Patterns in colored petri nets

PATTERNS IN COLORED PETRI NETS

N.A. Mulyar and W.M.P. van der Aalst

Department of Technology Management, Eindhoven University of Technology
P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands
{n.mulyar, w.m.p.v.d.aalst}@tm.tue.nl

April 15, 2005

ABSTRACT

Colored Petri Nets (CPN) is a graphical language, which is extensively used for modeling and analysis of distributed systems with elements of concurrency. It has proven to be a good platform for modeling of process-aware information systems, workflow analysis, design of communication protocols, embedded systems, and distributed systems, etc. The challenge that we undertake in this report is to document a set of 34 empirically gathered patterns in Colored Petri Nets in the pattern format. The main goal of the CPN pattern catalog is to serve as a source of sound solutions, proven by experience, for problems appearing during modeling by means of CPN. Furthermore, the CPN patterns can be used as a domain language for communicating problems and solutions. In order to help developers in selecting a suitable pattern, we made classification of patterns and analyzed relationships between patterns for easy navigation through the CPN pattern language.

TABLE OF CONTENTS

INTRODUCTION

Process-Aware Information (PAI) systems, i.e. systems that are used to support, control, and monitor business processes, are typically driven by models of different perspectives, i.e. process, organization, data, etc. In order to build a feasible model with help of a PAI system (e.g. WFM software) efficiently, all dimensions of requirements put on the system from process, data, resources and other perspectives, must be well understood. Developers, working in the same domain, experience similar difficulties while solving the same kind of problems. How to solve a problem, what are advantages and disadvantages of possible solutions, which solution to choose, how to realize the selected solution – these all are the questions, which every developer needs to answer. Since problems to be solved are often non-unique, i.e. they recur in many systems, developers invest their time on solving a problem and often reinvent already existing solutions.

A pattern language is one of the possible means to help developers in building their models efficiently, while avoiding reinventing already existing solutions of problems, which are common in the considered domain. Pattern languages are based on experience; they express sound solutions for problems frequently recurring in a certain domain in a pattern format. Knowing a problem underhand, a developer can look up a solution for the problem in the pattern catalog, while spending less effort on the development and ensuring the soundness of a solution.

Several steps have already been made in the direction of formalizing patterns in the workflow domain. As such, significant milestones in the form of workflow patterns [1], workflow data patterns [2], and workflow resource patterns [3] have been achieved. These pattern languages address the process, data, and resources perspectives in isolation. Note however that in a real system, the process, data, and resources perspectives interplay, thus considering every perspective in isolation is not sufficient. As far as we know, no attempts have been made to formalize the patterns combining several perspectives.

In this report, we focus on the patterns in the problem domain, where control flow and data flow interplay. We selected Colored Petri Nets (CPN) as an implementation language, since it allows modeling of data by means of colors on the top of classical Petri Nets (PN) suitable for representing the behavioral logic of the control-flow. We conducted an explorative research, based on the experts' experience, analysis of existing models and literature, which resulted in 34 implementation patterns in Colored Petri Nets.

On the one hand, the patterns described in this report are *implementation* patterns, i.e. they are mainly oriented to support model developers, working with CPN Tools, with sound solutions for problems frequently recurring during modeling. Therefore, these patterns are CPN language- specific. On the other hand, since CPN is a modeling language, which is often used for design and modeling of dynamic systems with elements of concurrency, these patterns can be also considered as *design* patterns, which grasp certain problems on the level of model design and offer solutions visualized by means of CPN. Similar to the 23 design patterns of Gamma [4], which considered as "Elements of Reusable Object-Oriented Software", CPN patterns also systematically name, motivate, and explain general design problems. However, due to major differences in concepts of object-orientation and Petri Nets, and validity in the CPN context, we will refer to the CPN patterns described in this report as implementation patterns.

The remainder of this report is organized as follows. First, we introduce some basic concepts, which we will be using in the description of patterns, and define the scope of the patterns in the context of the Petri Nets. Then we introduce the pattern format, followed by the description of the 34 identified implementation patterns. The series of presented patterns are analyzed to identify the pattern relationships, which are visualized by means of the relationship diagram. In order to reflect on pattern properties, patterns are divided into clusters and classified by several criteria. Finally, related and future work is discussed.

CENTRAL CONCEPTS AND SCOPE

In this report, we use main concepts of Petri Nets, i.e. a transition, a place, and a token. We apply concepts “event", "task", “actor” and "transition" interchangeably, as well as "token" and its mapping on an “object”. We do not refer to the definition of an object from object-oriented programming, but generalize it in such a way that by a token or “object” we can refer to any of:

- Physical objects, i.e. a chair, a stool, a table, etc;
- Conceptual objects, i.e. policies, insurances, etc;
- Information objects, i.e. anything what can be manipulated by a human or a system as a discrete entity [11].

Whenever a pattern operates with a specific type of objects, we will specify the type explicitly.

For gathering CPN patterns, we concentrate on discrete dynamic systems, which are systems with a certain state in any moment of time and a sequence of events, which bring a system from one state to another. The examples of discrete dynamic systems are distributed databases, decision support systems, e-mail systems, business systems like factories, transport companies, etc.

Discrete systems are made out of actors and objects. Actors are active components, which consume and produce objects, which are passive components [11]. Actors can be machines, humans, networks of other dynamic systems, etc.

A place is a location, where tokens reside. A place can be considered as a temporary or persistent data storage, e.g. either containing a variable or a constant number of tokens at any time.

Note that although major rules of classical Petri Nets (PN) are valid in these patterns, we concentrate on the extensions of PN by color and time. By means of colors, data can be specified, and we consider extension with time as a special color representing time[1]. Figure 1 visualizes the scope of the CPN patterns.

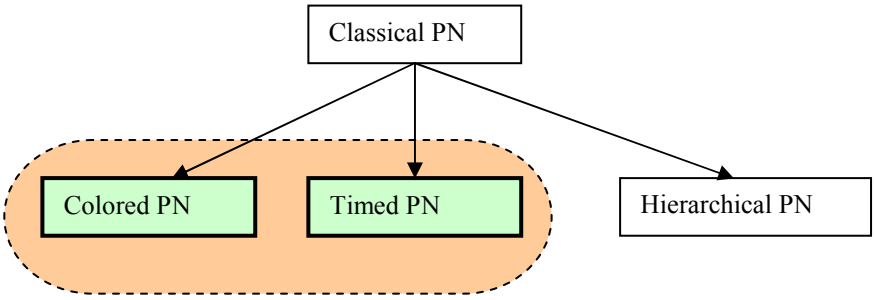


Figure 1 The scope of implementation patterns in CPN

We selected CPN Tools, as an application supporting CPN. CPN Tools is based on Petri Nets and has proven to be a good platform for modeling of process-aware systems. Using the simulation engine of CPN Tools, the design patterns presented in this report, become executable models.

[1] From theoretical point of view time is not just another color, since it is global

PATTERN FORMAT

Patterns provide an effective way to document sound solutions. Since there are multiple views on how to document the patterns and no consensus in the discussions related to selection of a single pattern format has been achieved yet, we propose the following pattern format, which in our opinion suits the best. Every CPN pattern adheres to the following pattern format:

- *Pattern name.* This is an identifier of a pattern, which captures the main idea of what the pattern does.

- *Also known as.* This section lines out the names, different from the *Pattern name*, under which this pattern might be known to the audience.

- *Intent:* This section describes in several sentences the main goal of a pattern, i.e. towards which problem it offers a solution.

- *Motivation.* This section describes the actual context of the problem addressed, and why the underlined problem needs to be solved.

- *Problem description.* This section presents the problem addressed by the pattern. For the sake of clarity, the problem is explained using a specific example. The majority of the patterns contain examples, which are also illustrated by means of CPN diagrams.

- *Solution.* This section describes possible solutions to the problem. Note that a single problem addressed by the pattern can be solved in several ways, depending on the requirements and/or context in which the pattern is to be applied. Since multiple solutions are possible, we consider every solution separately and for each of the solutions we include an implementation sub-section.

    - *Implementation of Solution.* This is a part of the solution section, which illustrates how to implement the described solution in CPN Tools. The implementation part shows not only the graphical representation of the pattern with CPN, but also describes how to integrate this solution into the example considered in the *Problem description* section. A solution may have several implementations. The presented implementations may be not the only way to implement a solution correctly. One should select an implementation depending on the context, within which the pattern is to be applied. Note that correctness of solution is not guaranteed if a tool different from CPN Tools is used for implementation purposes.

- *Applicability.* This section describes the typical situations in which the pattern can be applied.

- *Consequences.* This section outlines what are the possible advantages/ disadvantages of using the pattern. In case if the pattern supplies several solutions, this section elaborates on the differences between them.

- *Examples.* This section lists one or several examples demonstrating the use of the pattern in practice.

- *Related Patterns.* This section specifies relations of this pattern to other patterns.

PATTERN 1: ID MATCHING

ALSO KNOWN AS: REFERENCE

INTENT: to make identical information objects distinguishable

MOTIVATION:
In CPN a token can serve as a representation of an information object (i.e. a process instance, a case, an item, etc). One place may contain multiple objects of the same type. In some cases, it is necessary to compare an original value of an object with the value of the same object after it has been modified. Because of modification, the values of objects change and an object may loose its identity. Consequently, it becomes impossible to distinguish which of the modified values corresponds to which of the original objects values.

PROBLEM DESCRIPTION:
Nets in Figure 2 and Figure 3 illustrate the problem of object matching. Initially two objects *obj* of the same type *T* are present in place *Start*. These objects serve as an input for two functions *f1* and *f2*, which replace the values of processed objects with a randomly generated integer number. After applying the functions, newly produced by functions *f1* and *f2* values need to be matched in *Match pair* transition for every specific object.

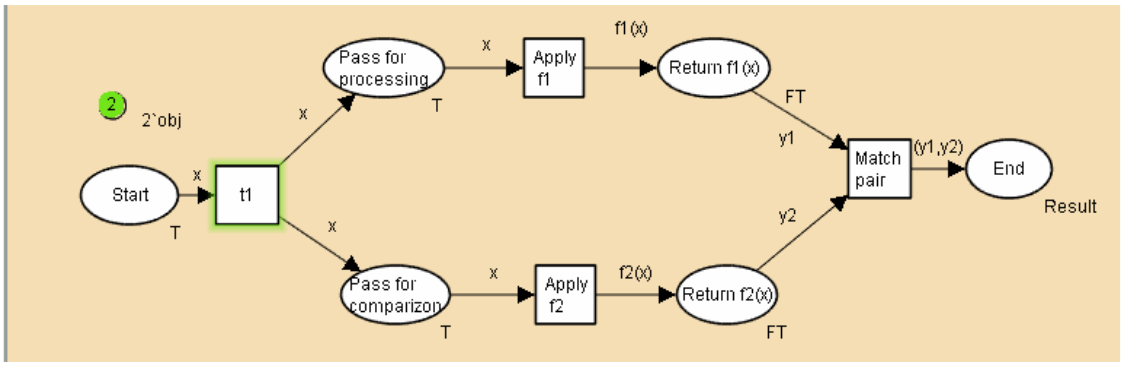


Figure 2

Figure 2 illustrates accumulation of tokens in places *Return f1(x)* and *Return f2(x)* after function *f1* and function *f2* were applied respectively. At the moment of modification, objects lost their identity, i.e. it became impossible to distinguish which of the values present in place *Return f1(x)* corresponds to which of the values in place *Return f2(x)*.

Since CPN may consume tokens from a place in non-deterministic order, the value, produced by function *f1* for one object, can be matched with a value produced by function *f2* for another object. Such behavior is undesirable and may lead to inconsistency of results and incorrectness of the matching operation whatsoever.

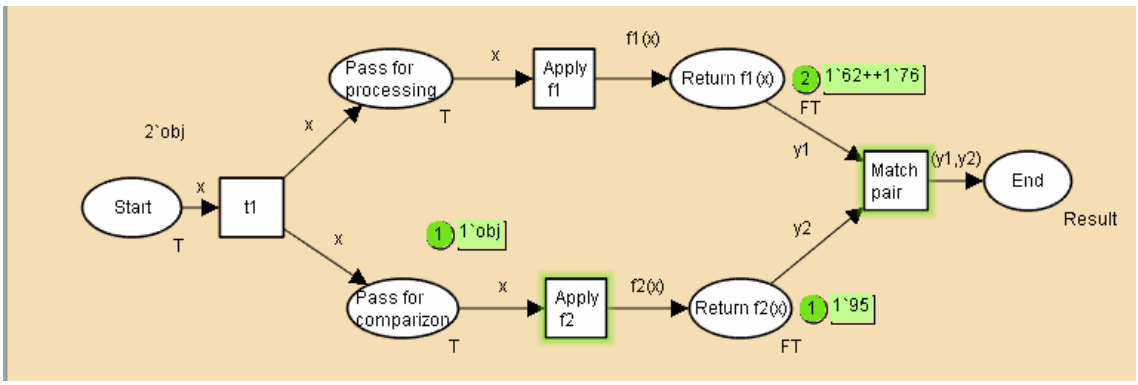


Figure 3

SOLUTION:
In order to solve the problem of referencing to originally non-distinguishable identical objects, couple every object with an identifier. The availability of identifiers makes it possible to distinguish objects of the same type.

Implementation of Solution:
The list of instructions below describes how to implement the ID MATCHING pattern:

- Replace the type *T* associated with a type of objects you want to distinguish by a multi-set *TxID*, where *ID* – is an arbitrary type selected to serve as an identification of an object (for example, *INT*, *STRING*, etc.), in all uses, where identification of the objects is relevant. For instance, in case of integers, the identifiers can be represented as *1, 2, 3, etc*. One could also use a complex/composite data type to create an ID, for instance, of the following type *(label, 1), (label, 2), etc*.
- Correlate each of the identical objects with a unique identifier (see the pattern ID MANAGER to address the problem of the ID uniqueness). For this, replace inscriptions on the arcs, containing a variable representing an object, with a correspondent pair (*variable_of_object_type, variable_of_ID_type*). For instance, replace a variable *x* of type *T* by a pair (*x,id*), where *id* is of type *ID*.
- Whenever you need to match either an initial value of an object with a modified value of the same object, or several values produced by multiple functions for the same object, pass in addition to the value of an object an object identifier *id.* Usage of the same identifier as a reference to the object will solve the problem of reference to the objects and prevent them from loosing their identity after object modification.
- In order to match values corresponding to the same object, introduce transition *Make pair,* which will be enabled only for the objects, identifiers associated to which match. Note that if different variables, for instance *id1* and *id2,* are used for representing identifiers, it is possible to match them using a guard of transition performing the matching, i.e. *[id1=id2].*

Figure 4 and Figure 5 show how to incorporate identifiers into the example presented in the *Problem description* section. Note that in the initial marking, place Start contains two identical objects *obj* that are coupled to integer identifiers *1* and *2*. Together with identifiers, objects form distinct pairs. Even if the value of an object changes, it will be possible to refer to the object by means of an identifier associated to it. In Figure 4 two objects with identifiers *1* and *2* and corresponding values *74* and *52* are present in place *Return f1(x),* while place *Return f2(x)* contains a token with an identifier *1* and value *77*.

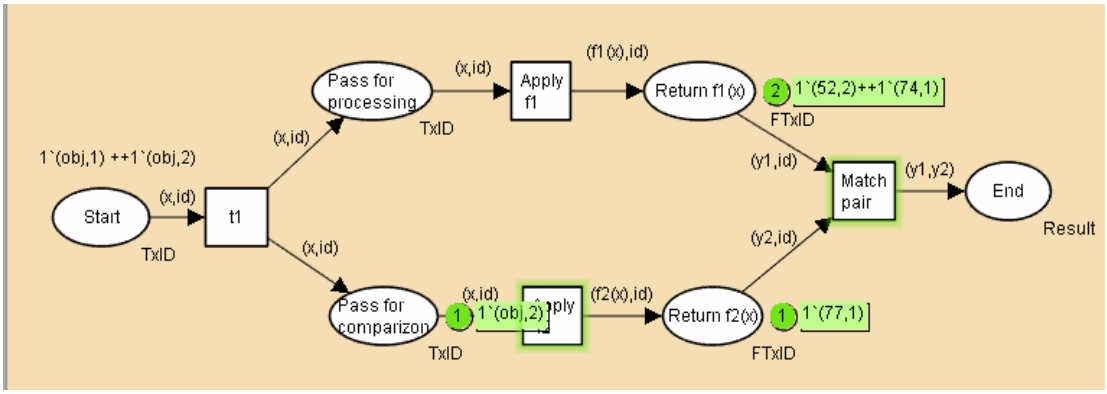


Figure 4

As Figure 5 illustrates, after firing transition *Match pair,* objects identifiers of which match, i.e. with *id=1*, were consumed. Thus, the use of identifiers preserves an object from loosing its identity.

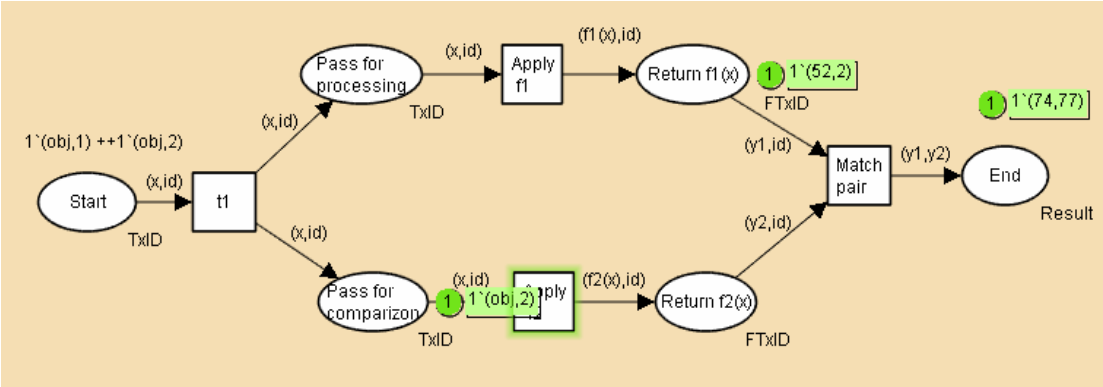


Figure 5

APPLICABILITY:
Apply this pattern to

- Refer explicitly to a specific object from a group of objects of the same type.
- Distinguish identical objects between each other.
- Structure and organize objects, based on referencing to identifiers rather then actual object values.

CONSEQUENCES:
The ID MATCHING pattern helps in solving the problem of referencing to non-distinguishable objects by means of object identifiers. However, this pattern does not guarantee that identifiers used for referencing to objects are unique. The problem of ID uniqueness is handled by the ID MANAGER pattern. In order to ensure that identifiers, used for referencing a specific object, are unique, combine the ID MATCHING pattern with the ID MANAGER pattern.

EXAMPLES:

- Several processes share the variables of the same type. In order to refer to the variables, the processes use names of the variables as identifiers.
- A teacher gives lectures to the group of students. By the end of a course, students must be evaluated: a student writes a paper, which is later discussed with the teacher. When a student arrives for the discussion, the docent selects the student's paper, using a name of the student as an identifier.

