

Service-Oriented Petri Net Model

Oleg Iakushkin(✉)

Saint-Petersburg University, 7/9 Universitetskaya nab., St. Petersburg 199034, Russia
o.yakushkin@spbu.ru

Abstract. In this paper, we consider the existing problems of colour stochastic Petri nets (CSPN) for modelling of systems of interconnected applications - services. The main issues that arise when modelling service applications and their interactions are as follows: support of composite types and indexed arrays, describing operations on top of them. It is also important to note the problem of using global time in model combinations – it is hard to describe different levels of abstraction. For example, to combine the network behaviour such as delays, duplication, and packet loss and its effects on business logic. To solve all of these problems, we propose a new model of a service-oriented Petri net (SOPN). It is built based on the stochastic Petri net (SPN) by adding some restrictions and extension components. We added tools for creating data structures, fundamental types such as integers, rational numbers, ordered arrays. The model presented in this paper provides the toolkit for creating complex atomic operations in terms of model-controlled time. Meanwhile, for all the new components and features transition to the basic SPN model is supported.

Keywords: Modelling · Petri net · Services · CSPN

1 Introduction

It is important to analyse the methods of problems solving using services in a distributed computing environment, and it is necessary to create a modelling apparatus that allows a comprehensive description of complex processes and to draw qualitative conclusions about the behaviour of the systems in question [2,7,8].

2 Problem at Hand

Service systems are characterised by characteristics that allow us to limit the space covered during the simulation. In this regard, we introduce several axioms.

Axiom 1: *In the service systems under consideration, only frequently recurring phenomena can be modelled using Bayesian networks, the modelling of the remaining events should be carried out unambiguously and decomposed into the logic of finite state machines.*

© Springer Nature Switzerland AG 2020
O. Gervasi et al. (Eds.): ICCSA 2020, LNCS 12254, pp. 51–66, 2020.
https://doi.org/10.1007/978-3-030-58817-5_5

The specifics of the systems justifies this assumption under consideration - service systems can be distinguished by a long term of continuous operation and cyclic processes of resolving the same type of user requests [4–6]. Such behaviour leads to the fact that most of the events of service systems are repeated many times by updating the probabilistic modelling of system events regardless of their primary source.

Axiom 2: *Objects that can be modified in the systems under consideration can be converted to the format of finite sets of real numbers.*

2.1 Inspecting Type Requirements

Let us inspect this assumption for fundamental programming types:

- Real number (including `int`, `float`, `double`, `long` etc.) $a = b/c$ can be represented as a set of $\{b, c\}$;
- A `string` of characters of a known size can be represented as $\{c_1, c_2, ... c_n\}$ where c_i is a character from a finite alphabet (represented as an integer)
- An `enum` is a straight set of integers by defenition
- If we know all of our application types we can create a set $\mathbb{T} = \{\tau_1, \tau_2, ...\tau_n\}$ that can be used to denote a set of pairs describing a type instance at hand - its current values v and type identifier i, $\tau = \{i, v\}$. Combinations of such pairs compose type instances and allow for complex types like `struct` and their nested versions.
- An indexed array such as `vector` or `List` can be described by pairs of integers $o_i = \{1, 2, ...N\}$ and their instance values so that array element would be a set $e = \{o_i, \tau\}$

Thus for complex types, we need integers and a way of their composition into sets.

2.2 Looking at SPN Capabilities

There are many works devoted to how Petri nets can be utilised to describe technological processes [9]. The concept of colours is often resorted to simplifying such descriptions. Colour is a special kind of label that can be unambiguously correlated with a specific value from a previously known set. Models CSPN, GCSPN retain the ability to switch from a model that uses colours to the original [1,12,13]. It is important to emphasise that the formalism of Petri nets involves the departure of labels in the transition one at a time [10,11].

In the framework of this paper, we will mostly adhere to the SPN definitions from [3,10]. Define the SPN by indicating the significant properties of this work.

