- set breakpoints (8) to interrupt automatic execution and to
- return to any previous state of the program (9).

All of this functionality is rooted in the linkage between the pseuCo program and the Petri net levels, mentioned above. The compiler, `pseuco-cpn-compiler`, annotates its output $C^{P}PN$ with a source map allowing the debugger to link elements of the net to the original program. When converting the $C^{P}PN$ to a colored Petri net, the `colored-petri-nets` module preserves these annotations and generates additional metadata identifying the newly introduced places.

When inspecting a marking, the debugger uses the agent id and recursion depth embedded in the token colors (see Section 4.2) to identify running agents, their local variables and stack frames. The source map information of an agent tokens' place identifies that agent's position in the source code. Global variables and their values are identified by looking for places added by the replacement rule introduced in Figure 6 and tokens stored in them. Asynchronous channel contents and in-progress handshaking events are handled similarly.

In summary, this approach allows pseuCo.com to present a debugger that supports a similar feature set and user experience than a traditional IDE. Being built upon on the complete Petri-net-based semantics, however, allows preserving the full nondeterministic behavior of the program, ensuring that every execution of the program that can occur in practice can not only be reproduced but also specifically chosen in the debugger.

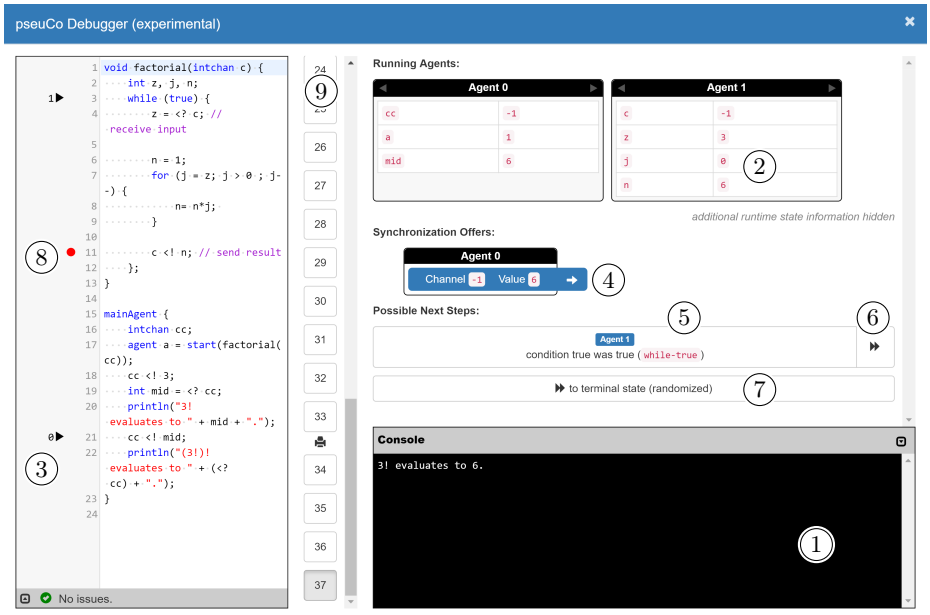


Fig. 8. PseuCo.com in a debugging session for the program from Listing 1.

6 Conclusion and Future Work

This paper has presented a translational semantics for PseuCo to Petri nets tailored to the use in education. Its intermediate step, the higher-level Petri net formalism $C^{P}PN$, is geared towards easy visualization of program semantics and reusability for other programming languages. The transpiler is highly integrated into PseuCo.com, providing easy access to it to students and teachers and extending PseuCo.com with a structured and concurrency-preserving semantics and a nondeterminism-preserving debugging facility. The transpiler and its underlying colored Petri nets implementation are available as open-source JavaScript libraries.

There is obvious room for improvement in these tools, most importantly expanding the transpiler to support the full feature set of PseuCo, which includes arrays, data structures and monitors with condition synchronization. Doing so cleanly requires the introduction of additional command transitions in $C^{P}PN$.

The Petri net extension is brand-new and has as such not been used in a lecture edition so far. This is planned for summer 2019, together with a shift in the theoretical course focus, now embracing the Petri net perspective on concurrency.

Acknowledgments. We would like to thank Bernd Finkbeiner for supervising an earlier phase of this project. Michaela Klauck provided helpful feedback during development of the debugging feature of PseuCo.com. This work was partially

supported by the ERC Advanced Investigators Grant 695614 (POWVER) and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 389792660 – TRR 248 (see https://perspicuous-computing.science).