RELATED PATTERNS: this pattern can be combined with the ID MANAGER pattern to ensure uniqueness of identifiers used for distinguishing identical objects.

PATTERN 2: ID MANAGER

ALSO KNOWN AS: ID GENERATOR

INTENT: to ensure uniqueness of identifiers, used for distinguishing identical objects

MOTIVATION:

The ID MATCHING pattern solves a problem of distinguishing identical objects by assigning an identifier to each of the objects, which need to be distinguished. However, it does not guarantee that used identifiers are unique. Since CPN allows usage of multi-sets, where the same object, together with an identifier associated with it, can be created multiple times, the identical objects may be coupled to the same identifier, and thus become non-distinguishable. This may lead to confusion, incorrectness of object matching operation and diminishing the added value of the notion of identity whatsoever.

PROBLEM DESCRIPTION:

Figure 6 illustrates the problem of the ID uniqueness using an example from the implementation section of the ID MATCHING pattern. Place Start contains two objects *a* and *b* of the same type *T,* and each of them is coupled to an identifier *1*. From the point of view of correctness of the specification used in the considered net, no problem can be detected. However, in the current marking both objects are associated with the same identifier. If to refer to these objects only by means of their identifiers, these objects cannot be distinguished due to non-uniqueness of the identifiers.

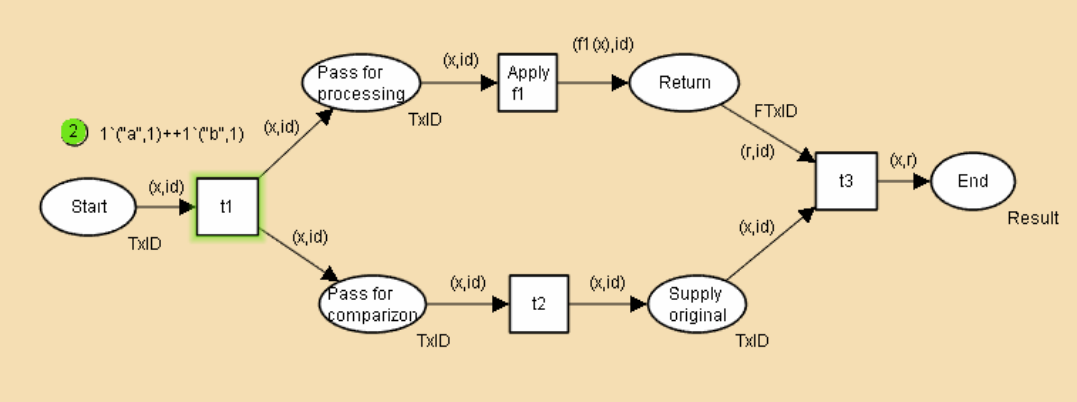


Figure 6

SOLUTION:

In order to ensure the uniqueness of identifiers, used for distinguishing identical objects, use an *ID manager*. The ID manager ensures that only unique identifiers, i.e. one of its kind, are generated. Optionally the ID manager may contain functionality for verification and control of consistency of allocated identifiers.

The mechanism of the ID management covers several aspects, i.e. ID generation, verification, and control of consistency. In general, it is sufficient to have the mechanism of ID generation implemented. The operations of verification and control of consistency are optional.

The *ID generation operation* is responsible for generation of fresh ID's. The *ID approval operation* is responsible for examining if the fresh ID, supplied by the ID generator, has not already been allocated. Finally, the *control of consistency operation* is responsible for storing all generated ID's, and keeping this storage up-to-date, i.e. inserting new ID's and removing returned ones.

Implementation of Solution:

- *The ID generation operation*

Figure 7 illustrates the mechanism of ID's generation. Unique identifiers, produced by transition *Generate ID*, are stored in place *Fresh ID*. Place *Last ID* keeps track of generated identifiers, by storing the last produced one. Initially, it stores a token with an arbitrary integer value, which will be incremented each time transition *Generate ID* fires. By memorizing the last identifier, and its incrementing, the uniqueness of identifiers is achieved. Note that in the diagram in Figure 7 identifiers of integer type are used, however any data type can be used instead.

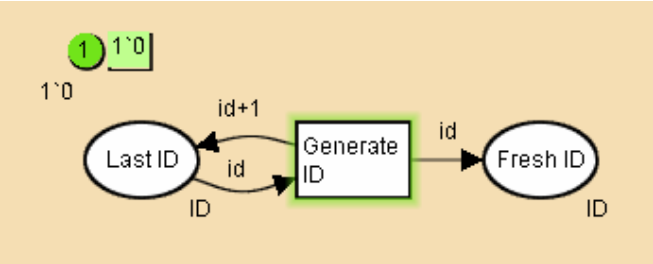


Figure 7 ID generator

- *The ID verification*

Figure 8 illustrates the mechanism of verifying the uniqueness of identifiers, supplied by the ID generator. In order to provide identifiers to place *Approved ID's*, which were not yet allocated, Solution 2 of the FILTER pattern is applied.

Place *Fresh ID* supplies a selected *id* to transition *Approve* ID, which checks if this *id* has not been already allocated. Transition *Approve ID* takes as an input a new *id* supplied for checking, and a list of the allocated identifiers *lid* stored in place *Existing ID's*. This transition checks if the *id* is an element of the list of the existing identifiers. This is done by means of the transition guard, containing the function *elt():*

fun elt(x,[])=false | elt(x,y::z) = if x=y then true else elt(x,z);

The *id* is approved as unique, if it is not an element of the list *lid*. In this case the *id* is also added to the list of ID's *lid* in order to keep the list up-to-date, and at the same time the *id* becomes available at the place *Approved ID's*. From this moment, the identifier *id* is available for coupling with an object.

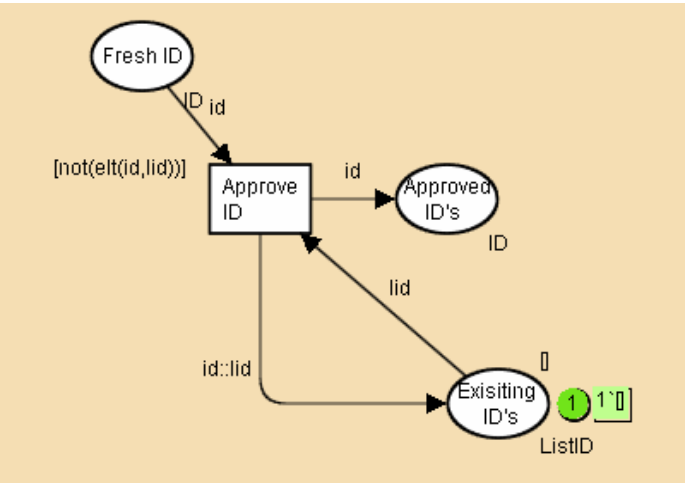


Figure 8 ID verification

- *The ID consistency control*

The mechanism of consistency control consists of two operations: inserting new ID's and removing ID's, which returned, for instance, because of the object destruction, to/from the list of existing identifiers. The insert operation was presented in Figure 8. Figure 9 illustrates the operation of deleting the returned identifier from the list of the allocated identifiers.

Transition *Destroy ID* takes the current list of used identifiers *lid* and removes returned identifier *id* using the following function:

fun del(x,y::z) = if x=y then z else y::(del(x,z));

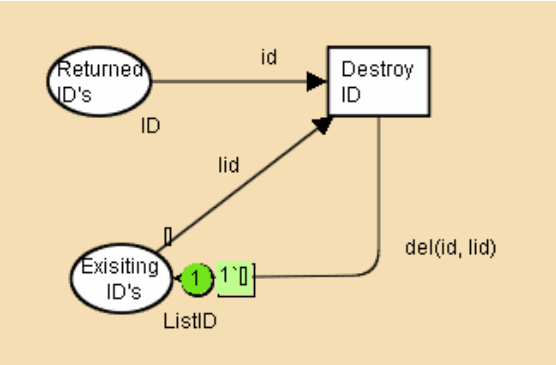


Figure 9 ID deletion

The consistency manager, combining both operations of inserting, deleting, and verification of identifiers is presented in Figure 10. Note that allocated identifiers are aggregated into the list, for which the AGGREGATE OBJECTS pattern has been applied. In addition, one can apply the QUEUE pattern to keep identifiers in the list in the strictly specified order.

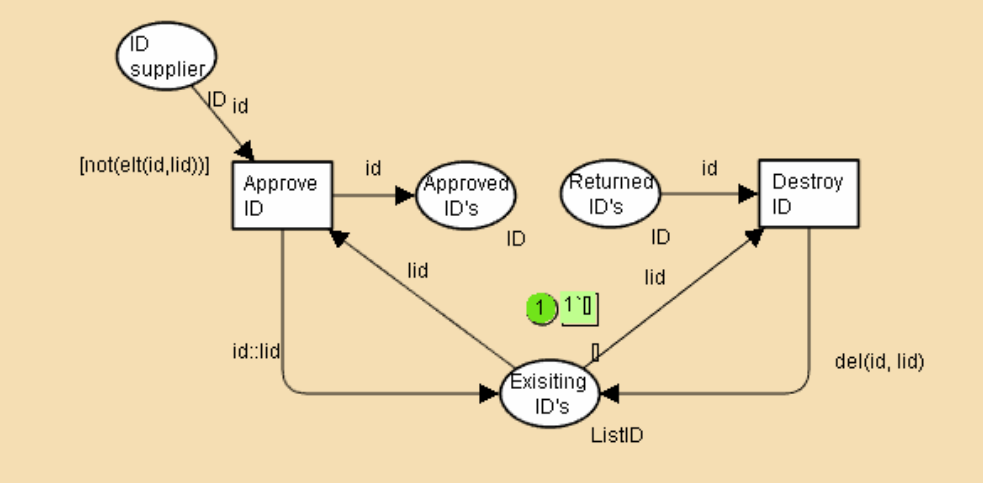


Figure 10 ID consistency control

The ID manager, combining all above-discussed functionality is presented in Figure 11. Note that in generic case, the ID manager performs only generation of identifiers, and the rest of functionality is optional and can be used as a supplement for identifiers' reallocation purposes.

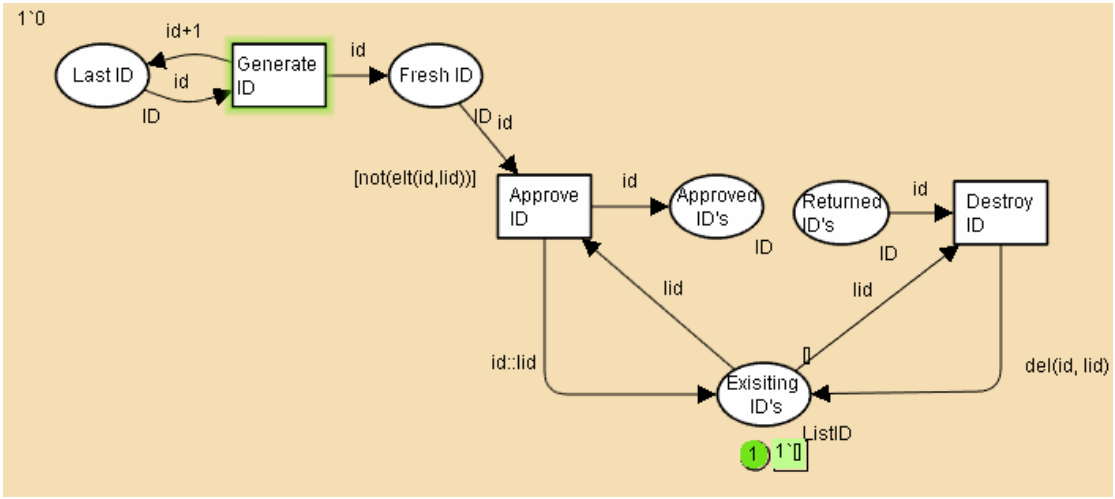


Figure 11 ID manager

Figure 12 shows how to incorporate the ID manager pattern into the example described in the *Problem description* section. Note that only functionality of generator of the ID manager is used, because (re-) allocation of identifiers is not required in this example.

In place *Start* objects *a* and *b* are originally stored without identifiers. Transition *t1* takes as an input an object from place *Start* and couples it to an identifier *id* provided from *Fresh ID* place.

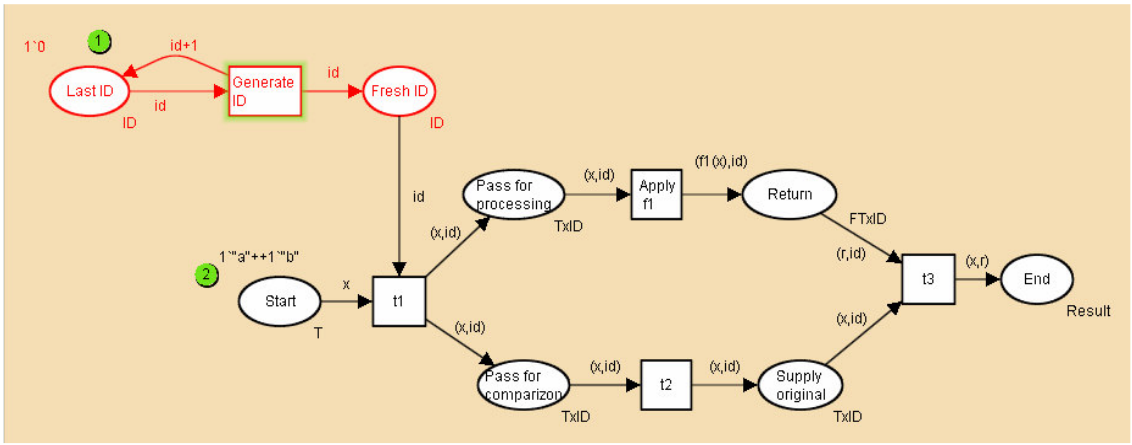


Figure 12

APPLICABILITY:
Apply this pattern in combination with the ID MATCHING pattern to

- Guarantee the uniqueness of identifiers used for distinguishing identical objects.
- Keep track of allocated identifiers.
- Ensure the consistency of allocated identifiers and allow reallocating the identifiers, which were returned and are not used in a model any more.

CONSEQUENCES:
The ID MANAGER pattern ensures uniqueness of identifiers used for referencing objects of the same type, when combined with the ID MATCHING pattern. In addition, this pattern solves the problem of data consistency, which might appear by inserting new or removing old identifiers.

Note that in its solution, this pattern uses BLOCKING STATE-DEPENDENT FILTER pattern in combination with the AGGREGATE OBJECTS pattern.

EXAMPLES:

- A tax office handles requests of visitors. When visitors arrive to the tax office, they receive a ticket with a number, which specifies the place of a visitor in the queue. A

ticket number must be unique in order to avoid several visitors being handled by the same tax officer concurrently.

RELATED PATTERNS: this pattern uses the BLOCKING STATE-DEPENDENT FILTER pattern and the AGGREGATE OBJECTS pattern in its solution.

PATTERN 3: AGGREGATE OBJECTS
ALSO KNOWN AS:

INTENT: to allow manipulation of a set of information objects as a single entity

MOTIVATION:
In many cases, it is natural to represent an information object (e.g., an order, a car, a message) as a single entity, i.e. there is a one-to-one correspondence between objects in a "real system" and tokens in the model. However, sometimes it is necessary to aggregate objects into one token, thus referring to the collection of objects as a single entity.

PROBLEM DESCRIPTION:
Figure 13 illustrates the problem addressed by this pattern. In the original model, place *object* is of type *T* and transitions *put* and *get* add and remove tokens from this place. Note that each token corresponds to an object.

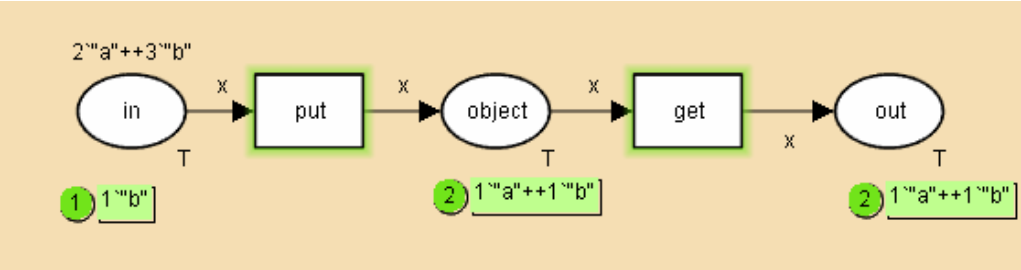


Figure 13

Suppose that it is necessary to perform an operation from the following list:

- Count the number of objects in place *object;*
- Select an object from place *object* with some property relative to the other objects (e.g., the first, the last, the smallest, the largest, the cheapest, etc.);
- Modify all objects in a single action (e.g., increase the price by 10 percent);
- (Re-) move all objects in one batch (e.g., remove a set of outdated files, items, etc. at once, rather then one by one).

None of these operations is possible in the diagram shown above. Note that it is only possible to inspect one token at a time and this is a non-deterministic choice. Moreover, this choice can be limited by transition guards and arc inscriptions, but it is memory-less and not relative to the other tokens in the place. This makes it very difficult or even impossible to realize the mentioned aspects.

SOLUTION:
In order to allow manipulation of a set of information objects as a single entity, aggregate the objects into a single token of "collection type".

Implementation of Solution:
The list of instructions below describes how to implement the AGGREGATE OBJECTS pattern (see Figure 14).

- Modify the type *T* of place *objects,* where multiple objects locate, to the collection type *LT*. In this example the collection type *list* is chosen: "*color LT = list T;*"
- Replace arcs between transitions *put* and *get* and place *objects* by bi-directional arcs with the following inscriptions. An arc which supplies an object to the collection has an inscription *x::l*, which adds an object *x* of type *T* to the list *l*. Return the current list *l* back to transition *put*. Similar, in order to *get* an object from the collection use *x::l*, and return the changed list. The described behavior represents LIFO (last-in-first-out) ordering.

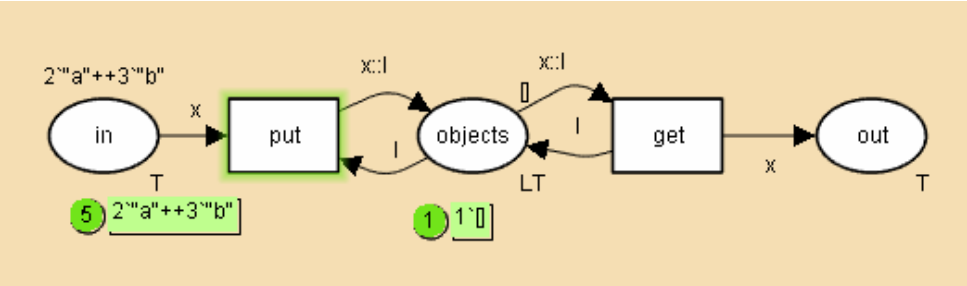


Figure 14 Aggregating objects into a list

By introducing a collection type, it becomes possible to refer to the collection of objects as to a single entity and perform operations on multiple objects contained in the collection at once. Several examples in Figure 15 and Figure 16 show how to implement some operations, from the ones mentioned in the *Problem description* section, by extending the net presented in Figure 14.