Definition 1. *SPN building blocks:*

- *A finite set of places* $P = \{p_1, p_2, ..., p_n\}$*;*
- *A finite set of transitions* $T = \{t_1, t_2, ..., t_n\}$*;*

- *A subset of immediate transitions $E' \subset T$. An immediate transition fires the instant it becomes enabled, whereas a timed transition fires after a positive amount of time;*
- *A marking s is configuration $s = (s_1, s_2, ...s_n)$ of marks assigned to places $d_1, d_2, ...d_n$ respectfully. All possible markings compose a countably infinite set G;*
- *Sets $I(t), L(t), J(t) \subseteq D$ of normal input places, inhibitor input places, and output places, for each transition $t \in T$. We will call them places connected to transition t;*
- *A transitions $T(s)$ are enabled when $T(s) = \{t \in T : s_j \geq 1$ for $d_j \in I(t)$ and $s_j = 0$ for $d_j \in L(t)\}$. in other words when all connected inhibitor input places are empty and marks can be obtained from all connected normal places;*
- *The marking of an SPN changes when one or more enabled transitions fire. When a transition fires it removes at most one token from each of its input places and deposits at most one token in each of its output places;*
- *$T* \subseteq T(s)$, denote by $p(s', s, T*)$ the probability that the new marking is s' given that the marking is s and the transitions in the set $T*$ fire simultaneously;*
- *For each transition t the priority $\varrho(t)$ is defined as a finite, non-negative integer. Whenever transitions t and t' are in a conflict, meaning they can t fire simultaneously, the net behaves as if only one with highest priority fires.*

Criticism of Traditional SPNs. Two main problems arise to have capabilities of service model description:

- Creation of a transition as a function of the total number of marks, is not possible without creating special schematic solutions. Thus integers can not be used out of the box;
- If a transition is defined as timed and fires only after a certain time δ, it is hard to combine it with other transitions to gain predetermined time characteristics. Thus item grouping and combination requires additional support in terms of additional markings and places.

2.3 Task Definition

We need to create an extension set that would allow SPN users to create:

- Integers;
- Composable types;
- Operation combinations with fixed time characteristics;
- Composable groups fro pattern definition and reuse.

So we will get a modelling framework that allows users to:

- Create their type combinations;
- Create transitions that operate on such types;
- Combine network subsections into reusable patterns.

3 Our Solution - Service-Oriented Petri Net Model

First, we will suggest a set of restrictions that will simplify work with a network structure, keeping SPN compatibility. Then using graph notation of Petri networks [5]. We will create a set of extensions that solve our problem, having transparent convertibility to basic SPN. We call our solution Service-Oriented Petri Net Model (SOPN).

3.1 Core Extensions

Several main building blocks must be discussed to be capable of building complicated SOPN structures.

Probability Function for Each Transition. Let us reformulate the way we deal with state change probability $p(s', s, T*)$. Such approach generalises the way states change on a subset of T and a state s as a whole. This makes it hard to formulate the network in terms of transitions and places. One would like to define a set describing transition firing probability functions $\Gamma = \{\gamma_1, \gamma_2, ...\gamma_k\}$ for each transition in $T = \{t_1, t_2, ...t_k\}$. So that γ is by itself free from system configuration state s and is evaluated when a transition is enabled by $I(t)$ and $L(t)$. This is a restriction that reduces SPN modelling power, yet provides clarity to network definition from a practical system modelling standpoint.

Time as a Marking. Service systems are composed of indirection layers. Processes on such layers may have different time scales while doing similar operations. To solve such a problem, we make all time-related transactions dependent on network marking configuration s instead of abstract system time. We do it by adding a place-transition subsystem to the network and cause all timed transitions dependent on it.

A time counter is connected so that each transition in the system can to fire only when time transition is enabled. When time transition is enabled will be calling it a unit of time - a *tick*.

As you can see on Fig. 1 a transaction may require any integer amount of time before it will be enabled to fire from the *time* point of view.

Note that on Fig. 1 we depict actual transaction that shall be started after integer time m as a blue square.

Generalised Locking Strategy. Service-oriented programs need to allow transition compositions that can be executed in a single time step - like functions in programming composed of operation sets. Thus we had to create a locking extension that can see if a given transaction can start its execution and when it has finished it.

– A lock place with one mark thru a transition is connected to a “work” place. this transition is also connected to all “enablers” of all required transitions.