References

1. Computer Science Curricula 2013. ACM Press and IEEE Computer Society Press (2013). https://doi.org/10.1145/2534860, https://doi.org/10.1145/2534860
2. Best, E., Fraczak, W., Hopkins, R.P., Klaudel, H., Pelz, E.: M-nets: An algebra of high-level petri nets, with an application to the semantics of concurrent programming languages. Acta Inf. **35**(10), 813–857 (1998). https://doi.org/10.1007/s002360050144, https://doi.org/10.1007/s002360050144
3. Best, E., Hopkins, R.P.: B(pn)^2 - a basic petri net programming notation. In: Bode, A., Reeve, M., Wolf, G. (eds.) PARLE '93, Parallel Architectures and Languages Europe, 5th International PARLE Conference, Munich, Germany, June 14-17, 1993, Proceedings. Lecture Notes in Computer Science, vol. 694, pp. 379–390. Springer (1993). https://doi.org/10.1007/3-540-56891-3_30, https://doi.org/10.1007/3-540-56891-3_30
4. Biewer, S., Freiberger, F., Held, P.L., Hermanns, H.: Teaching academic concurrency to amazing students. In: Aceto, L., Bacci, G., Bacci, G., Ingólfsdóttir, A., Legay, A., Mardare, R. (eds.) Models, Algorithms, Logics and Tools - Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 10460, pp. 170–195. Springer (2017). https://doi.org/10.1007/978-3-319-63121-9_9, https://doi.org/10.1007/978-3-319-63121-9_9
5. Dijkstra, E.W.: Letters to the editor: go to statement considered harmful. Commun. ACM **11**(3), 147–148 (1968). https://doi.org/10.1145/362929.362947, https://doi.org/10.1145/362929.362947
6. Ecma International: ECMAScript© 2015 Language Specification, 6th edn. (June 2015), standard ECMA-262
7. Eisentraut, C., Hermanns, H.: Teaching concurrency concepts to freshmen. Trans. Petri Nets and Other Models of Concurrency **1**, 35–53 (2008). https://doi.org/10.1007/978-3-540-89287-8_3, https://doi.org/10.1007/978-3-540-89287-8_3
8. Gansner, E.R., Koutsofios, E., North, S.C., Vo, K.: A technique for drawing directed graphs. IEEE Trans. Software Eng. **19**(3), 214–230 (1993). https://doi.org/10.1109/32.221135, https://doi.org/10.1109/32.221135
9. Garavel, H.: Nested-unit petri nets. Journal of Logical and Algebraic Methods in Programming **104**, 60 – 85 (2019). https://doi.org/https://doi.org/10.1016/j.jlamp.2018.11.005, http://www.sciencedirect.com/science/article/pii/S2352220817302018
10. The Go programming language specification. http://golang.org/ref/spec
11. Holzmann, G.: The Spin Model Checker – Primer and Reference Manual. Addison-Wesley Professional, first edn. (2003)
12. Jensen, K.: Coloured petri nets and the invariant-method. Theor. Comput. Sci. **14**, 317–336 (1981). https://doi.org/10.1016/0304-3975(81)90049-9, https://doi.org/10.1016/0304-3975(81)90049-9

13. Jensen, K.: Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use - Volume 1. EATCS Monographs on Theoretical Computer Science, Springer (1992). https://doi.org/10.1007/978-3-662-06289-0, https://doi.org/10.1007/978-3-662-06289-0
14. Kristensen, L.M., Christensen, S.: Implementing coloured petri nets using a functional programming language. Higher-Order and Symbolic Computation **17**(3), 207–243 (2004). https://doi.org/10.1023/B:LISP.0000029445.29210.ca, https://doi.org/10.1023/B:LISP.0000029445.29210.ca
15. Meyer, R., Strazny, T.: Petruchio: From dynamic networks to nets. In: Touili, T., Cook, B., Jackson, P.B. (eds.) Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6174, pp. 175–179. Springer (2010). https://doi.org/10.1007/978-3-642-14295-6_19, https://doi.org/10.1007/978-3-642-14295-6_19
16. Milner, R.: A Calculus of Communicating Systems, Lecture Notes in Computer Science, vol. 92. Springer (1980). https://doi.org/10.1007/3-540-10235-3, https://doi.org/10.1007/3-540-10235-3
17. Milner, R.: Communicating and mobile systems - the Pi-calculus. Cambridge University Press (1999)