Figure 15 shows how to calculate the size of the collection, i.e. number of objects the collection contains. Note that there is always precisely one token in place *objects* representing all objects.

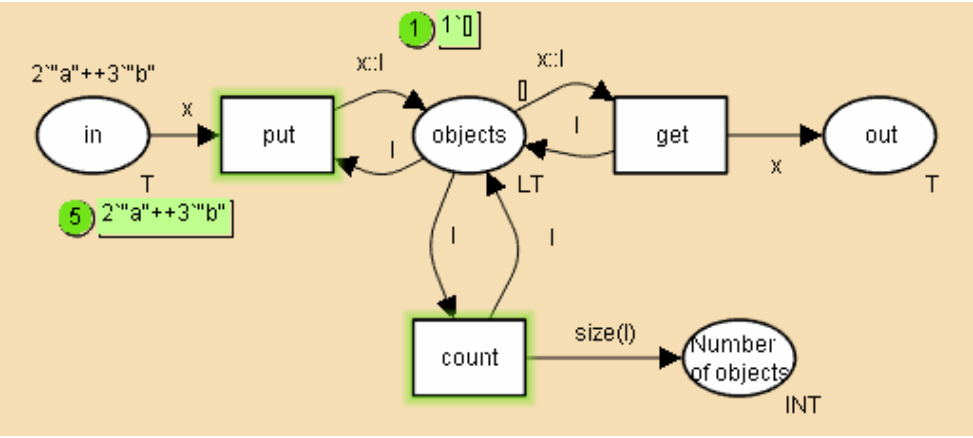


Figure 15 Defining the size of the collection

Transition *count* takes the current list of objects and sends the size of the list to place *Number of objects*. Note that a function *size(l)* for determining the size of the collection is predefined and available in the CPN Tools.

It is also possible to select an object from place *objects* with some property relative to the other objects. For example, the object represented with the first name (i.e., lexicographical order) can be obtained by transition *select* as follows in Figure 16.

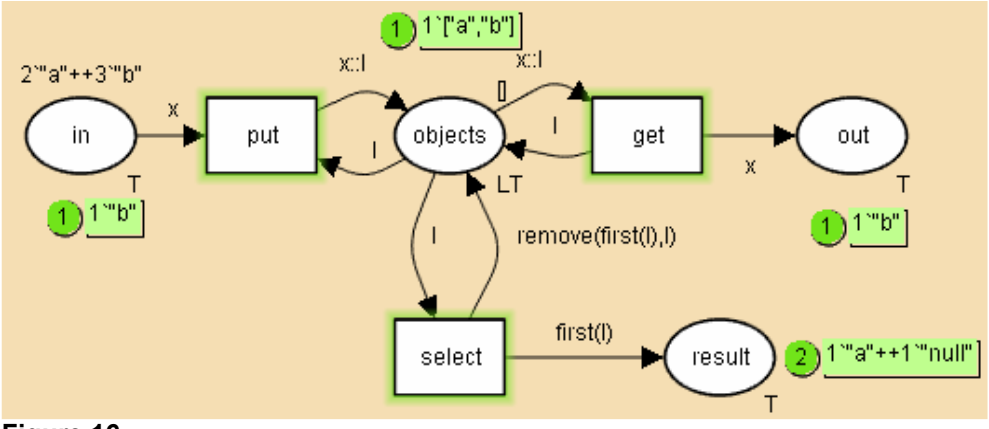


Figure 16

Function *first* selects the right object while function *remove* is used to remove the object.

```
fun first(x::l : LT) = f(x,l) | first([]) = "null";

fun remove(x,[])=[] | remove(x,y::l) = if x=y then l else y::remove(x,l);
```

In a similar way, it is possible to modify all objects in a single action (for instance, increase the price by 10 percent) and to remove all tokens (simply by returning a token with a value *[]*).

APPLICABILITY:
Apply this pattern to

- Organize multiple objects into a collection.
- Perform an operation on a group of objects or the whole collection at once.

CONSEQUENCES:
In principle, this pattern is *not* concerned with the order in which tokens are taken from the collection. The example used in the implementation section uses last-in-first-out ordering (see LIFO QUEUE). Nevertheless, if the problem of ordering is relevant, one should apply an extension of this pattern by the QUEUE pattern, or one of its specializations.

Note that although some of the functions to manipulate the collection of objects are already predefined in CPN Tools, applying special kinds of operations requires writing corresponding function(s) from scratch.

EXAMPLES:

- The salary administration of a university divided employees in different groups: students, PhD students, professors. All PhD students got a salary increase of 10%. The salary administration does not need to adjust the salary slips for every PhD student individually, but does it at one-step by increasing the salary of the whole group.
- A set of the documents is stored in a file for organization purposes. In such a way it is easy to take a whole file or select a group of the documents and send them for processing elsewhere, keeping the documents structured and centralized.

RELATED PATTERNS: this pattern is extended by the QUEUE pattern.

PATTERN 4: QUEUE
ALSO KNOWN AS:

INTENT: to allow manipulation of the queued objects in a strictly specified order

MOTIVATION:
In many systems, there are buffers where a variable number of objects are needed to queue in-between two steps in the process. This pattern assumes an unbounded queue. The queued objects need to be placed in the queue and retrieved according to different queuing policies. By exchanging one policy with another, it is possible to obtain the desired ordering of the buffered objects.

PROBLEM DESCRIPTION:
Assume that a collection of objects in form of a queue is given, and that it is necessary either to add an object to the collection or select an object from the collection and remove it from the queue. The order in which the objects are being added/removed to/from the collection may depend on the properties of an object (e.g., age, weight, etc.), the location in the queue (FIFO, LIFO), or the timestamp.

SOLUTION 1:
In order to enforce elements of a queue to move in the strictly specified order, such that several queued elements can be moved in one go, extend the solution of the AGGREGATE OBJECTS pattern. The objects are stored in a list, and added and removed from the list based on the predefined ordering algorithm.

Implementation of Solution 1:
If objects to be queued are of type T, then the type of place *queue* is *LT*. This is a list type "*color LT = list T;*". In this example, objects are sorted in the order of arrival (see the FIFO QUEUE).

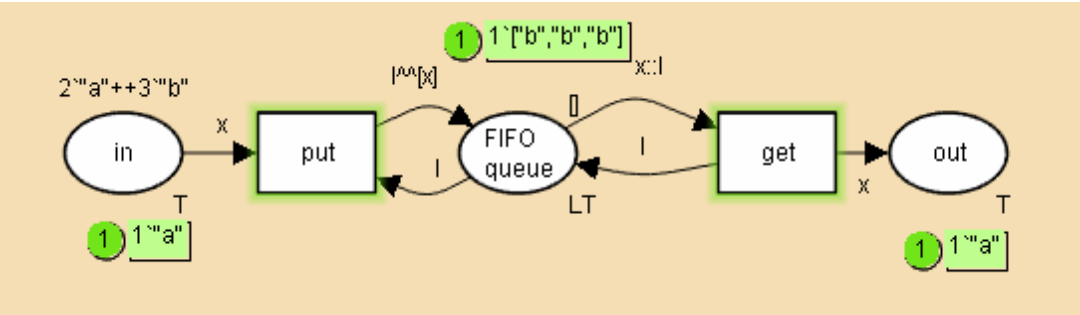


Figure 17

Transition *put* adds an objects *x* to the end of the list *l* in with help of the concatenation function: *l^^[x].* After the object *x* is added, the updated list is returned as an input to the transition *put*. Transition *put* removes the first element of the list *x::l* and puts an updated list back to the *queue* place.

By changing the arc inscriptions, it is easy to obtain last-in-first-out policy, as it is shown in the implementation of solution of the AGGREGATE OBJECTS pattern and the LIFO QUEUE pattern respectively.

SOLUTION 2:
In order to enforce elements of a queue to move in the strictly specified order, such that every queued element is distinguished separately and represented by a separate token, augment tokens with two numbers (i.e. dynamic queue bounds), which keep track of objects manipulation on the basis of predefined ordering algorithm.

Implementation of Solution 2:
For implementing Solution 2, use two variables pointing at the queue bounds, i.e. the relative to the moment of insertion in the queue position of the first and the last elements. When adding a new element to the queue, increment the upper bound; when removing an element from the queue increment a lower bound. In this way, by using a lower/upper bound one can refer to the first/last element placed in the queue.

In Figure 18 every object *x* is coupled to a number. Variables *a* and *b* keep the track of elements in the queue, i.e. variable *a* stores the position of the first element stored in the queue, while variable *b* points to the last added element. As soon as a new element added to the queue, the value of *b* increases by 1, thus moving the pointer to the last element. When an element should be removed from the queue, the element with position *a* is supplied (the first element of the queue) to transition *get.* After the object is taken, variable *a* is incremented to point to the beginning of the queue.

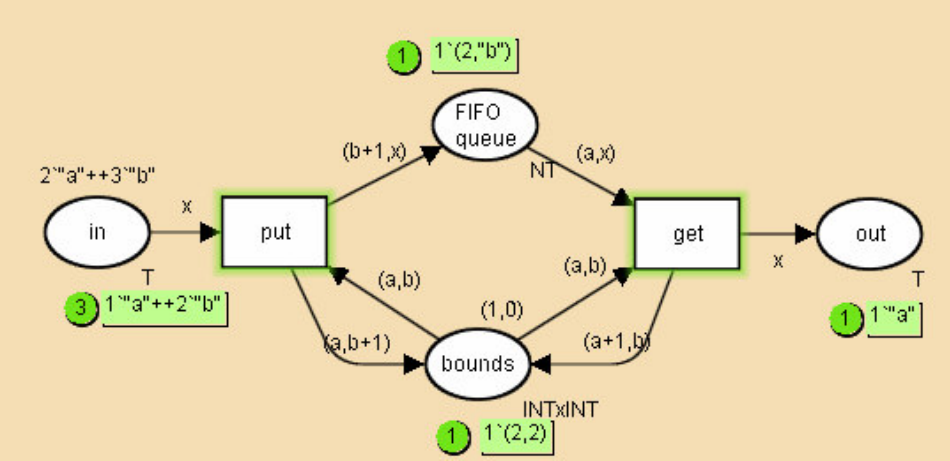


Figure 18

SOLUTION 3:
In order to enforce elements of a queue to move in the strictly specified order, such that every queued element, which consists of an objects and the object identifier, is distinguished separately and represented by a separate token, introduce a sorted collection of object identifiers, and apply the ID MATCHING pattern to define a queue element to be taken first.

Implementation of Solution 3:
For implementing Solution 3, create a place *Identifiers queue*, where collection of object identifiers will be stored. In order to maintain ordering of objects by means of identifiers, sort identifiers according to a desired policy, i.e. FIFO, LIFO, random, etc. When adding a new element in the queue *Queued elements*, place the identifier of an object to the list of identifiers. Note that in this example, the identifiers are added to the tail of the list, in order to achieve the first-in-first-out behavior.

To retrieve elements from a queue, take an identifier from the top of the list, and by applying ID MATCHING pattern retrieve the element with the same identifier from the queue.

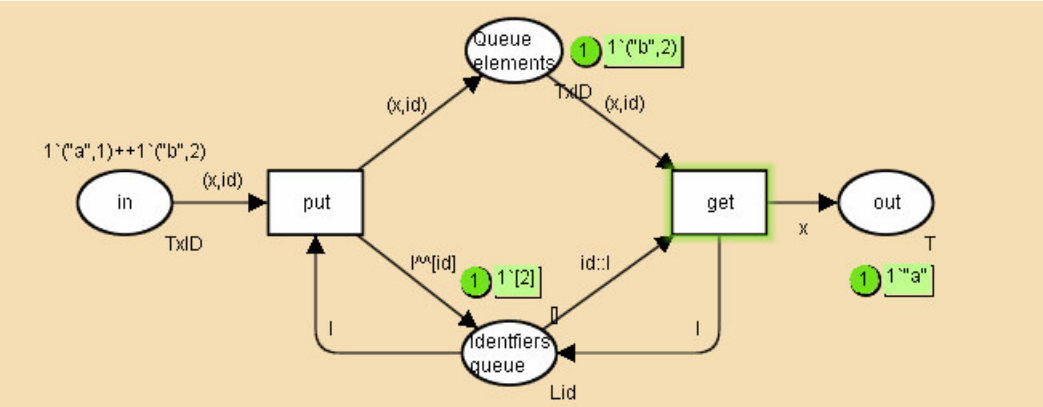


Figure 19

APPLICABILITY:
Apply this pattern to

- Ensure strict ordering of objects, while inserting or removing them from the collection.

CONSEQUENCES:
The QUEUE pattern presents three solutions, each of which can be applied for managing the order of objects, in which they were inserted/removed to/from a certain place. Although the first solution is more flexible, it hides the behavior inside the functions. In its turn, the second solution provides less flexibility and has more complex diagram. However, it captures the behavioral logic in the diagram structure itself and does not encapsulate the ordering functionality into functions. The third solution captures the flexibility of the first solution and behavioral logic of the second solution.

In contrast to the second solution, which allows pointing only on the first and the last elements of the queued objects, the third solution provides more flexibility in defining the order to withdrawing of elements from the queue and similar to the first solution can be realized by first-in-first-out, last-in-last-out, or other scheduling policy.

The selection of the solution depends on the context, within which the pattern needs to be applied, and whether it is necessary to distinguish objects as separate tokens (second and third solution) or being able performing operations on several objects simultaneously (first solution).

This pattern is a generic representation of the queue management; the specializations of this pattern, addressing a specific ordering policy, are described in the FIFO QUEUE, LIFO QUEUE, and RANDOM QUEUE correspondingly.

In this pattern, the ordering algorithm is fixed either by functions or by the net structure, so that all elements of a queue are treated in a uniform way. In some situations, there is a need to make ordering of objects more flexible and responsive on certain object properties, i.e. age, weight, timestamp, etc. The mentioned problem is addressed by the PRIORITY QUEUE pattern, which is a special type of QUEUE.

EXAMPLES:

- The city hall handles requests of citizens. In order to keep the fair waiting time, visitors that arrived first are served first.
- In order to rent a house, people register in the housing agency. Based on the date of an application, the (FIFO) order in the waiting queue is defined.

RELATED PATTERNS: this pattern uses the AGGREGATE OBJECTS pattern in Solution 1 and ID MATCHING pattern in Solution 2. The PRIORITY QUEUE, FIFO QUEUE, LIFO QUEUE, RANDOM QUEUE patterns are specializations of this pattern.

PATTERN 5: FIFO QUEUE

ALSO KNOWN AS: FIRST-IN-FIRST-OUT

INTENT: to allow manipulation of objects from the collection in a strictly specified order, such that an object, which arrived first, is consumed first

MOTIVATION:
The QUEUE pattern allows a variable number of objects to queue in-between two steps in the process, providing the means for manipulation of objects in a strictly specified order. As it was mentioned in the QUEUE pattern, there are many scheduling policies, according to which manipulation of queued objects can be done. In some situations, there is a need to retrieve objects from the queue in the order of arriving.

PROBLEM DESCRIPTION:
Assume that a collection of objects in form of a queue is given and that it is necessary either to add an object to the collection or select an object from the collection and remove it from the queue, ensuring that an object, which arrived first, is retrieved first.

SOLUTION:
In order to enforce elements of a queue to move in the order of arrival, use a specialization of Solution 1 of the QUEUE pattern. Add new elements to the tail of the list, where the queued elements are stored, and remove element from the head of the list.

Implementation of Solution:
If objects to be queued are of type T, then the type of place *queue* is *LT*. This is a list type "*color LT = list T;*".

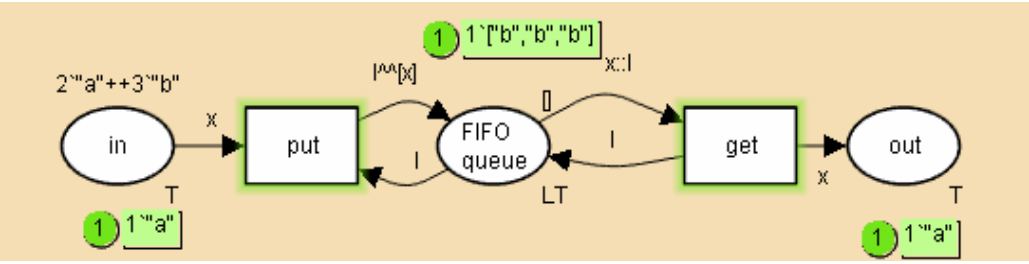


Figure 20

Transition *put* adds an objects *x* to the end of the list *l* in with help of the concatenation function: *l^^[x].* After the object *x* is added, the updated list is returned as an input to the transition *put*. Transition *put* removes the first element of the list *x::l* and puts an updated list back to the *queue* place. In this ways objects arrived first are retrieved first.

APPLICABILITY:
Apply this pattern to

- Ensure strict ordering of objects, while inserting or removing them from the collection, so that an object arrived first is retrieved first.

CONSEQUENCES:
The FIFO QUEUE pattern is a specialization of the QUEUE pattern, applied in situation when multiple objects are aggregated into a collection, which is sorted in the order of arrival.

EXAMPLES:

- The city hall handles requests of citizens. In order to keep the fair waiting time, visitors which arrived first are served first.
- In order to rent a house, people register in the housing agency. Based on the date of an application, the (FIFO) order in the waiting queue is defined.

RELATED PATTERNS: this pattern is a specialization of the QUEUE pattern.

PATTERN 6: LIFO QUEUE

ALSO KNOWN AS: LAST-IN-LAST-OUT

INTENT: to allow manipulation of objects from the collection in a strictly specified order, such that the mostly recently added object is retrieved first

MOTIVATION:
The QUEUE pattern allows a variable number of objects to queue in-between two steps in the process, providing the means for manipulation of objects in a strictly specified order. As it was mentioned in the QUEUE pattern, there are many scheduling policies, according to which manipulation of queued objects can be done. In some situations, after placing objects into a queue there is a need to retrieve the mostly recently added object first.

PROBLEM DESCRIPTION:
Assume that a collection of objects in form of a queue is given, and that it is necessary either to add an object to the collection or select an object from the collection and remove it from the queue, ensuring that an object, last added is the first one to retrieve.

SOLUTION:
In order to enforce elements of a queue to move in the order of arrival, use a specialization of Solution 1 of the QUEUE pattern. Add a new object to the head of the object list, where the queued elements are stored, and remove an object from the head of the list.

Implementation of Solution:
If objects to be queued are of type T, then the type of place *queue* is *LT*. This is a list type "*color LT = list T;*".

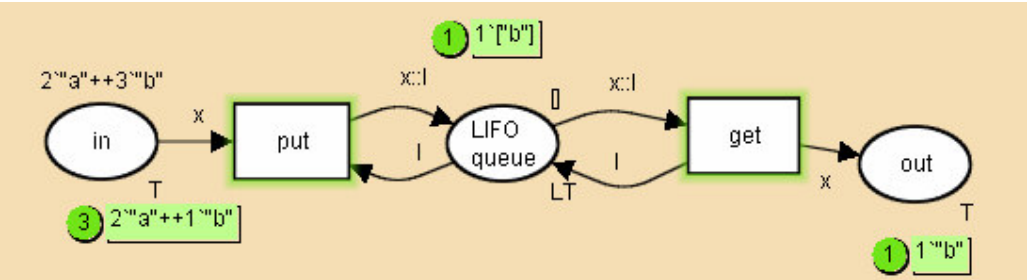


Figure 21

Transition *put* adds an objects *x* to the head of the list *l*, i.e. *x::l.* After the object *x* is added, the updated list is returned as an input to the transition *put*. Transition *put* removes the first element of the list *x::l* and puts an updated list back to the *queue* place. In this way, the most lately arrived object is retrieved first.