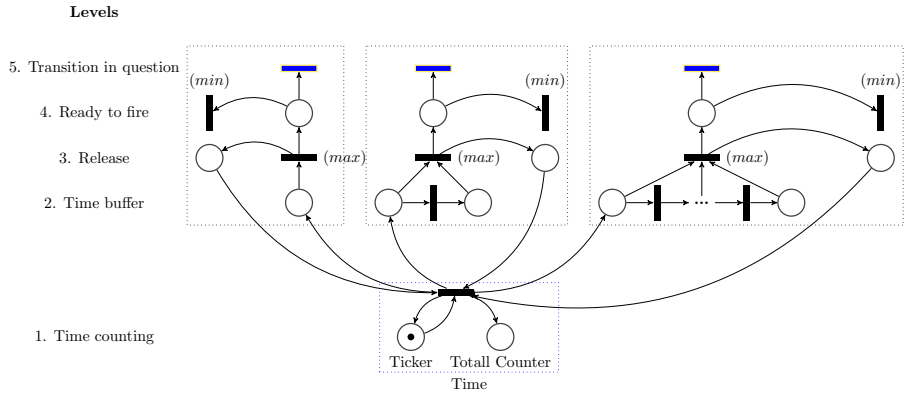


Fig. 1. A time structure embedded into state configuration

– if a transition *is not enabled* that marks are collected, and transition did not fire.
– Otherwise transition logic fires. Now we can look at transition combination as a separate network, operating as one immediate transition.
– After all required operations finished execution and are ready to return results to output places we empty "done" places equipped with one mark and release the lock.

All locks are allowed to enable "work" no more than once in a time tick.

Thus we present (see Fig. 2) a lock logic extension that enables transition combination. Note that transition composition inputs are depicted as a blue rectangular and outputs as orange ones.

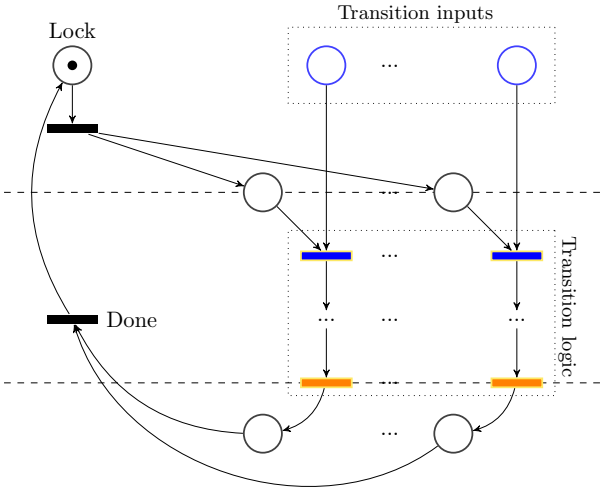


Fig. 2. A transition lock structure

Transition Composition. Generalised locking strategy enables us to create *functional transactions* that are composed of sets of operation transactions. In other words combination of a lock structure enabled for all such groups of transactions, that in essence depict functions, in combination with a global time clock allow us to define complex functions following SPN formalism. Such functions can be defined as $f(t, r, p, I) -> O$, where t is the required integer amount of time steps, r conveys function execution priority compared to others, I depicts input requirements (incoming transitions) and p show execution probability when all requirements are enabled, while O depicts output transition options. Note that any such *fuctional transition* is just an SPN graph with some nodes defined as *Inputs* and some defined as *Outputs*.

It is essential to be able to combine as many transitions into one as needed so that their combination can fire in a predictable amount of time (in one or more system ticks). So *fuctional transformations* can be combined allowing complicated behaviours.

When we can be assured that our *fuctional transition* is executed in a single *time step* we can start to reason about such transition as an SPN graph separated from its surroundings. It is important to note that all of such subgraph required inputs shall be *passed on to it* - locked or consumed so that other transitions will not be able to change them in parallel. Thus we can deal with *"all"* input place items as a form of *functional transition* requirement.

Copy and Move. A good examples of simple *functional transitions* are *"copy all inputs from one place to another"* and *"move all items from one place to another"*.



Fig. 3. *Functional transitions* to copy and move input

To replicate values between two nodes, a copy *functional transition,* that keeps all source items and clones them into output place is depicted in Fig. 3.

Its SPN implementation graph details can be found on Fig. 4. Here we depict:

- Input place is depicted by a blue circle (all of its inputs shall be frozen before this transition fires);
- Input transition by a blue rectangle which is enabled when the time, probability and general locks are enabled;
- Output place for all input place items as an orange circle;
- Output transition as an orange rectangle, so that transition would unlock the system when it has finished its execution.

Move implementation is similar Fig. 4.

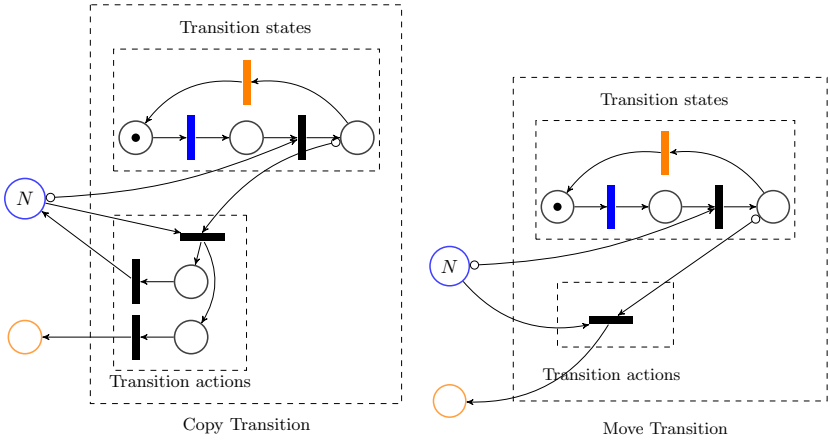


Fig. 4. Implementations of copy and move *functional transitions*

Condition Swithces. It is paramount to be capable of selecting an option depending on all items in a place. So we need a *functional transition* for a switch.

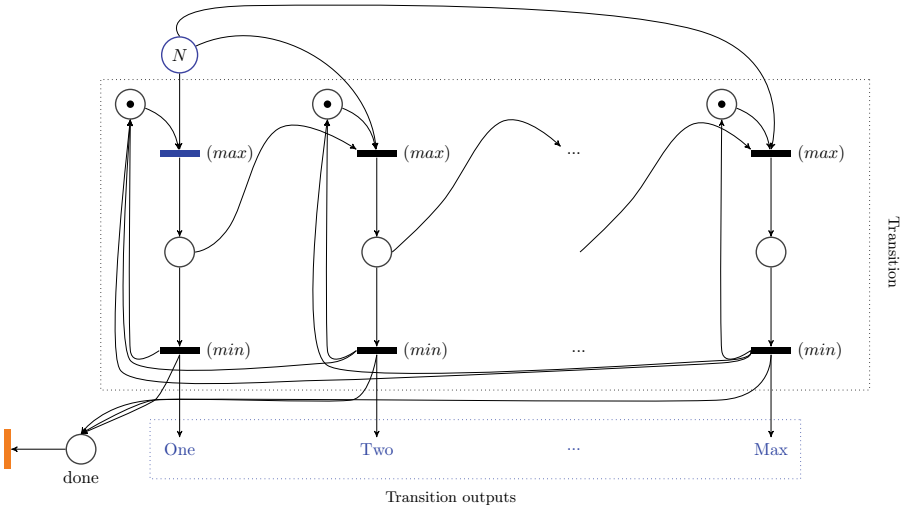


Fig. 5. A switch condition structure

In other words, it is essential to be able to switch depending on the input. An example of such a switch is presented in Fig. 5. We utilise priorities (in parenthesis near transitions) to prioritise transition activation and release the lock when a switch has selected a value. Note that this *condition switch* approach

can be used to implement all kinds of behaviours, including inhibitor transitions for empty checking.

The main disadvantage of such graph is that to convey with SPN formalism it requires *fixed maximum input size of* N. Yet as it is an extension, such capacity can be set and fixed before SOPN network execution.

Functional Transition Composition. As we can copy parts of our SPN graphs and combine them we can create functions composed from sets of operations.

An example of *functional transaction* combination possibilities is shown on Fig. 6.

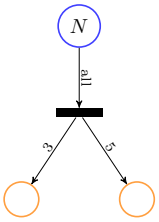


Fig. 6. A subnet operating on input and outputing a const value depending on it

Implementation is shown in detail here Fig. 7. As you can see we have a switch case leading to copy as preexisting *functional transactions*.

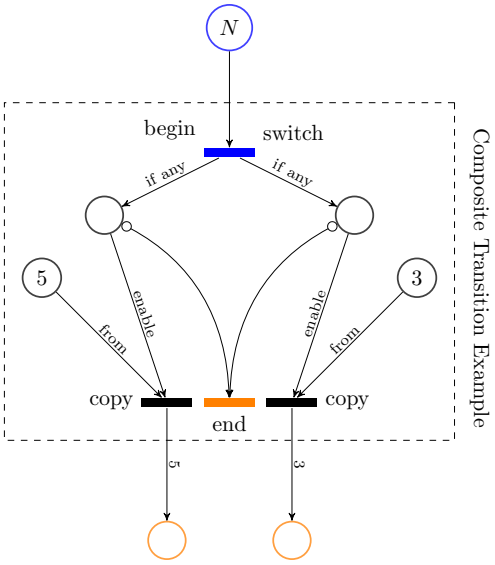


Fig. 7. A subnet operating on input and outputing a const value depending on it implemetation option