APPLICABILITY:
Apply this pattern to

- Ensure strict ordering of objects, while inserting or removing them from the collection, so that an object arrived first is retrieved first.

CONSEQUENCES:
The LIFO QUEUE pattern is a specialization of the QUEUE pattern, applied for manipulation of objects aggregated into a collection.

EXAMPLES:

- Inventory accounting in which the most recently acquired items are assumed to be the first sold

RELATED PATTERNS: this pattern is a specialization of the QUEUE pattern

PATTERN 7: RANDOM QUEUE

ALSO KNOWN AS:

INTENT: to allow manipulation of objects from the collection, such that objects are added to the queue in any order, and an arbitrary object is consumed from it

MOTIVATION:
The QUEUE pattern allows a variable number of objects to queue in-between two steps in the process, providing the means for manipulation of objects in a strictly specified order. In some situations, the order in which objects are inserted in the queue is out of importance, since an arbitrary object from the queue needs to be consumed.

PROBLEM DESCRIPTION:
Assume that a collection of objects in form of a queue is given, and that it is necessary either to add an object to the collection or select an arbitrary object from the collection and remove it from the queue.

SOLUTION:
In order to enforce elements of a queue to move in the order of arrival use a specialization of Solution 1 of the QUEUE pattern. New objects are added either to a tail or to a head of the list, where the queued elements are stored, and randomly removed from the it.

Implementation of Solution:
If objects to be queued are of type T, then the type of place *queue* is *LT*. This is a list type "*color LT = list T;*".

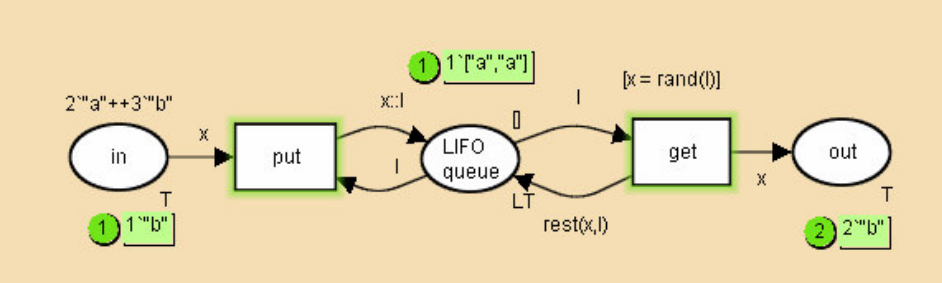


Figure 22

Put objects into the list in any order, i.e. either to the head of the list *x::l*, or to the tail of the list *l^^[x]*. In Figure 22 objects are added to the head of the list. Transition *get* picks the random element of the list:

fun rand(l) = List.nth(l, discrete(0,size(l)-1));

and then puts an updated list, i.e. the list without withdrawn element, back:

fun rest(x, h::l)= if x=h then l else h::rest(x,l);

APPLICABILITY:
Apply this pattern to

- Ensure add elements into the queue in any order, but retrieve an arbitrary element, i.e. not necessary the first or the last element of the queue, from the queue.

CONSEQUENCES:
The RANDOM QUEUE pattern is a specialization of the QUEUE pattern, applied in situation when multiple objects are aggregated into a collection.

EXAMPLES:

- In order to rent a house, a person must subscribe in a housing agency. After subscription, registered members may react on the available houses. A notary of the housing agency peeks randomly a person who will get the house.

RELATED PATTERNS: this pattern is a specialization of the QUEUE pattern

PATTERN 8: PRIORITY QUEUE
ALSO KNOWN AS:

INTENT: to allow manipulation of objects from the collection in the order of the objects' priority

MOTIVATION:
In many systems, as solution 1 of the QUEUE pattern describes, there are buffers where a variable number of objects, aggregated into a collection, are queued in-between two steps in the process. The order in which the objects are being added/ removed to/from the collection may involve object properties, thus be based on the priority associated with an object.

PROBLEM DESCRIPTION:
Assume that a collection of objects in form of a queue is given, and that it is necessary either to add an object to the collection or select an object from the collection and remove it from the queue. The QUEUE pattern solves this problem by capturing predefined ordering algorithm into the net structure and functions. Specializations of the QUEUE pattern treat all objects in uniform way, i.e. no matter what the value of an objects is, this object is retrieved only when its turn comes (first-in-first-out, last-in-first-out, etc). However, they do not allow the order of objects in a queue vary depending on certain object properties, i.e. the value of an object, or a priority associated with it.

SOLUTION:
In order to allow manipulation of objects in the order, defined by object-specific properties, determining the object priority, use the *priority queue*. The priority queue is a specialization of Solution 1 of the QUEUE pattern. When accessing objects in a queue, the object with the highest priority is removed first. A priority queue has a largest-in, first-out behavior.

Assumptions:
1) Sorting of the queue based on the objects' priority is done upon objects' insertion. See corresponding implementation alternatives 1 and 2.
2) Sorting of the queue based on the objects' priority is done upon objects' retrieval. See corresponding implementation alternatives 3 and 4.
3) Sorting of the queue based on the objects' priority after objects' insertion but before objects' retrieval.

Implementation 1 of Solution 1
Sort the collection of the queued objects upon insertion in the order of ascending priority. Define the priority of an object, based on the object's value compared to the values of objects already stored in the collection. First element of the list, i.e. an object with the highest priority, is retrieved first.

Assumption: initially, the collection of objects stored in place *Objects*, is either empty or sorted.

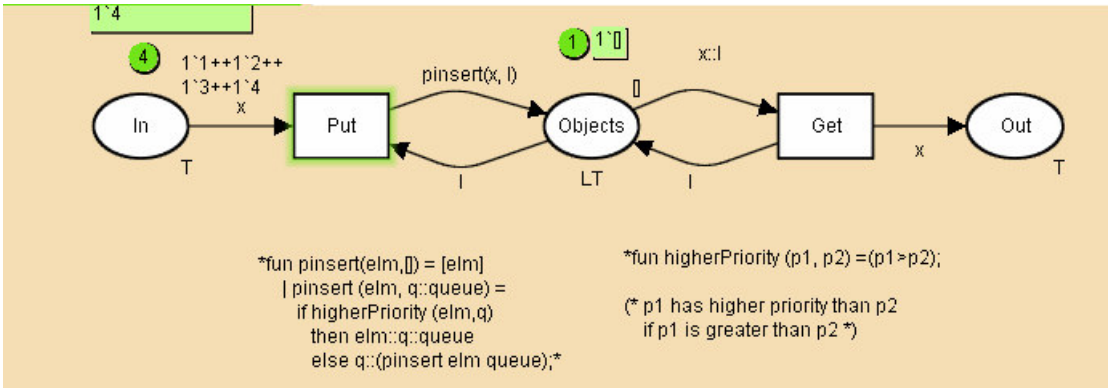


Figure 23

- Place *Objects* is of type *LT*. This is a list type "*color LT = list T;*", which collects objects of type T. In this example, objects are sorted in the order of the highest priority, i.e. an object with the highest priority is withdrawn from the queue first. In this example, from two objects of integer type the object with the highest priority is the one, whose value is bigger (see function *higherPriority*).

- Objects stored in place *Objects* are stored in ascending order, i.e. the first element has the largest value, while the last element – the smallest value. Transition *put* initially defines the priority of the new object and inserts it in the queue, ensuring that the queue is sorted properly (see function *pinsert()*). In its turn, transition *get* removes an object with the highest priority, and correspondingly the largest value, from the queue by taking the first element of the collection *x::l*.

Implementation 2
Sort the collection of the queued objects upon insertion in the order of ascending priority. In contrast to the implementation 1, the object's priority is not calculated based on the value associated with an object, but is passed as a separate value coupled to the corresponding object. Since the collection of the queued elements is composed from pairs (*obj_value, priority_value*), the second element of a pair, i.e. priority, is a parameter for sorting. Similar, a first element that will be retrieved from the queue is an element with the highest priority.

Assumption: initially, the collection of objects stored in place *Objects*, is either empty or properly sorted.

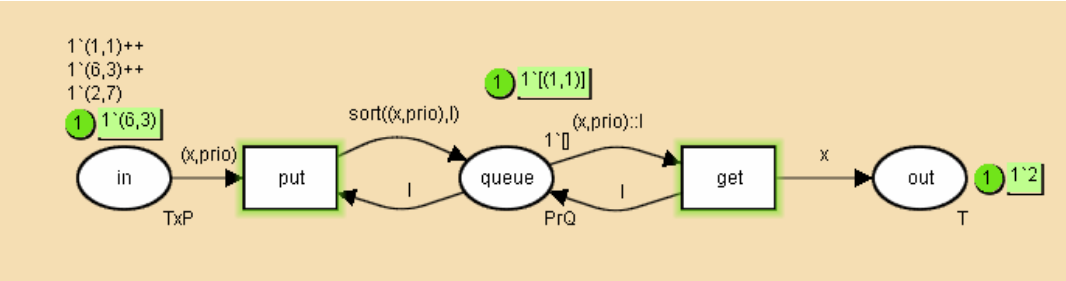


Figure 24

Figure 24 shows that initially three pairs *(5,1), (6,3)* and *(2,7)*, where first the element is an object's value, and the second element is the priority, are supplied to transition *put*.

- Transition *put* takes a pair and sorts the queue using the function *sort() :*

```
sort (x,p) [] = [(x,p)]   | sort (x,p) ((y,q)::queue) =
 if higherPriority (p,q)  then (x,p)::(y,q)::queue   else (y,q)::(sort (x,p) queue);
```

- The collection of objects *l* is sorted. Transition *get* takes the first pair from a list *(x,prio),* which is a pair with a highest priority *prio*, and returns back updated list *l.*

Implementation 3
Insert objects in the queue in any order, such that objects stored in the collection are not sorted. Sort the collection, based on the value of an object, upon retrieval.

In this example, a merge-sort algorithm is used for sorting the list in the descending order, however any other sorting algorithm may be used instead.

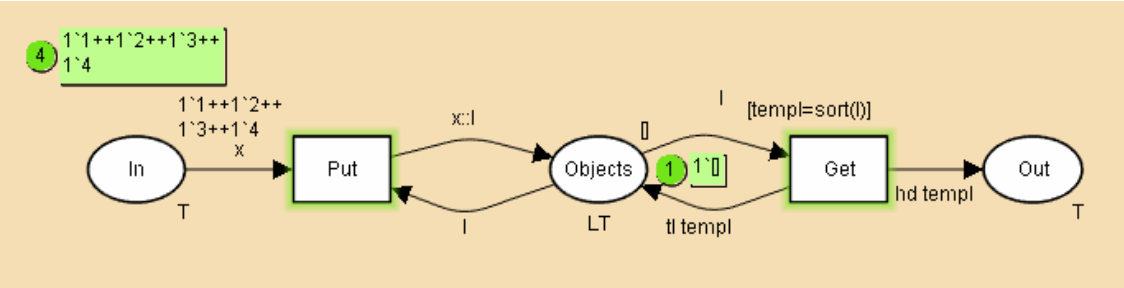


Figure 25

- Insert objects into the collection *l* in any order, i.e. either at the beginning *x::l* or at the end of the list *l^^[x]*.
- Add a transition guard *[templ=sort(l)]* to transition Get, the result of evaluation of which is assignment of the sorted list *l* to a new list *templ*. The first element of this list, i.e. an object with the smallest value *hd templ*, is passed to place *Out*, while the rest of the list *tl templ* is put back to place *Objects.*

Alternatively, one could avoid referring to the tail and head of the list by using the following construct:

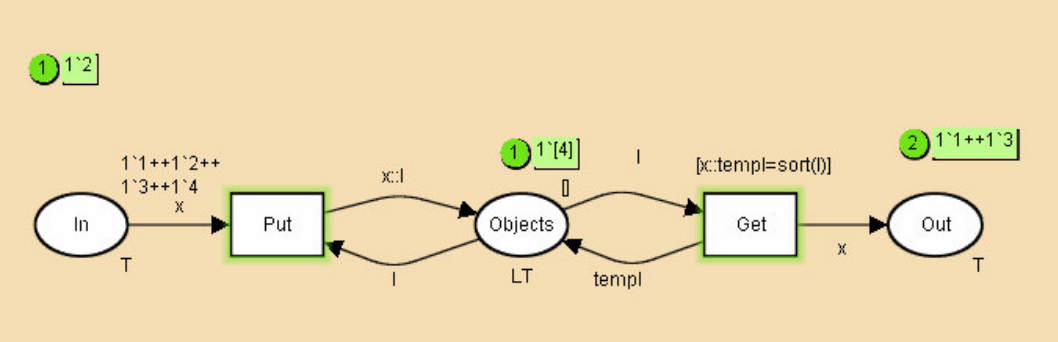


Figure 4

Implementation 4
Insert pairs of objects and priorities associated with them objects in the queue in any order. Sort the collection, based on the value of object priority, upon retrieval.

Pairs of objects and priorities associated with them are stored in pairs in the list *L* in any order. However, when an object needs to be retrieved from the collection, the list *L* is sorted based on the value of the priority element. This implementation, similar to implementation 2, uses pairs of elements, where priority of an object is not defined upon the value of an object, but is already given; and similar to implementation 3 uses sorting upon retrieval with help of the merge-sort algorithm. Note that any other sorting algorithm can be used instead.

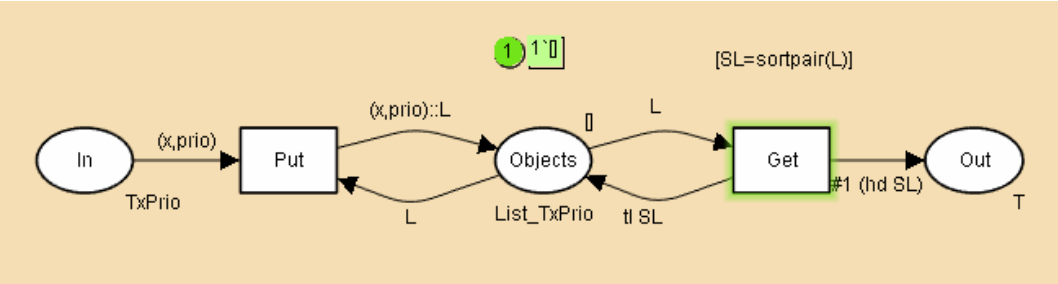


Figure 26

The implementation of the merge algorithm function is the same as in *Implementation 3,* with only difference that value *sortpair* is used instead of value *sort.*

val sortpair = mergesort TxPrio.lt;

In addition, a new sorted list *SL* is created. Since this list consists of pairs, only the first element of a pair, i.e. the value of an object is passed to place *Out, i.e. #1 (hd SL).*

In case if the whole pair, i.e. an object together with its priority, should be passed to place *Out*, use the diagram structure shown in Figure 26 and an arc inscription *hd SL* correspondingly.

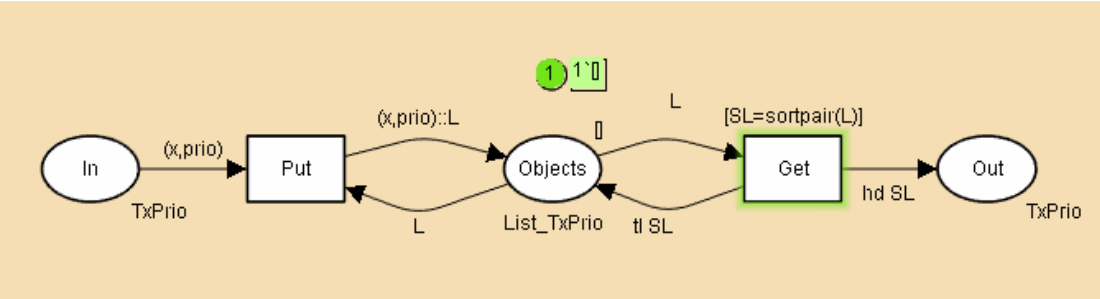


Figure 27

Implementation 5

Objects are handled in the FIFO (first-in-first-out) order. Sorting of objects is neither done upon insertion nor upon retrieval. In contrast to all considered implementations, sorting here is done externally, i.e. with help of transition *Sort*. This transition can fire when a new object is added to the collection, because of which the ordering balance in the collection is lost.

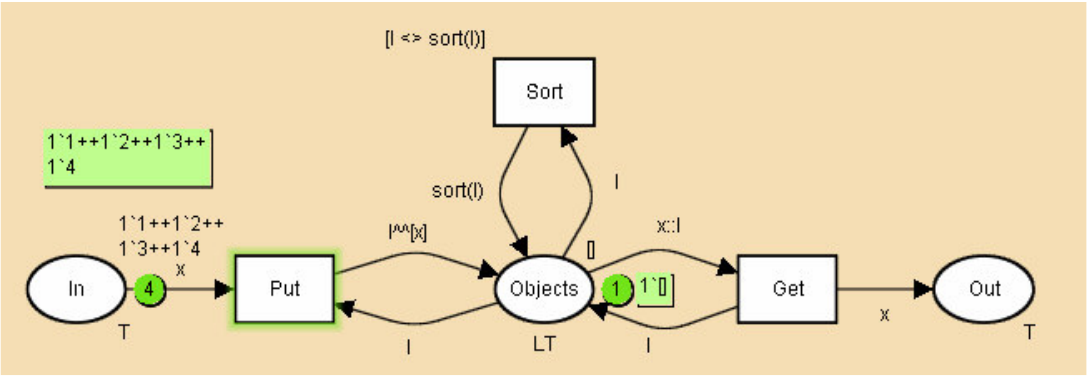


Figure 28

This implementation alternative is "non-safe", given that there is no strict ordering of firing transitions *Sort* and *Get.* In an ideal case, transition *Sort* must fire before transition *Get* in order to ensure that all elements of the collection are sorted and an element with a higher priority is taken first. However, it might happen the other way around. If a new element is added to the top of the list, and its priority is lower than of the preceding element, it should be placed by transition *Sort* in the correct location. Assume, that transition *Get* fired first, before the list was sorted. In this case, a newly arrived element with a lower priority will be consumed and the element with the highest priority will stay in the collection.

APPLICABILITY:

Apply this pattern to

- Ensure retrieval from a queue in order defined by object priorities, which are a-priori predefined or defined based on object properties like price, weight, age, etc.
- Sort a queue of objects in the order of ascending/descending priorities for structuring/organization purposes.

To apply this pattern, one should define how object priority is defined and the moment of sorting, i.e. upon insertion, when inserted, or upon retrieval from the queue respectively.