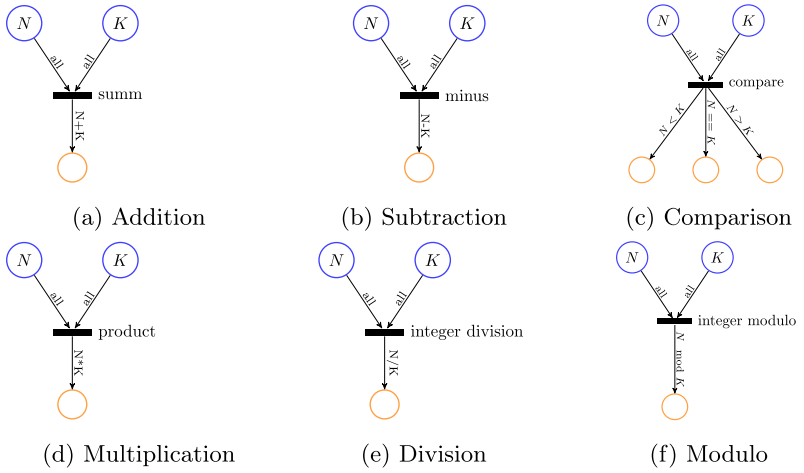
Theorem 1. *For any set F composed of functional transitions, if all of each of its outputs are reachable, all outputs of its combinations will also be reachable.*

Proof. Proof by contradiction: say not all of its outputs combinations are reachable. Then at least one of the transition nodes in a SOPN graph is newer reached. Yet we know that all of the graph building blocks, thus all nodes are reachable. This creates a contradiction. Therefore for any set F composed of *functional transitions*, if all of its outputs are reachable than all outputs of its combinations will also be reachable.

3.2 Integers

3.3 Integer Operations

On top of integers, we need to be able to perform a set of different mathematical operations. In this paper we present a basic set of integer operations such as $\{+, -, >, ==, <, *, /, \%\}$ in Fig. 8.



(a) Addition (b) Subtraction (c) Comparison

(d) Multiplication (e) Division (f) Modulo

Fig. 8. Basic math operations $+, -, >, ==, <, *, /, \%$

Addition and Subtraction. Basic mathematical operations such as addition and subtraction are depicted in Figs. 9, 11.

Compare. The comparison allows us to take two nodes with values N and K and to get any of the three results, namely $K == N$ or $K > N$ or $K < N$. As shown in Fig. 8c we deplete original places values thus sometimes copy operation may proceed as shown in implementation details here Fig. 12.

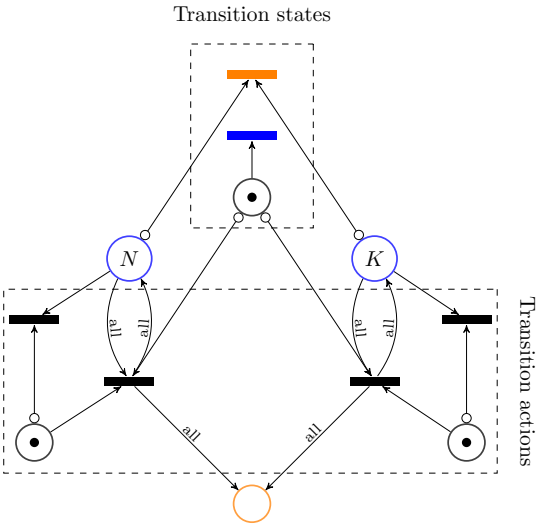


Fig. 9. An addition subnet

Multiplication, Division and Mod. Multiplication extension relies heavily on data copying and inhibitor connections as shown in Fig. 13. Division operation is relying on subtraction (see Fig. 14). Mod transition is heavily dependent on division (as shown in Fig. 14)

3.4 Nested Types

Arrays Extension. Complex types such as arrays with fixed length and operations on top of them can be implemented as a set of *functional transactions* on top of a group of places as shown on Fig. 15. Here we depict read operations, and others can be implemented similarly.

Nested Types Extension. We need a description of where the entire array, integer or some other combination of places set with *functional transitions* on top of them are located. So we present a notation for that: *type-place bonding*. Alike they do it in programs we have a separation of flow logic and memory management. SOPN user is managing a higher-level abstraction of places that are associated with type instances indirectly:

- For each place that will be handling complex types on SOPN network we have a unique integer address number - thus we call it a *holder place*;
- Each type is composed of its data places (on top of which functional transitions are defined) and a place that keeps its address, its *location Id*;
- Before simulation is executed maximum allowed type instance counts are defined;

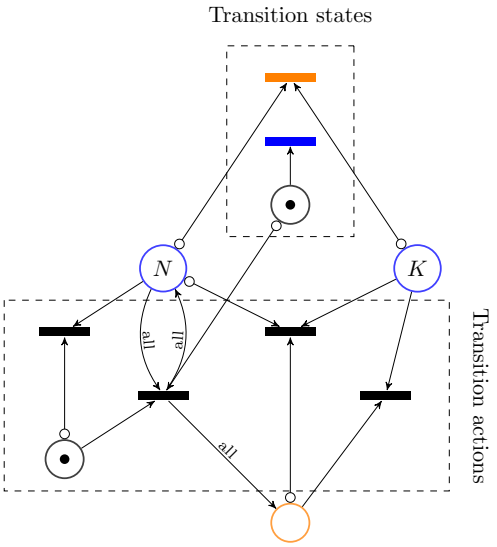


Fig. 10. A minus operation subnet

– When a transaction takes a type instance from one holder place to another, it changes instance location Id.

Thus SOPN user is working with a graph of type instances bound to places (similar to pointers). When a *functional transition* operation is activated, it changes or creates new cases not only by setting data, yet also by changing type location Id.

Structured Types. We have defined integers, arrays, timesteps and locking logic for basic colourless mark token types. Yet how to distinguish an array of *type A* from integer of *type B*? If we add colour to nodes using CSPN as a base, we can define a complex type structure like this:

```
struct data {
    int number;
    string text;
}
```

As a colorset like this: $data = \{int, \{int_1, ..., int_n\}\}$ where n is max array length, red color set would indicate integer type, blue color set would indicate a character type and green colorset would indicate fixed size character array type. Here colors will allow CSPN type conversion, separation and compatability operations, while SOPN extensions will allow *transition functions*, global time and locking operations, type-place bonding.

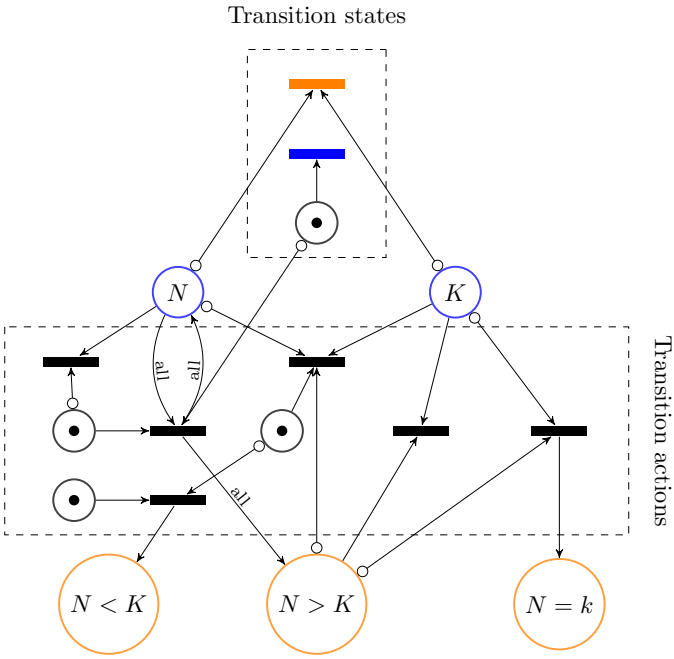


Fig. 11. A comparison transition

3.5 Modules

Portable types and operations on top of them provide a low-level abstraction for general logic description and RPC types. Yet, when it comes to a service-oriented network, it is paramount also to have interaction patterns that can describe services interactions with each other. Thus we present a *module* concept. A module is a SOPN subgraph that consists of places and transitions and can be easily replicated. It has publicly available transitions and places. Time inside of it can be scaled by all transitions time multiplication by a constant. An example of a module can be seen in Fig. 10.

Comparing Module Implementations. We will be calling a modules SOPN subgraph places connected to a module *public transitions* and that module's *public places* - *related* places. A configuration evolution distribution (CED) $O(s) -> s_o$ means that for any initial configuration s, of places *related* to a given module, we know what marks that will be placed into its *related* places. For different module implementations, the time t and probability p that any such transformation will happen and how long it will take can change. CED computation can usually be done if no external modifications will happen to the configuration of places *related* to a module, or a module locks interactions with external components while it is in operation.