CONSEQUENCES:

The priority queue pattern allows manipulating of objects in the order, defined by object-specific properties. Note that to make this possible, this pattern makes use of a sorting algorithm, hidden inside of the functions. There are many sorting algorithms developed up

until now, which differ in efficiency and speed of sorting. We leave selection of the sorting algorithm out of the scope of this pattern, since this information can be found elsewhere.

Implementations 1 to 4 are safe in the sense that they ensure the retrieval of objects in the order of the specified priority. Implementation 5 in contrast to other implementation alternatives, also performs sorting of objects, however is not safe, since it cannot guarantee that the sorting will be done on the right moment, i.e. preceding the object retrieval.

EXAMPLES:

- The service desk of a company that distributes coffee-machines, handles complaints of clients. Every day there is a list of urgent complaints received, which must be solved within the same day. However, if too many complaints are received at the same day, and the service desk is not able to handle them all in time, they are scheduled as tasks with the highest priority for the next day.

RELATED PATTERNS: this pattern is a special case of the QUEUE pattern

PATTERN 9: CAPACITY-BOUNDING

ALSO KNOWN AS: FEEDBACK, ANTI-PLACES

INTENT: to prevent over-accumulation of objects in a certain place

MOTIVATION:
Originally, places in CPN are unbounded and may accumulate a non-limited number of tokens. In some situations, for instance for modeling of a network buffer, it is necessary to limit the number of tokens, which a certain place is allowed to contain.

PROBLEM DESCRIPTION:
In the net presented in Figure 29 different resources are stored in place *Objects*, which are accessed by transitions *Put* and *Get.* Assume that it is necessary to prevent the over-accumulation of the resources due to the limited size of the storage. Let the storage size be *N*. Unfortunately, in such a construct it is not possible to limit the capacity of place *Objects* since such a feature is not incorporated in the concept of place in CPN.

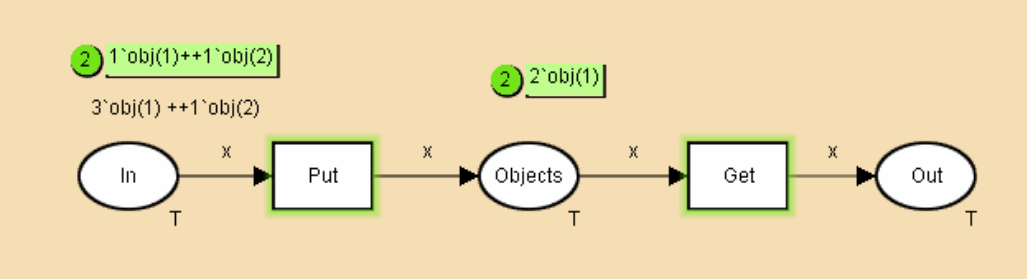


Figure 29

SOLUTION 1:
In order to prevent over-accumulation of information objects in a certain place, introduce an *anti-place*, i.e. a place corresponding to an original place, which in combination with the original place, and its incoming/outgoing transitions forms the feedback construct.

We talk about feedback if two transitions *Put* and *Get* depend on each other in such a way that *Get* both consumes an output token from *Put* and produces an input token for *Put.*

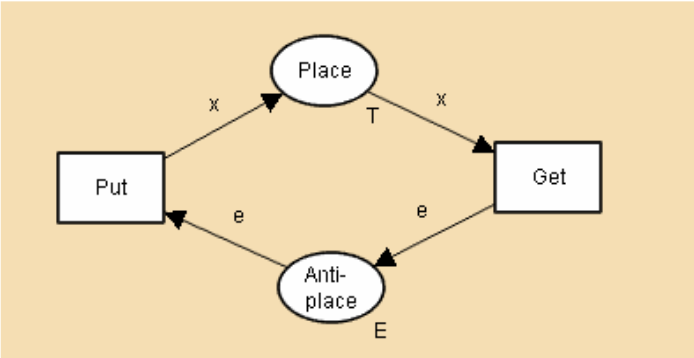


Figure 30

Implementation of Solution 1:
The list of instructions below describes how to implement the Solution 1 of the CAPACITY-BOUNDING pattern:

- Add a new place (*Anti-place*) with the type "*E*". Note that you may also use a multi-set type, which allows having multiple instances of tokens of the same type.
- Add in the initial marking of *Anti-place N* tokens "*N*`*e*", where N – is a bound of the original place *Objects*. In this particular example *N*=2, which means that at most two tokens can be present in place *Objects* at once.

- Inverse the outgoing and incoming transitions of place *Objects* and connect them to *Anti-place*, so that incoming transition of *Objects* corresponds to the outgoing transition of *Anti-place* and other way around.
- Note that this solution incorporates the BLOCKING STATE-DEPENDENT FILTER, which examines the state of an anti-place, and based on the token availability in the anti-place, defines whether it is allowed to add a new object.

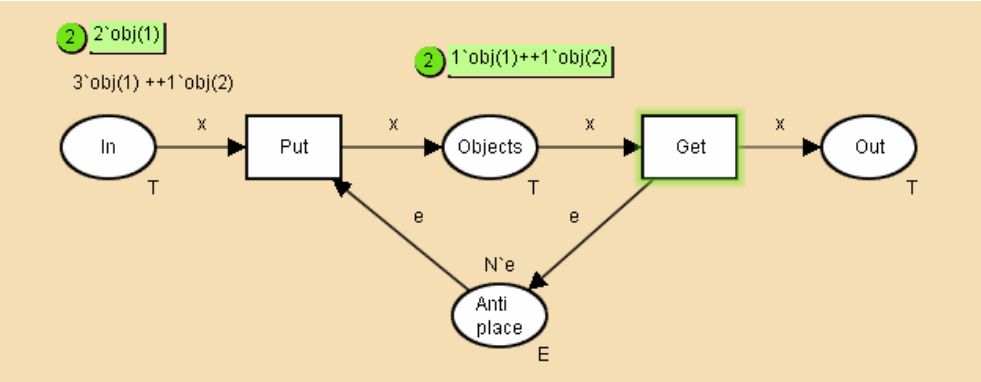


Figure 31

SOLUTION 2:
In order to prevent over-accumulation of objects in a certain place, introduce a *counter-place*, which will count how many objects are present in the place. Use a BLOCKING STATE-DEPENDENT FILTER to examine a state of the counter and prevent adding of new objects if the counter reached the maximal place capacity.

Implementation of Solution 2:
The list of instructions below describes how to implement Solution 2 of the CAPACITY-BOUNDING pattern:

- Add a new place *Counter* with the type *INT,* which will count how many objects are accumulated in place *Objects.*
- Connect the counter to the transitions *Put* and G*et,* which add and remove objects from place *Objects.* When a new object is added to the *Objects* the counter will be incremented *n+1*. When an object is removed from place *Objects*, the counter will be decremented, and thus stay up-to-date.
- Declare the value *N,* to represent a bound of the maximal capacity of place *Objects*. In this particular example *N=2*, which means that at most 2 tokens can be present in place *Objects* at once.
- Add a transition guard *[n<N]* to transition *Put*, which will compare the status of the counter, i.e. how many objects are contained in place *Objects,* with the maximal capacity of this place. If there is some free capacity available, then a new object can be added. Otherwise, transition *Put* will stay disabled until some of the objects will be removed.
- Note that this solution involves the BLOCKING STATE-DEPENDENT FILTER pattern, which examinees the state of the counter and prevents from adding new objects if the capacity of the place has been reached.

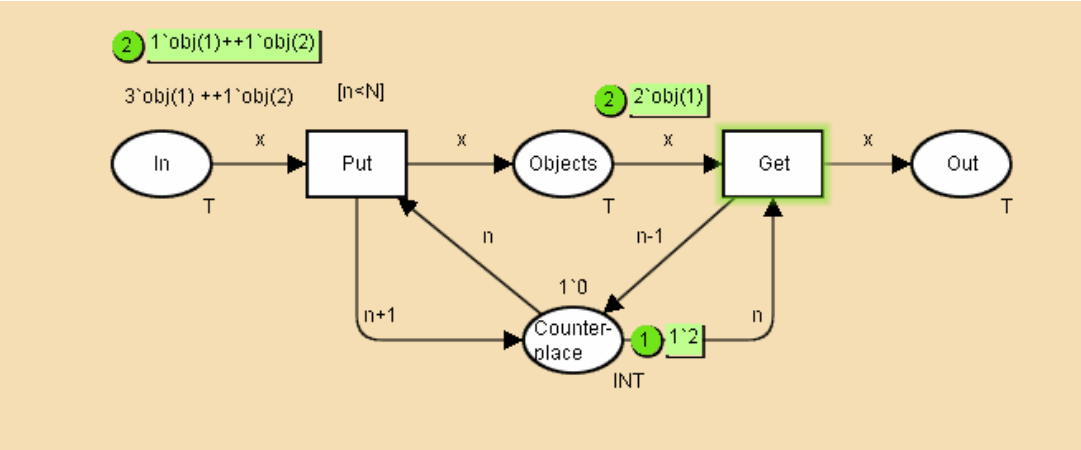


Figure 32

SOLUTION 3:
In order to prevent over-accumulation of information objects in a certain place, aggregate objects into a collection using the solution of the AGGREGATE OBJECTS pattern. Use the solution of the BLOCKING STATE-DEPENDENT FILTER to examine the current size of the collection and prevent adding new objects if the collection size reached the maximal place capacity.

Implementation of Solution 3:
The list of instructions below describes how to implement Solution 3 of the CAPACITY-BOUNDING pattern:

- Change the type *T* of place *Objects* to the collection type "*LT=list T*".
- Apply one of the variants of the QUEUE pattern to specify in which order objects must be retrieved from the collection. In this example, the LIFO QUEUE pattern is applied.
- Declare the value *N,* to represent a bound of the maximal capacity of place *Objects*. In this particular example *N=2*, which means that at most 2 tokens can be present in place *Objects* at once.
- Add a transition guard *[size(l)<N]* to transition *Put*, which will evaluate the size of the collection, i.e. how many objects it contains, and compare it with the maximal capacity of this place. If there is some free capacity available, then a new object can be added. Otherwise, transition *Put* will stay disabled until some of the objects will be removed.

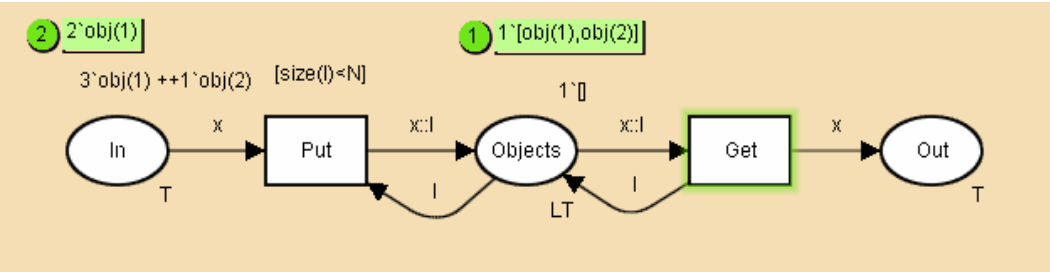


Figure 33

APPLICABILITY:
Apply this pattern to

- Bound the capacity of a place, which is used as a buffer, or dynamic storage with variable number of objects.

CONSEQUENCES:
This pattern provides a means for bounding the capacity of places. However, it is not applicable for the places, an upper bound for which cannot be defined or does not exist.

All three solutions presented in this pattern require knowledge about the maximum number of objects a place can hold. In contrast to solutions 1 and 2, where every object is represented

by a separate token, the solution 3 aggregates all objects into one collection, and provides extra flexibility by allowing performing operations on a group of objects at once.

Although the second solution has more complex diagram then the second solution, it allows monitoring the exact value of the current place capacity and using it in a model elsewhere.

EXAMPLES:

- A supermarket has a set of items to be sold to the customers. Since the storage capacity of the supermarket is limited, it is not acceptable to order more items then the supermarket area allows to store.
- A souvenir company supplies shops with different kinds of gifts packed in the wooden boxes. Limited number of gifts can be placed in the box. Placing more gifts then allowed will leas to damaging the shape of gifts.

RELATED PATTERNS: this pattern uses the BLOCKING STATE-DEPENDENT FILTER in all solutions; solution 3 uses the AGGREGATE OBJECTS pattern in addition.

PATTERN 10: INHIBITOR ARC
ALSO KNOWN AS:

INTENT: to support “zero”-testing of places

MOTIVATION:
In some situations when using CPN, there is a need to use an inhibitor arc. An inhibitor arc is an arc connecting a place and a transition, which cannot fire if the input place along the inhibitor arc contains at least one token. Eventually, it might be necessary to have a transition, which is enabled if a place is empty.

PROBLEM DESCRIPTION:
In Petri nets, it is easy to test the presence of a token in a certain place by firing a transition, which consumes the correspondent token. However, it is not possible to test the absence of tokens in a place. Consider the situation presented in Figure 34. Objects of type *T* are placed on and taken from place *Object*.

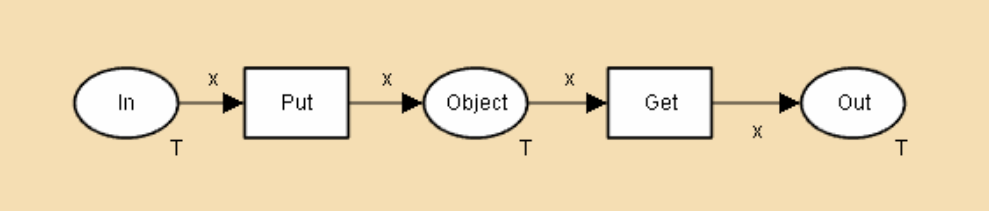


Figure 34

Now suppose that there is a cycle (see Figure 35), which involves a consumer of the objects. If the consumer is *Ready*, it can get an object and process it. If there are no objects to be consumed, then the consumer goes to the *Sleeping* state. In this example, transition *Sleep* can only fire if place *Object* is empty. A variant of this situation is the situation where place *Object* does not hold a token with a specific value.

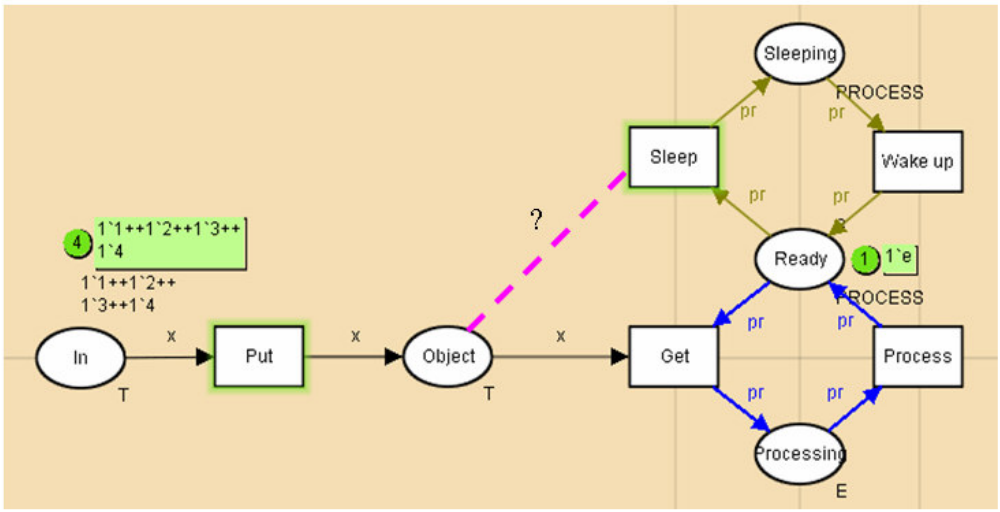


Figure 35

As it was already mentioned, according to the firing rule of the Petri net it is not possible to fire transition *Sleep* when no tokens (with specific value) are present in place *Object*. Thus, the addressed behavior of inhibitor arc is directly not supported.

SOLUTION 1:
In order to test whether no objects are contained in a place, able to store unlimited number of objects, use the AGGREGATE OBJECTS pattern to organize objects in a collection and connect an *inhibitor arc* to it.

Implementation of Solution 1:
Figure 36 demonstrates how to add an inhibitor arc for testing that no objects are contained in the collection.

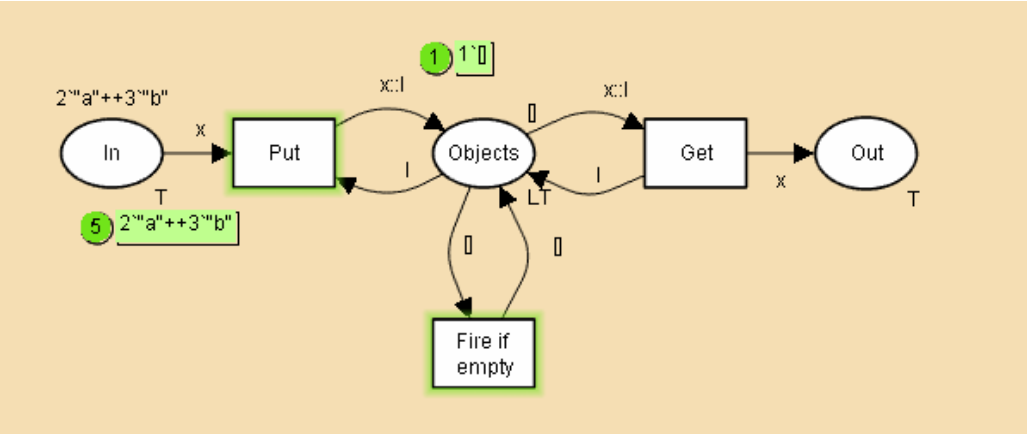


Figure 36

The list of instructions below describes how to implement the Solution 1 of the INHIBITOR ARC pattern:

- Replace place *Object*, which contains objects of type *T*, by place *Objects,* which is a collection of type *LT*. In this case a list type "*color LT = list T;*" is used. Note that the order of objects is out of importance.
- Change initial marking of place *Objects* to the empty list *[],* if originally place *Object* contained no elements. Otherwise, fill in the list with the corresponding elements.
- Replace an inhibitor arc by a bi-directional arc with an inscription *[]* (this arc is highlighted with the pink color in Figure 3). By means of the specified arc, transition *Sleep* will become enabled only when the collection stored in place *Objects* is empty, i.e. no objects are stored in the collection.

In this example transition *Get* is coupled to the guard function [*l*<>*[]*], which ensures that this transition may fire only if the list of objects *l* is not empty. This allows an inhibitor arc react upon the empty collection.

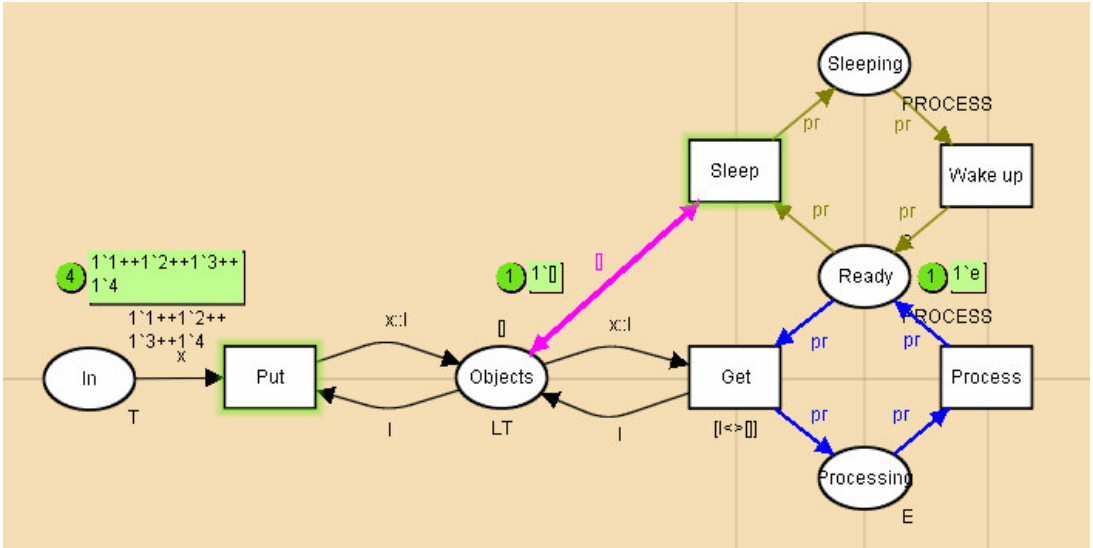


Figure 37

SOLUTION 2:
In order to test whether no objects are contained in a place, able to store a limited number of objects, use an "anti-place" described in Solution 2 of the CAPACITY-BOUNDING pattern and connect an *inhibitor arc* to it.

Implementation of Solution 2:

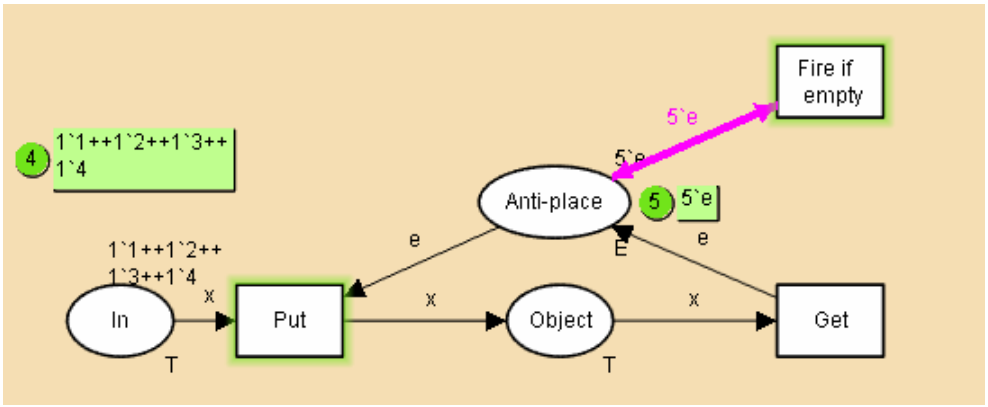


Figure 38

The list of instructions below describes how to implement Solution 2 of the INHIBITOR ARC pattern:

- Add an extra place *Anti-place* and connect it to the incoming and outgoing transitions (*Put* and *Get*) of the place (*Objects*), which needs to be connected to the inhibitor arc.
- Select a bound of the place, i.e. the maximum number of elements, which a place may have. Use this bound to specify the number of elements contained in the initial marking in place *Anti-place*. By doing this, anti-place turned to be a limit place (see the CAPACITY-BOUNDING pattern). In this case the bound *5* is selected
- Add a bi-directional inhibitor arc connected from place *Anti-place* to the required transition (*Sleep*).
- Add an inscription "*N'e*" to the inhibitor arc, where *N* is a bound chosen above. Transition *Fire if empty* will fire only if no objects are contained in place *Object* and correspondingly N tokens are contained in the anti-place.

Note that if place *Object* was originally bounded, an introduction of an anti-place has not changed its behavior and did not influence the correctness of the construction.

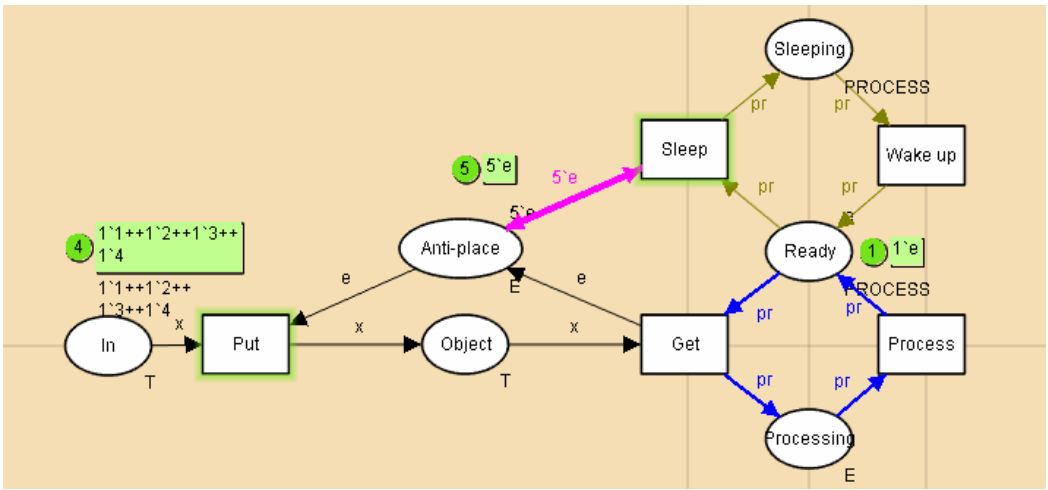


Figure 39

SOLUTION 3:
In order to test whether no objects are contained in a place, able to store a limited number of objects, use a "counter-place" described in Solution 2 of the CAPACITY-BOUNDING pattern and connect an *inhibitor arc* to it.

Implementation of Solution 3:

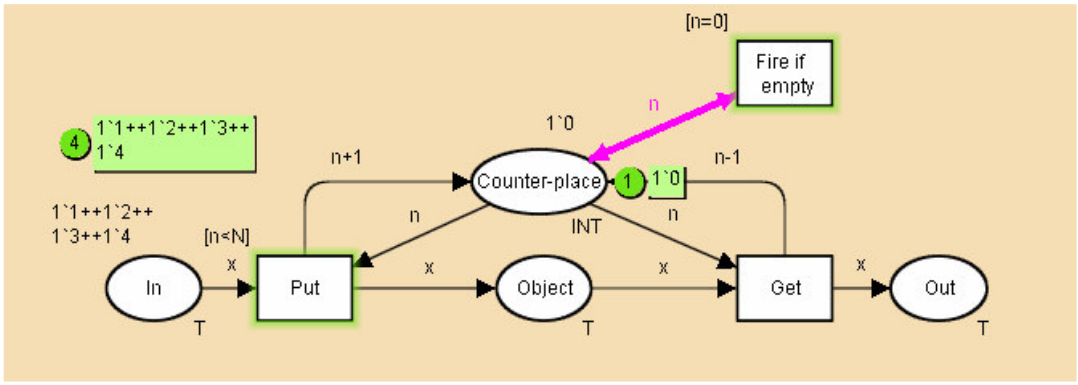


Figure 40

The list of instructions below describes how to implement Solution 3 of the INHIBITOR ARC pattern:

- Add an extra place *Counter-place* and connect it to the incoming and outgoing transitions (*Put* and *Get*) of the place (*Object*), which needs to be connected to the inhibitor arc.
- Declare the value *N,* to represent a bound of the maximal capacity of place *Objects*. In this particular example *N=2*, which means that at most 2 tokens can be present in place *Objects* at once.
- Add a transition guard *[n<N]* to transition *Put*, which will compare the status of the counter, and prevent adding new objects if the place capacity has been reached.
- Add a bi-directional inhibitor arc connected from place *Counter-place* to the required transition (*Sleep*).
- Add a transition guard *[n=0]* to transition *Sleep*, to specify that this transition may fire if no objects are contained in place *Object.*

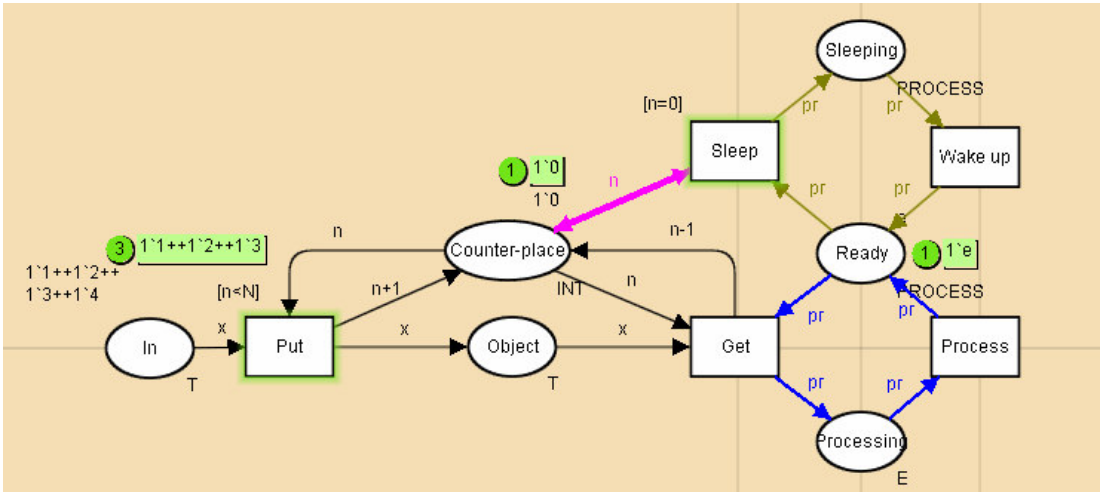


Figure 41

APPLICABILITY:
Apply this pattern to

- Test the absence of tokens or presence of tokens with specified properties in place with either bounded or unlimited capacity.

CONSEQUENCES:
The INHIBITOR ARC pattern offers three solutions. The first solution involves the aggregation of objects and should be applied when the size of the collection of objects is not bounded. This solution is a trade-off between the high level of encapsulation, i.e. hiding the behavior inside the functions, and a high degree of net flexibility. In its turn, the second and third solutions make use of anti-place and counter-place correspondingly, and it should be applied only if a place, connected to the inhibitor arc, has a limited capacity.

This pattern is extended by the COLORED INHIBITOR ARC pattern for testing of "non-containment" property of places to check that an object satisfying a certain property is not present in a place.

EXAMPLES:

- A medical assistant makes appointments with patients through the telephone. As long as patients are on the phone, the assistant continues handling their requests. When no patients left on the phone, the assistant switches to the patient waiting in the queue at the reception counter.

RELATED PATTERNS: this pattern uses AGGREGATE OBJECTS and CAPACITY-BOUNDING patterns in its solutions. It is extended by the COLORED INHIBITOR ARC pattern.

PATTERN 11: COLORED INHIBITOR ARC
ALSO KNOWN AS:

INTENT: to support "non-containment" property of places

MOTIVATION:
The INHIBITOR ARC pattern allows testing the absence of tokens in a certain place. In some situations, instead of testing "zero"-property of place, it might be necessary to test the color of tokens present in a place. Eventually, it might be necessary to have a transition, which is enabled if a place does not contain a token satisfying a certain property.

PROBLEM DESCRIPTION:
In Petri nets, it is easy to test the presence of a token in a certain place by firing a transition, which consumes the correspondent token. However, it is not possible to test the absence of tokens in a certain place. Consider the situation presented in Figure 42. Objects of type *T* are placed to and taken from place *Object*.

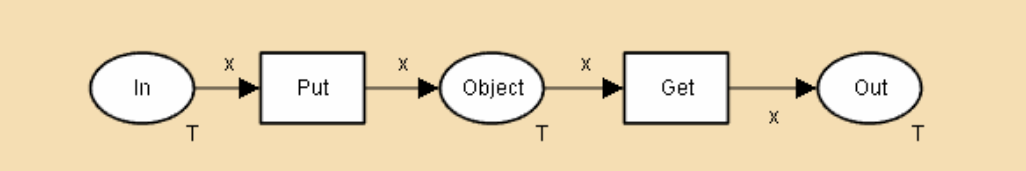


Figure 42

Now suppose that we want to check whether place *Object* does not hold a token with a specific value.

SOLUTION 1:
In order to check whether no objects satisfying a certain property are contained in a place, able to store an unlimited number of objects, organize objects into a collection as a Solution 1 of the INHIBITOR ARC pattern suggests, and extend the inhibitor arc with "color" for examining the content of the collection.

Implementation of Solution 1:
The list of instructions below describes how to implement Solution 1 of the COLORED INHIBITOR ARC pattern:

- Aggregate objects of type T into a list "*LT=list T"*.
- Take the current list of objects *l* and check by means of the function in the guard of transition *Move* if *y* is an element of the list. Function "*fun elt(y,[]) = false | elt(y,x::l) = if x=y then true else elt(y,l);"* is a Boolean function, which returns true if *y* is an element of the list, otherwise false.

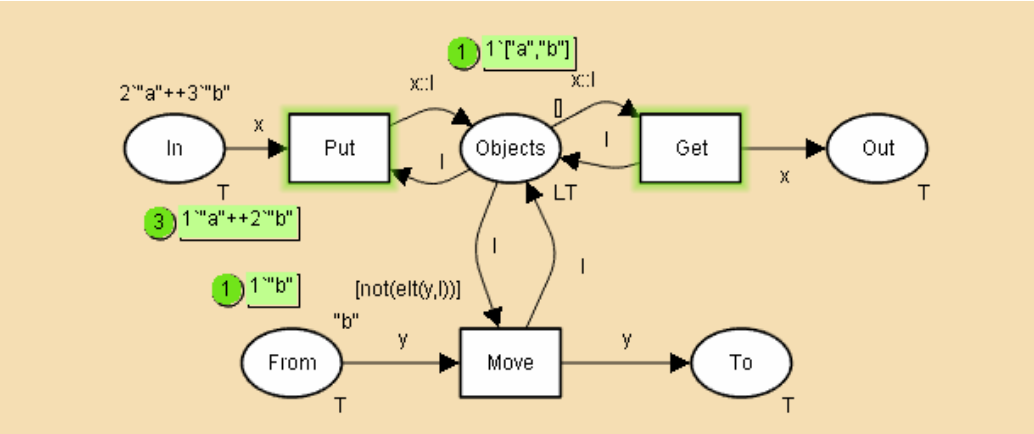


Figure 43

SOLUTION 2:
In order to test whether no objects satisfying a certain property are contained in a place, able to store a limited number of objects, use Solution 3 of the CAPACITY-BOUNDING pattern to bound the capacity of objects collection, and connect a *colored inhibitor arc* for examining the content of the tested place.

Implementation of Solution 2:
The list of instructions below describes how to implement Solution 2 of the COLORED INHIBITOR ARC pattern:

- Aggregate objects of type T into a list "*LT=list T"*. Limit the capacity of the collection by transition guard [*size(l)<N*]
- Take the current list of objects *l* and check by means of the function in the guard of transition *Move* if *y* is an element of the list. Function "*fun elt(y,[]) = false | elt(y,x::l) = if x=y then true else elt(y,l);"* is a Boolean function, which returns true if *y* is an element of the list, otherwise false.

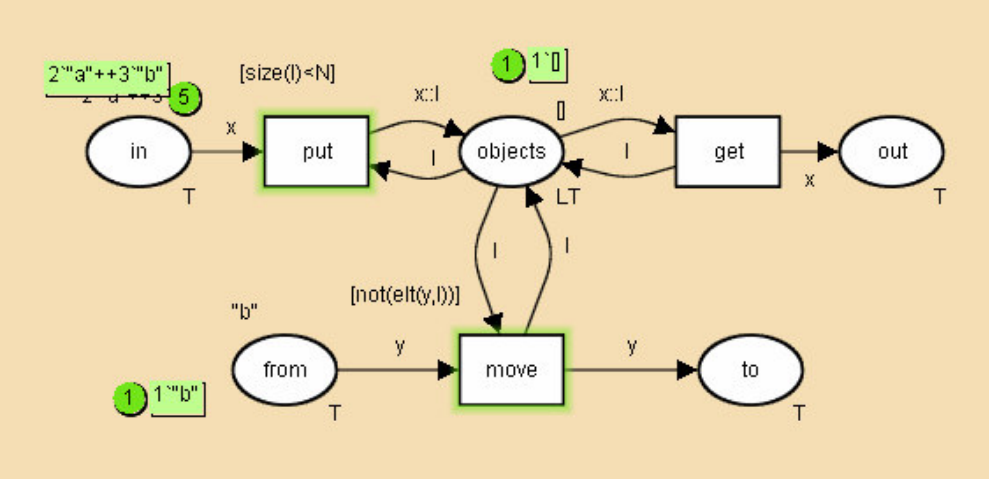


Figure 44

APPLICABILITY:
Apply this pattern to

- Test the absence of tokens satisfying a certain property in place with limited or unbounded capacity.

CONSEQUENCES:
The COLORED INHIBITOR ARC pattern offers two solutions. Both solutions allow testing the absence of an object satisfying a certain property in a collection of objects. The first solution is applicable to the unbounded collection, while the second one is applicable to the collection with a limited capacity.

EXAMPLES:

- A medical assistant makes appointments with patients through the telephone. As long as patients are on the phone, the assistant continues handling their requests. When no patients left on the phone, the assistant switches to the patient waiting in the queue at the reception counter.

RELATED PATTERNS: this pattern extends the INHIBITOR ARC pattern in its solution 1, and uses solution 3 of the CAPACITY- BOUNDING pattern in its solution 2.

PATTERN 12: SHARED DATABASE

ALSO KNOWN AS: DATA VISIBILITY MANAGER, SHARED PLACE

INTENT: to enable centralized storage of data shared between multiple transitions, supporting different levels of data visibility (i.e. local, group, or global)

MOTIVATION:
In Petri nets, a transition is only aware of data available in input places connected to it, but has no knowledge about the data inputs of other transitions. In other words, the visibility of data relatively a transition is local. In some cases, it is necessary to make data visible to a group of transitions or to all transitions contained in a model, providing the group and global visibility respectively.

PROBLEM DESCRIPTION:
Assume that a chain of transitions is given (Figure 45), and it is necessary to pass some data from the start of the chain *A* to the end *D*. Although intermediate steps do not change the data at all or change it not frequently, the data is passed through the whole sequence of transitions rather then being available upon request, i.e. only at the moment when it is needed.

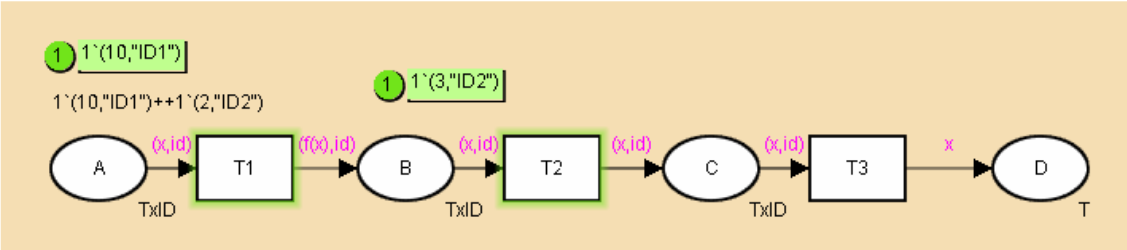


Figure 45

Data *x* with an identifier *id* is sequentially passed from place *A* to *D* through transitions *T1, T2* and *T3.* Although transition *T2* does not use the data *x,* it anyway has knowledge about it, since it passes the data through. In terms of network transfers, this leads to overloading of the traffic, increases duration of the transfer and lows down the overall performance. From the security point of view, it might be desirable to limit a set of transition to a group of transitions authorized to access data, thus preventing information outflow.