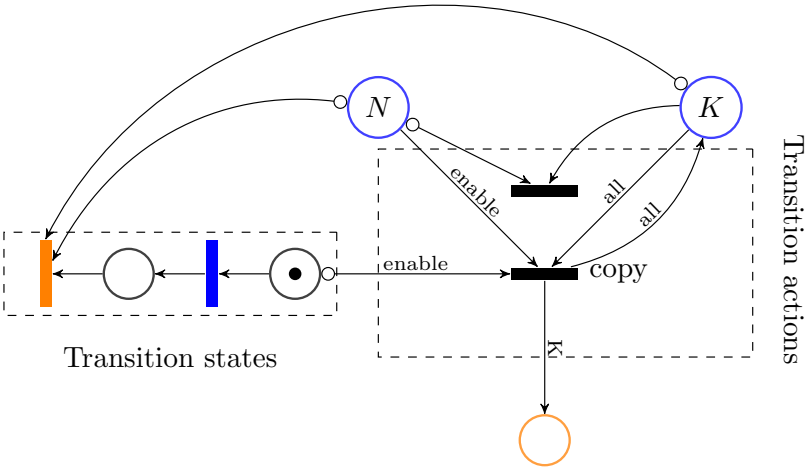


Fig. 12. An multiplication transition

Theorem 2. *If for a module its CED is known, module inner makings will not affect its behaviour while CED is not changed, and thus module inner graph can be changed in any way while it respects CED in SOPN graph.*

Proof. Proof by construction: say we know a CED of a module in SOPN graph, and another module - a SOPN subgraph with the same CED. Now if we replace one with another, its graph will change, yet as CED is not changed and thus SOPN graph will function as before concerning new module implementation timings and probabilities.

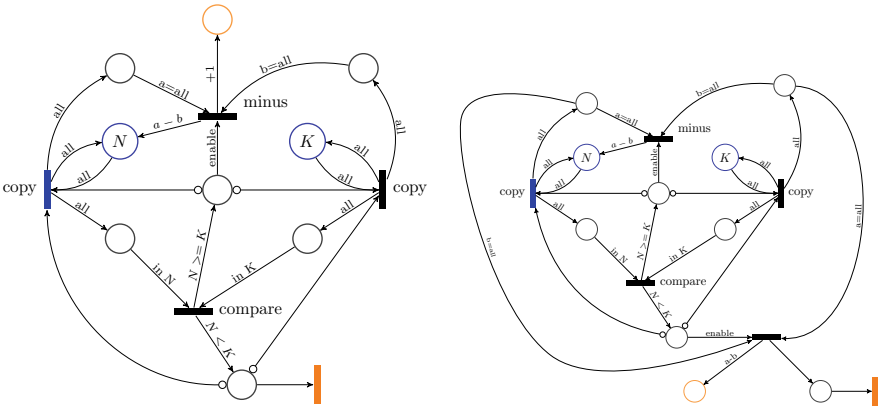


Fig. 13. A mod and division transition

Theorem 3. *A SOPN graph composed from a set G of modules with known CED will always have a CED related only to CEDs of its modules.*

Proof. Proof by construction: Now say we have a SOPN composed of modules with known CEDs. If we change any module implementation while keeping its CED for all other modules CED will not modify, only timings and probabilities will. So general SOPN graph system as a whole CED will not change. If we replace a *module* with a different CED public SOPN system, CED will change with it. Thus system CED is directly related to its modules CED.

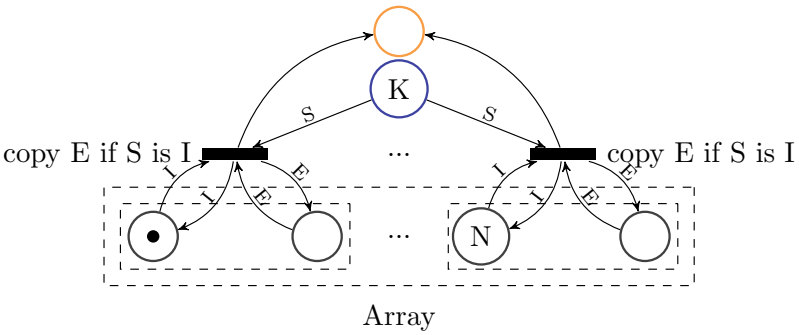
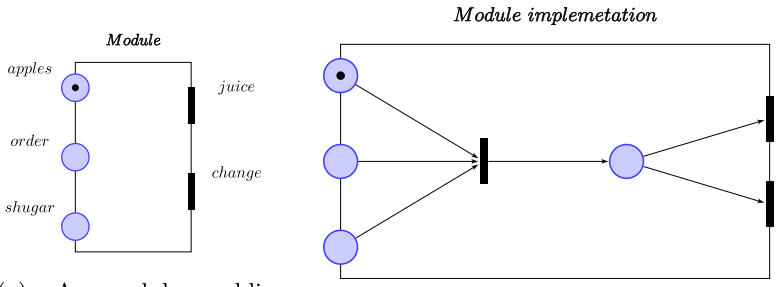