SOLUTION:
In order to centralize storage of data shared between multiple transitions, use a *shared database.* The shared database is a place that provides rapid access to data it contains by transitions, connected to it.

In order to enable the *global visibility* of data in a model, connect all transitions to the shared database. In order to limit the visibility of data to a certain group of transitions and thus obtain the *group visibility*, connect to the shared database only to those transitions, which are allowed accessing the data. The *local visibility* can be obtained by connecting only one transition to the database. Thus, number of transitions connected to the shared database defines the visibility of data in a model.

Implementation of Solution:
The list of instructions below describes how to implement the SHARED DATABASE pattern with a group visibility (a group, which is composed out of all transitions in a model, provides the global visibility):

- Introduce a place *Shared data,* where data for sharing, will be stored. Define the format of data stored in the database, providing that all data stored in the database can be referenced by means of unique identifiers (as required by the DATABASE MANAGEMENT pattern, which defines an interface for accessing the shared database). Instead of passing data through the whole chain of transitions, which has no access to the shared database, it is sufficient to pass only an identifier correspondent to the data.

- Define the level of data visibility, and thus the number of transitions, which will be connected to the database.
- Connect the identified transitions to the shared database by means of bidirectional arcs. An arc in the direction of transition supplies transition with a current value of data, identifier of which matches to the identifier of data requested by transition. For details of an interface for accessing data from a database, consult the DATABASE MANAGEMENT pattern.
- In order to organize data stored in the shared database into a data structure of the collection type, so that the whole database corresponds to a single token, the AGGREGATE OBJECTS pattern can be applied.

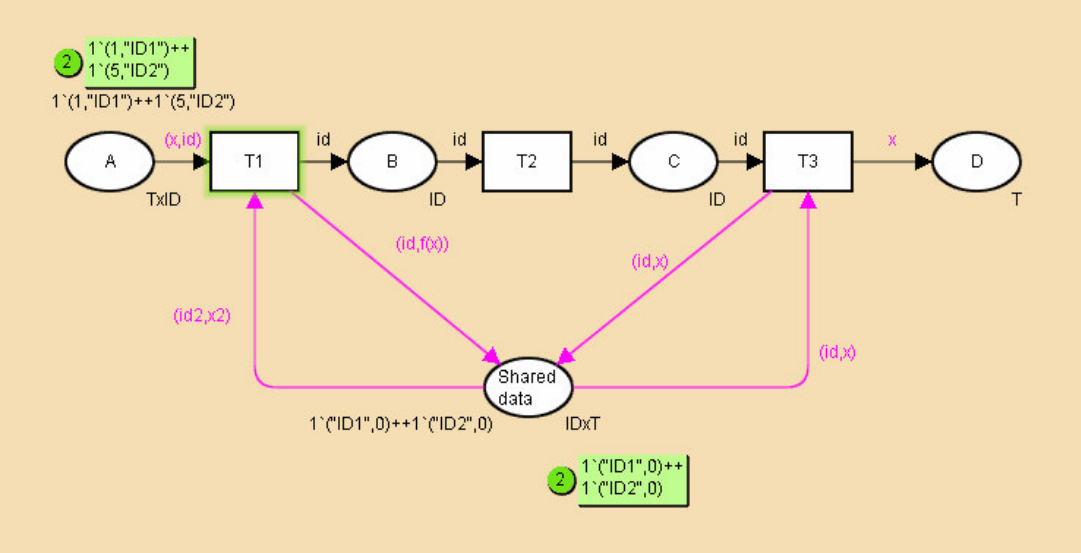


Figure 46 Shared database with Group visibility

APPLICABILITY:
Apply this pattern to

- Share some data between a group of transitions, so that each transition may access the data from the database whenever it needs to;
- Restrict an access to data by unauthorized transitions;
- Use data with different levels of visibility, for instance, local or global variables and constants.

CONSEQUENCES:
The definition of a level of data visibility is one of the most important decisions the developer of a model should take at an early stage of the development. Carefully selecting the visibility of data, may help in preventing unauthorized data access, information outflow, or making model cumbersome by passing through irrelevant or complex data. In almost every model, there is a need in one global database, which stores the persistent information (often non-variable or less-frequently variable), and several local databases, the visibility of data contained in which is limited to a certain group of transitions.

The drawback of implementing this pattern is that it can make a model look "spaghetti-alike" due to the multiple arcs connecting transitions and a shared place. The model complexity may increase dramatically if you need to introduce multiple databases, thus creating for each group an extra place with corresponding arcs. Therefore, one should make a trade-off between the increased model complexity, introduced by external shared places, and an importance of (non-)limited access to data.

As it was mentioned in the implementation section, for structuring the database, this pattern can be combined with the AGGREGATE OBJECTS pattern.

In some situations, there is a need to make data available in one shared database, also available in other locations. To make this possible, this pattern should be combined with the COPY MANAGER pattern.

EXAMPLES:

- A supervisory board is composed of ten people, from which three persons are responsible for the budget affairs. Each of the three persons has an access to the bank account. The visibility of account in this case is limited to a group of three people.
- An Internet-shop allows any visitor of the site to view the products sold by the shop. The visibility of *products* is global.
- Two companies, each of which has a number of internal projects, are involved in a joined project. The members of both companies have an access to the joined project (*global visibility*). However, the members of one company have no access to the internal projects of the other company, since the visibility of those projects is limited to the group of people working in the company-owner of the project.

RELATED PATTERNS: this pattern includes the DATABASE MANAGEMENT pattern. To structure the database in one collection, this pattern can be combined with the AGGREGATE OBJECTS pattern. To make the data stored in the shared database also available at other locations, this pattern can be combined with the COPY MANAGER pattern.

PATTERN 13: DATABASE MANAGEMENT

ALSO KNOWN AS: READ/WRITE

INTENT: to specify the interface of accessing data, stored in a shared database, for read-only and modification purposes

MOTIVATION:
The SHARED DATABASE pattern provides a solution for centralizing the data shared between several transitions, allowing support of different levels of data visibility. Usually a shared database is used either as a static data provider, which contains data only for read purposes and prohibits modifications, or as a dynamic storage, where data is being accessed for read/write purposes. To make the distinction between these two types of databases, their access interfaces should be clearly specified.

PROBLEM DESCRIPTION:
Assume that two independent threads need to retrieve data from the *Shared Database* for read-only and modification purposes correspondingly. According to the SHARED DATABASE pattern, it is necessary to connect the transitions *Read-only* and *Modify* to the *Shared Database*, making the data stored in the database visible and shared between these transitions. However, when connecting transitions to the database, it is not clear what interface must be used for retrieving the data for read-only and modifying purposes correspondingly.

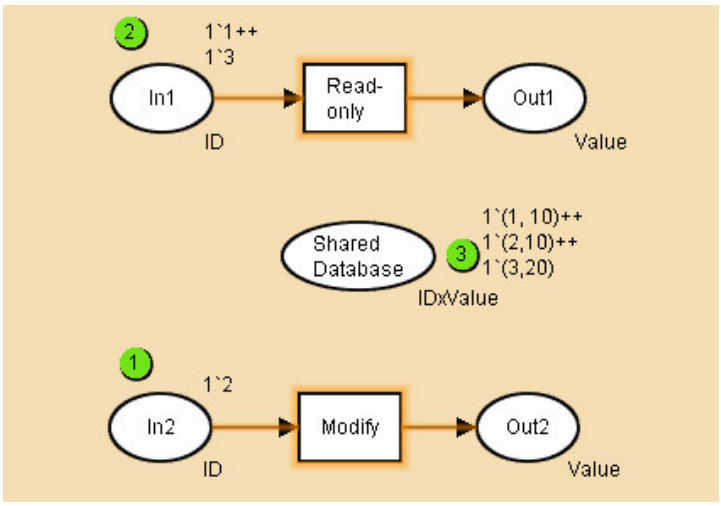


Figure 47

SOLUTION:
Assumption: the shared database stores data in the format, according to which for every data stored in the database there is a unique identifier referencing to it.

In order to retrieve a data element from the shared database, use an identifier associated with the data as a reference. For read-only purposes retrieve the value of data and place it back when finished reading. For modification purposes, retrieve the value of data, modify it, and put the modified value back to the database.

Implementation of Solution:
Figure 48 visualizes the pattern for read/write access of data stored in the shared database. It is a generalization of both read/modify operations, which allows the shared database of dynamic type both to read and modify data stored in it. The specialization of this pattern, an interface for accessing the shared database of static type, can be derived from the interface for accessing the dynamic database by removing the elements responsible for data modification and keeping the ones for read-only purposes.

- Connect the shared database *Shared DB* to transition *Modify data,* providing the value of data element with requested identifier *(id1, val1).* Note that the data identifier

serves as a key for retrieving the value of data from the database, thus using the ID MATCHING pattern.

- Add a guard to transition *Modify data*. In particular, for modification purposes the content of a function *F()* defines what value will be assigned to the data element.
- After the guard has been evaluated, the new value *val1_m* is placed back to the *Shared DB*.

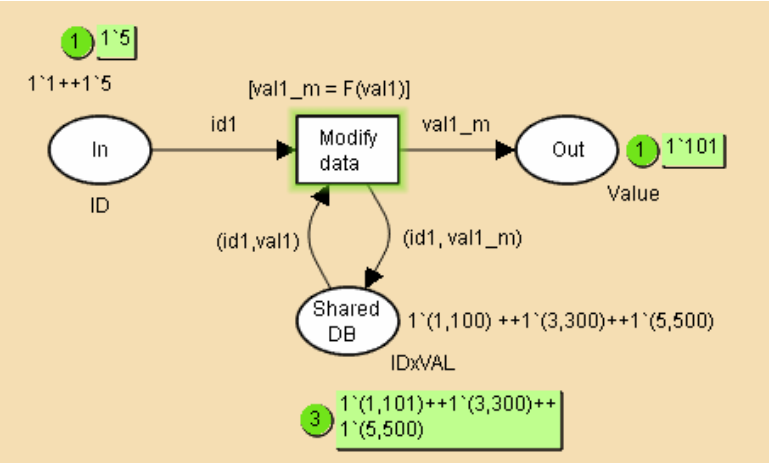


Figure 48 Read/write access

Figure 49 illustrates an interface for “read-only” access.

- To provide read-only access to the data stored in the shared database, retrieve the value of data *val1* with an identifier *id1,* supplied to transition *Retrieve* and return it to the database without being changed.

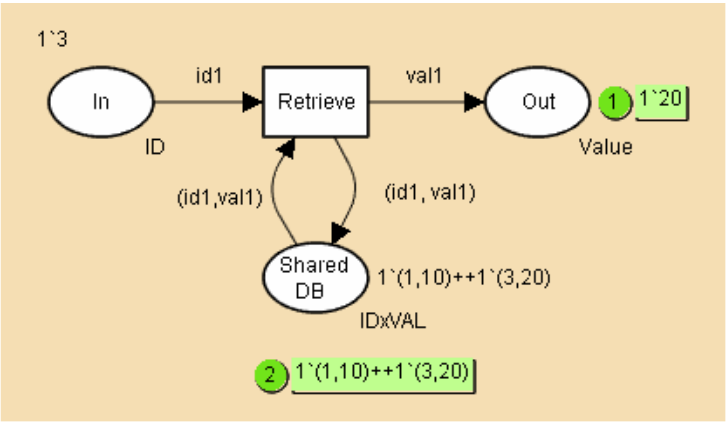


Figure 49 Read-only access

Figure 50 shows a special case of read/write access to the database: a data stored in the centralized storage needs to be retrieved and modified, so that a new modified value of the data is placed to the database, but an old value *val1* of data with *id1* is transferred to the output *Out* of transition *Access data*.

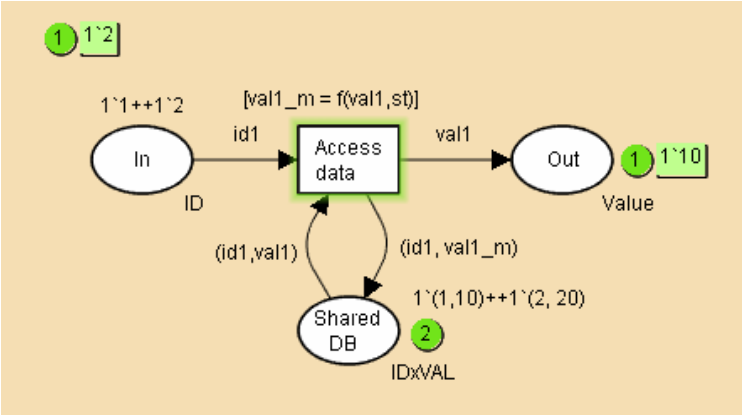


Figure 50 Retrieve and then modify

The list of steps below contains instructions for implementing access interfaces in the example presented in the *Problem description* section of the DATABASE MANAGEMENT pattern.

- Since every data in the shared database is associated with a unique identifier, the latter should be used as a reference for the data when retrieving the data from the database. Therefore, provide the values of identifiers as an input for transitions *Read-only* and *Modify*.
- For *Read-only* operation, draw an arc from the Shared Database with an inscription *(id, val1),* so that the identifier provided from *In1* is the same as of the data element, taken from the database. After reading the data, put the data back to the database by means of an arc directed to the *Shared Database* and the same arc inscription.
- For *Modify* operation, first retrieve the current value of the data with a referenced identifier. For this, draw an arc to the Shared Database in the direction of transition *Modify*. Add an arc inscription (*id2,value*), where *id2* is a value of the identifier supplied from *In2.* Add a transition guard *[val2=f(value)]*, which takes the retrieved data value as a parameter, modifies it and returns the modified value of the same type back. Draw an arc from the transition *Modify* in the direction of the *Shared Database* and put on the arc inscription the modified value *(id2, val2)* back to the database.

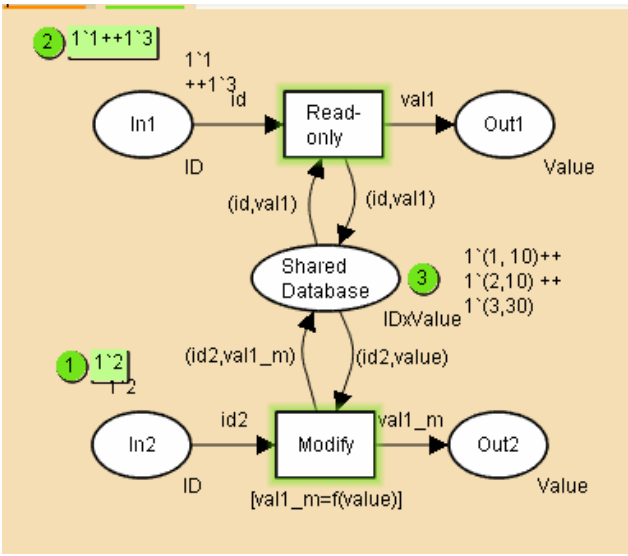


Figure 51

APPLICABILITY:
Apply this pattern in order to

- Realize an access interface for the shared database of the static type, which provides data for read-only purposes.
- Realize an access interface for the shared database of the dynamic type, which allows both reading and modifying data stored in the database.

CONSEQUENCES:
The DATA MANAGEMENT pattern clarifies interfaces for accessing the shared databases of static and dynamic types. However, this pattern does not deal with problems, which might appear during simultaneous access of data for modification by several transitions. In order to synchronize the concurrent access to shared data the SHARED DATABASE pattern should be combined with the LOCK MANAGER pattern for providing an exclusive access for all data, or with BI-LOCK MANAGER for providing a shared access for reading but an exclusive access for writing.

EXAMPLES:

- Database with bibliographical information relating to books, articles, and other published materials.
- Centralized storage of patient data in hospital, available in any functional department of the hospital.

RELATED PATTERNS: this pattern uses the ID MATCHING pattern in its solution. To solve synchronization problems of concurrent data access this pattern can be combined with the LOCK MANAGER pattern or BI-LOCK MANAGER.

PATTERN 14: COPY MANAGER

ALSO KNOWN AS: DATA REPLICATION

INTENT: to make data, stored in the shared database, available at other locations for local use, maintaining the consistency of data in all places

MOTIVATION:

In many situations, data stored in a central database needs to be accessed concurrently in different, often independent of each other, locations. Such local places need to be able to work with the data even when the connection to the central database is not available. The data stored in the central database must be up-to-date, i.e. any change to the data should be taken into account for ensuring the overall data consistency.

PROBLEM DESCRIPTION:

Assume that there is a central database, in which data of employees is stored. Different departments, i.e. financial department, housing service, education center, etc. need to use this data in their work concurrently. Due to an exclusive access to the database, one department will not be able to use the data until it is not released by another department. Such dependency results in a long waiting time and inefficient work whatsoever. In addition, under condition that departments connect to the central database through the network, if the latter is temporarily unavailable, the access to the data will be limited for all departments. Thus, there is a need in making the data stored in the central database available locally.

SOLUTION:

In order to make data stored in the shared database also available at other locations, use a *copy manager.* The copy manager replicates the data from the central database and stores it in the local storage. The copy manager maintains data consistency by updating data in the main database if some data in the local copy was modified, and correspondingly synchronizing local copies with the original databases on the regular bases for incorporating data changed in the original database.