Fig. 14. Get element from array transition



(a) A module public places and transitions

(b) A module implementation

Fig. 15. A module

So We can change modules with the same outputs yet different performance characteristics and perform implementation comparisons.

4 Conclusion

For formal modelling of the interaction of service systems, the ability to describe the interaction protocols of nodes is essential. This requires support for a set of seemingly simple data types: integers, floating-point numbers, enumerations, strings. However, existing implementations of Petri nets focusing on mathematical generality operate in terms of transferring units of information-labels from positions to transitions one at a time. This principle in their formal definitions also guides existing varieties of colour Petri nets such as CSPN and GCSPN. To solve the problem of describing service interactions, we developed a new model Service-Oriented Petri Net Model based on stochastic Petri nets described in this article. The main difference in our approach is the assumption of the known maximum sizes of the data types depicted in the model. Also, we use the method to determine the time in the form of a position-label pair specified inside the system.

Acknowledgement. The authors would like to acknowledge the Reviewers for the valuable recommendations that helped in the improvement of this paper.

References

1. El Akchioui, N., Dib, F., Lefebvre, D., Ncir, N., Sebbane, S., Leclercq, E.: About the fluifidication of SPN by CPN for complex dynamical systems: critical regions. Institute of Electrical and Electronics Engineers Inc. (2020)
2. Gischer, J.: Shuffle languages, petri nets, and context-sensitive grammars. Commun. ACM **24**(9), 597–605 (1981)
3. Haas, P.J.: Stochastic Petri Nets: Modelling, Stability Simulation. Springer, London (2006). https://doi.org/10.1007/b97265
4. Iakushkin, O.: Intellectual scaling in a distributed cloud application architecture: a message classification algorithm, pp. 634–637. Institute of Electrical and Electronics Engineers Inc. (2015)
5. Iakushkin, O.: Cloud middleware combining the functionalities of message passing and scaling control, vol. 108. EDP Sciences (2016)
6. Iakushkin, O., Grishkin, V.: Unification of control in P2P communication middleware: towards complex messaging patterns, vol. 1648. American Institute of Physics Inc. (2015)
7. Iakushkin, O., Sedova, O., Valery, G.: Application control and horizontal scaling in modern cloud middleware. In: Gavrilova, M.L., Tan, C.J.K. (eds.) Transactions on Computational Science XXVII. LNCS, vol. 9570, pp. 81–96. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-50412-3_6
8. Iakushkin, O., Shichkina, Y., Sedova, O.: Petri nets for modelling of message passing middleware in cloud computing environments. In: Gervasi, O., et al. (eds.) ICCSA 2016. LNCS, vol. 9787, pp. 390–402. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42108-7_30
9. Khetkarn, K., Vatanawood, W.: Formal modeling of priority queue using generalized stochastic petri nets, pp. 39–43. Association for Computing Machinery (2019)

10. Kordon, F., Linard, A., Paviot-Adet, E.: Optimized colored nets unfolding. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 339–355. Springer, Heidelberg (2006). https://doi.org/10.1007/11888116_25
11. Liu, F., Heiner, M., Yang, M.: An efficient method for unfolding colored petri nets. In: Proceedings of the 2012 Winter Simulation Conference (WSC), pp. 1–12. IEEE (2012)
12. Ma, M., Xihou, S.J., Huang, J.G.: Study on hierarchical modeling and performance evaluation method about parallel automatic test system. J. Syst. Simul. **22**(12), 2779–2783+2787 (2010)
13. Ma, T., Pang, S., Zhang, W., Hao, S.: Virtual machine based on genetic algorithm used in time and power oriented cloud computing task scheduling. Intell. Autom. Soft Comput. **25**(3), 605–613 (2019)