Implementation of Solution:

The list of instructions below describes how to implement the COPY MANAGER:

- In order to be able to replicate data from the shared database, ensure that data in the shared database is aggregated into a collection, thus allowing to copy all database elements in one go. Copying one element after each other is inefficient and time-consuming, therefore is not considered.
- Create a new place *Local copy*, where a local copy of the data available in the central database will be stored. Note that the types of places *Shared Database* and *Local copy* should be the same.
- Connect the place *Shared Database,* where the central database is stored, to the newly created local database, through the transition *Replicate*, which copies all data from one place to the other.
- After data stored in the local database was modified, update the old value stored in the central database with a new one, produced by transition *Modify*. Which data and how it modified is out of the scope of this pattern. Note that there is no need to execute *Report change* transition if the data in local copy was read-only but not modified.
- Since there is no need to update the whole database after one of its elements was modified in local copy, pass only the modified data, together with its identifier, to place *Modified value.*
- Transition *Report Change* takes the whole collection of data from the *Shared Database* and replaces the value of data element with a specified *id.*

  ```
  fun update((id,a)::queue,id2,y) = if (id=id2) then (id,y)::queue else
  (id,a)::update(queue,id2,y);
  ```

- Regularly synchronize *Local copy* with *Shared Database* by executing transition *Replicate*, which consumes old data from *Local copy* and puts new data provided by *Shared Database.*

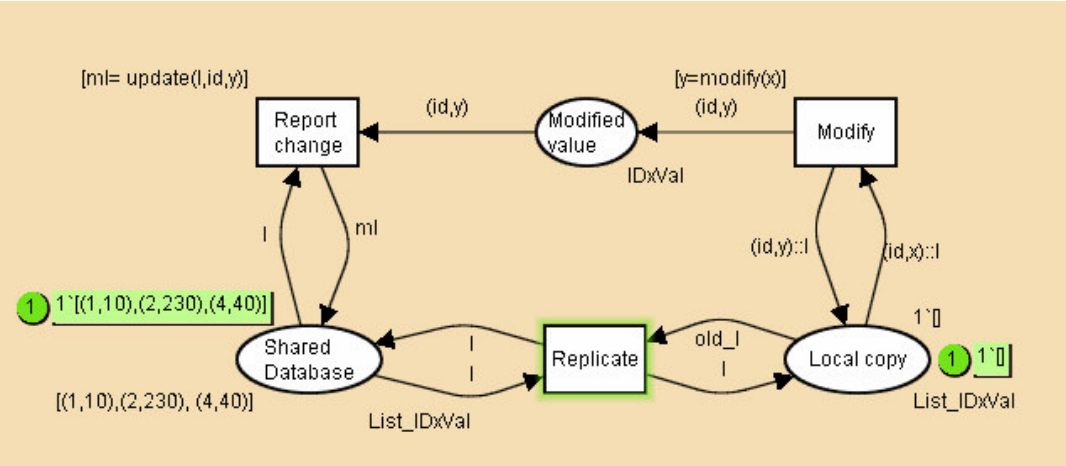


Figure 52

APPLICABILITY:
Apply this pattern to

- Provide availability of data stored in one place also at another places, by means of data replication.
- Maintain consistency between two or more databases by reporting changes and synchronizing with changes introduces by other.

CONSEQUENCES:
Since multiple local databases may modify the data stored in their copies, these changes should be communicated to the central database to ensure the data consistency. Because different sources may want to update the same data in the central database concurrently, there is a need to incorporate the (BI)-LOCK MANAGER pattern, which solves synchronization problems inherent to concurrent access.

Note that even if no data is changed in the local copy, it is necessary to periodically replicate the data for incorporating the last changes introduced either by the central database itself or by other local databases in order to provide overall data consistency.

EXAMPLES:

- Mobile workers require a reliable software solution that allows them to access their organization's data locally on a mobile device, modify this data, and synchronize the changes with a database on a remote server in a timely fashion.

RELATED PATTERNS: this pattern can be combined with the LOCK MANAGER or BI-LOCK MANAGER pattern to solve synchronization problems, which might appear during updating the database.

PATTERN 15: LOCK MANAGER
ALSO KNOWN AS: MUTUAL EXCLUSION

INTENT: to synchronize access to shared data by means of exclusive locks

MOTIVATION:
When several related processes execute concurrently, often they share some resources or data stored in a shared database. The SHARED DATABASE pattern allows multiple actors to access data stored in the shared database concurrently, but does not protect from overwriting each other's changes accidentally or reading inconsistent data due to in-progress changes.

PROBLEM DESCRIPTION:
Assume that two owners of the same bank account withdraw money from different cash-dispensers simultaneously. If both cash-dispensers access the account concurrently, it is not clear either one or both amounts will be subtracted from the account and what would be the final status. Thus, unsynchronized access to information stored in the shared database can lead to data inconsistency.

SOLUTION:
In order to synchronize access to shared data, use a *lock manager*. The lock manager allows only one actor to use a shared data at a time. In order to get an access to a shared data, an actor must acquire a lock. An actor that owns a lock must release a lock before the shared data can be used by another actor.

Implementation 1 of Solution:
The lock manager consists of two parts:
- Lock acquisition;
- Lock release.

Figure 53 illustrates the mechanism of lock acquisition.

- Place *Object Locks* contains for each of the information objects *obj*, which are shared between multiple consumers *p*, a list *cl* of actors, which obtained a lock on the correspondent object. Thus, each object is associated with a list of consumers (*obj,cl*).
- Place *Consumer Locks* contains for each of the consumers, which have access to the shared database, a list of objects, for which the consumer obtained a lock. Thus, each consumer is associated with a list of locked objects (*p,l*).

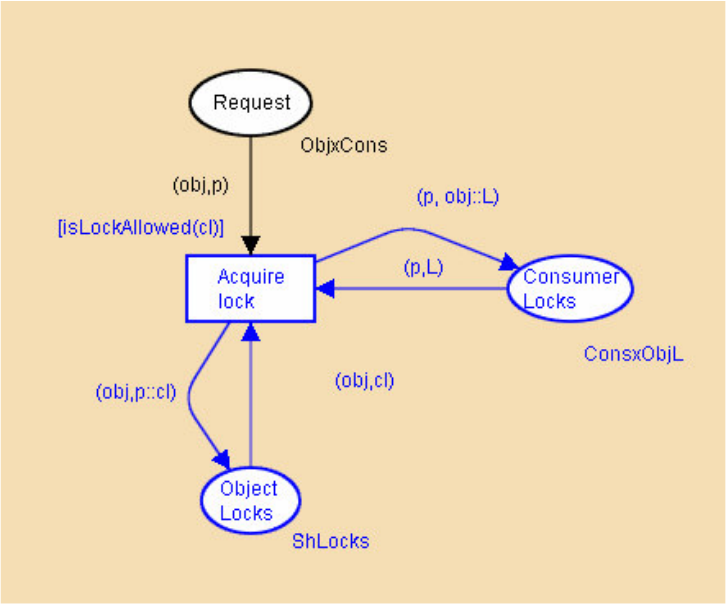


Figure 53 The mechanism of lock acquisition

- Place *Request* contains requests of consumers to access a certain object in form of (*object_id, actor_id*).
- Transition *Acquire lock* checks whether it is allowed to grant a lock for the specified object by means of *isLockAllowed()* function, which examines if there was already lock acquired for the specified object. If no lock was granted, then the actor gets the lock, and the status of *Object Locks* and *Consumer locks* are updated as it is shown in arc inscriptions, otherwise transition Acquire blocks the request if no lock can be obtained.
- Note that the described mechanism is used for acquiring exclusive locks, which means that only one consumer at a time may have a lock on a certain object. However, this mechanism is suitable for shared lock acquisition, i.e. when multiple actors may have a lock on the same object. This is why in place *Object Locks* every object is associated not with a single consumer, but with a list of consumers, which acquired a lock for this object.

Figure 54 illustrates the mechanism of lock release.

- Transition *Release lock,* after a consumer finished using an object, releases the lock on the correspondent object. Delete functions *delete(obj,L)* and *del(p,cl)* specified in arc inscriptions remove the released object from the list of objects locked by the consumer in *Consumer Locks*, and remove a consumer from the list of consumers having the lock on the released object in *Object Locks* respectively.

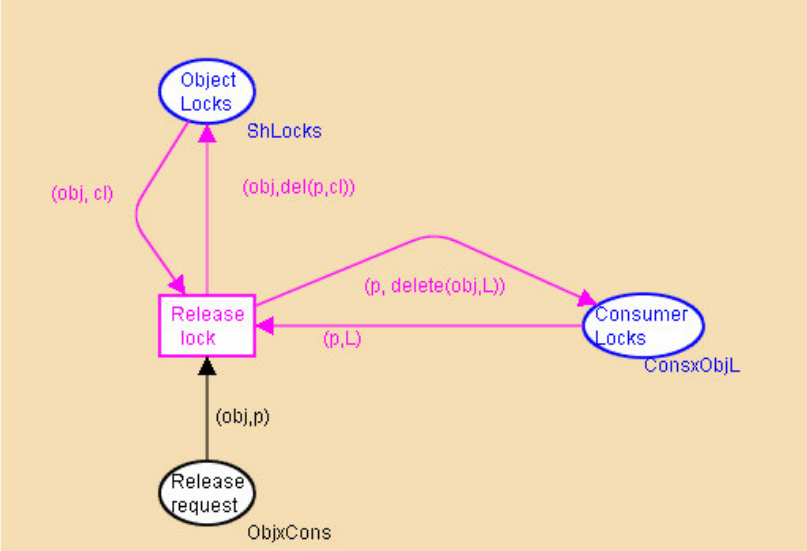


Figure 54 The mechanism of lock release

The lock manager, which contains both mechanisms for lock acquisition and lock release, is presented in Figure 55. Note that this implementation incorporates an individual lock release, which means that if an actor has locks on several objects, then the locks will be released sequentially one-by-one.

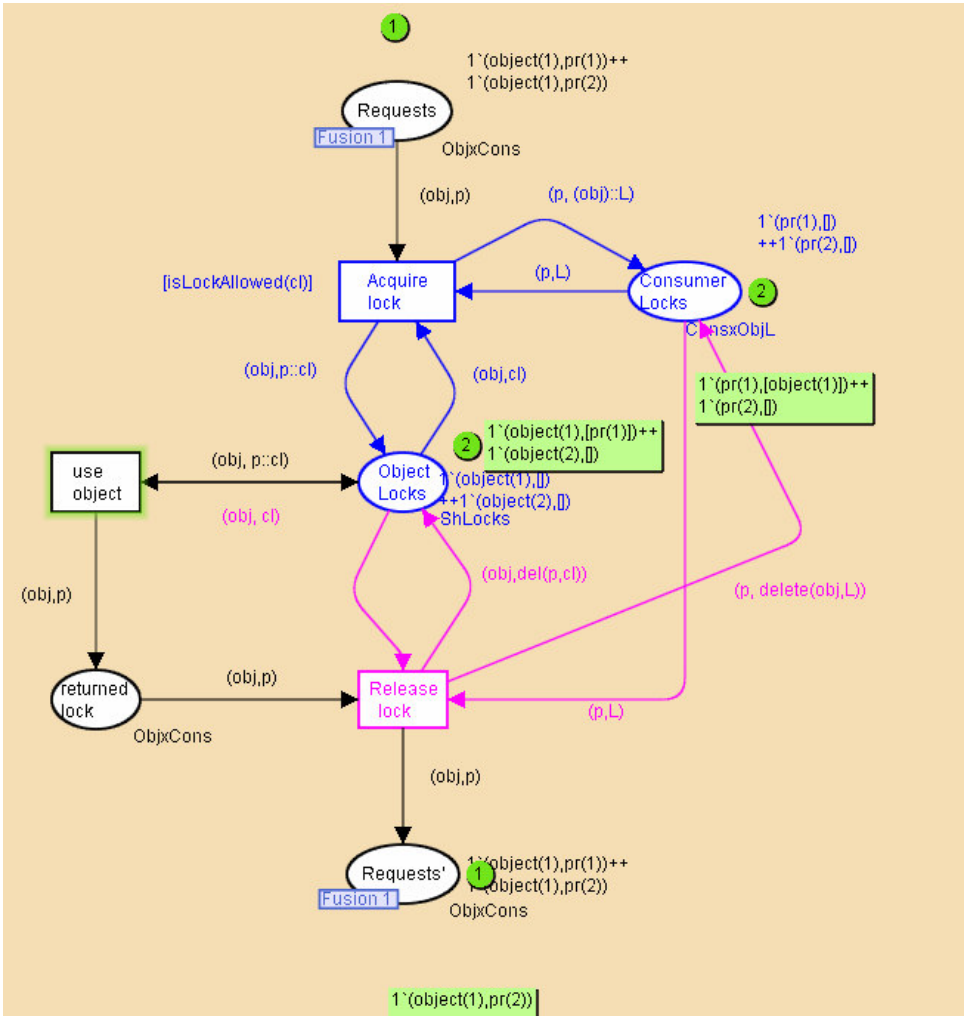


Figure 55 The lock manager (individual lock release)

Implementation 2 of Solution:
This implementation alternative, which is visualized in Figure 56, incorporates collective lock release. In contrast to individual lock release of implementation alternative 1, if an actor has locks on several objects, and the actor finished processing the objects, the locks are removed and updated in one batch. Note however that transition *Release lock* still fires multiple times. In this case, the mechanism of lock release is indirect and hidden from the collection of *Consumer Locks*.

APPLICABILITY:
Apply this pattern in order to

- Ensure synchronization of concurrent access to data shared between multiple actors, by means of exclusive locks, i.e. allowing only one actor at a time accessing the data.

CONSEQUENCES:
The LOCK MANAGER pattern solves the problem of unsynchronized access to shared data. We suggest to apply this pattern in combination with shared databases described in the SHARED DATABASE pattern, in order to maintain consistency of data in the database and provide exclusive access to the data stored in the database.

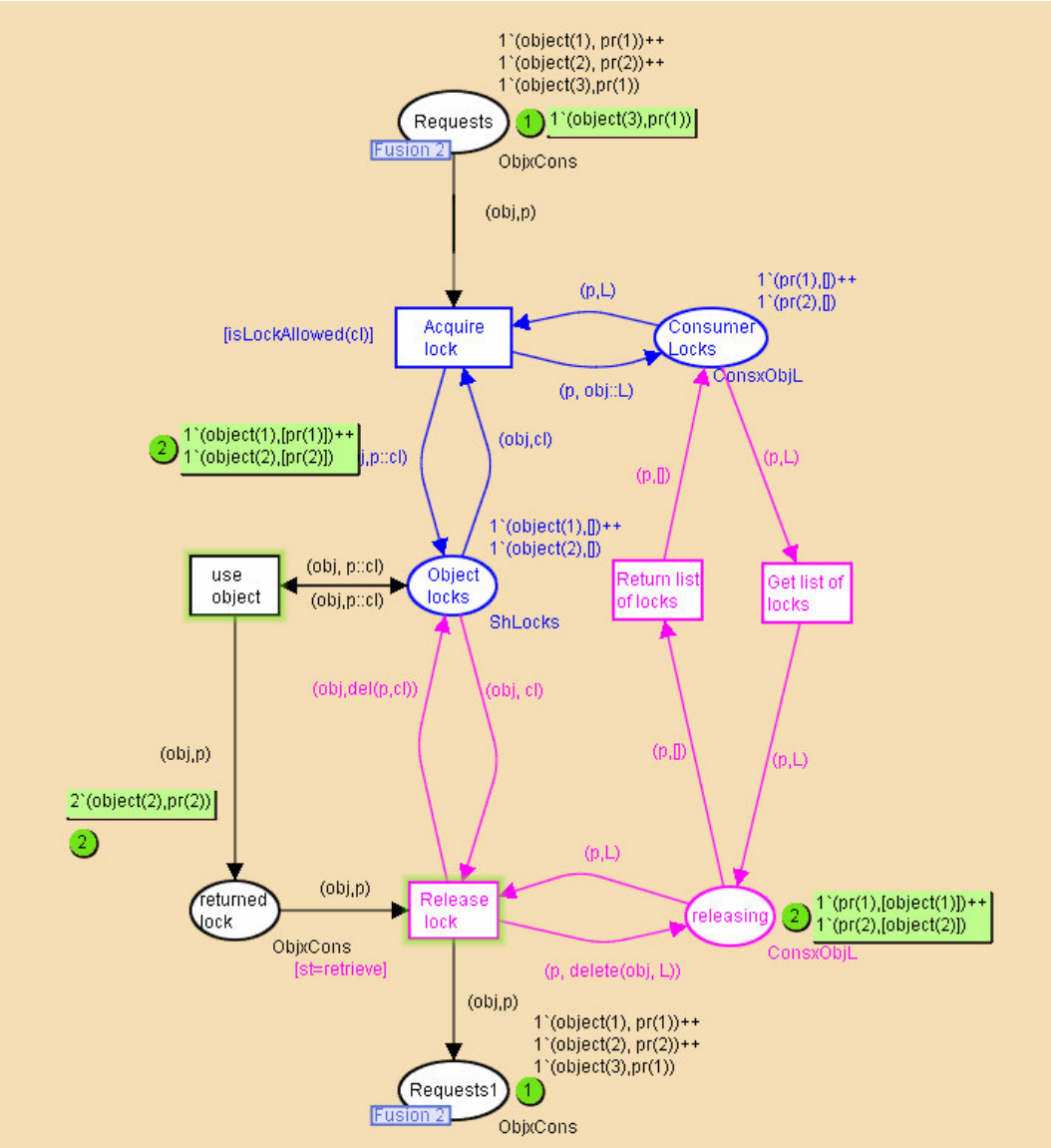


Figure 56 The lock manager (collective lock release)

Note that this pattern incorporates the ID MATCHING pattern in its solution for analyzing consumer requests. In order to keep track of locks this pattern aggregates them into a list, thus using the AGGREGATE OBJECTS pattern.

In case if an exclusive access to the shared data needs to be combined with a shared access, instead of this pattern use its extension, i.e. the BI-LOCK MANAGER pattern.

EXAMPLES:

- Account access by banks, credit card companies and insurance companies
- Critical section, i.e. a section of code that should be executed by only one processor at a time

RELATED PATTERNS: this pattern uses the AGGREGATE OBJECTS pattern and the ID MATCHING pattern in its solution. This pattern is extended by the BI-LOCK MANAGER pattern.

PATTERN 16: BI-LOCK MANAGER

ALSO KNOWN AS: READ/WRITE LOCKS

INTENT: to synchronize access to shared data for reading and writing purposes by means of shared and exclusive locks

MOTIVATION:

When several related processes execute concurrently, often they share some resources or data stored in a shared database. The SHARED DATABASE pattern allows multiple actors to access data stored in the shared database concurrently, but does not protect from overwriting each other's changes accidentally or reading inconsistent data due to in-progress changes. Often in practice, there is a need to provide shared and exclusive access to data stored in shared database, i.e. multiple actors should be able to read data concurrently, however only one actor must be able to modify the correspondent data at a time.

PROBLEM DESCRIPTION:

The LOCK MANAGER pattern solves the problem of synchronizing access to a shared resource by means of exclusive locks. No matter how an actor will use a data, i.e. read or modify, only one actor at a time may access the data. However, in some situations it is necessary to differentiate between two types of access, i.e. access for reading or access for writing, so that multiple actors can access an object for reading purposes, but for writing purposes only by one actor at a time.

SOLUTION:

In order to synchronize access to shared data, allowing both shared and exclusive access, use a *bi-lock manager*. The bi-lock manager, which is an extension of the LOCK MANAGER pattern, allows multiple actors access data for reading purposes and only one actor to modify data at a time. In order to get an access for reading, an actor must acquire a shared lock. In order to get an access for writing, an actor must acquire an exclusive lock.

Implementation of Solution:

The list of instruction below describes how to implements the BI-LOCK MANAGER pattern:

- Extend the mechanism of lock acquisition for the LOCK MANAGER pattern in the following way. Introduce two separate transitions for acquiring the shared and exclusive locks respectively. Connect these transitions to place containing requests for accessing an object posted by customer in form of (*object_id, type_of_access, consumer_id*).
- In order to make a choice between the types of requested lock, use the DETERMINISTIC XOR-SPLIT pattern. Add transition guards *[st=retrieve]* and *[st=modify]* to transitions *Acquire shared lock* and *Acquire excl lock* respectively.
- Connect both transitions to place *known locks,* which keeps track of objects locked by each of the customers so that every customer *p* has a list of *(object_id, type_of_access)* on which locks were granted.
- Introduce two separate places *shared locks* and *exclusive locks*. The former one will store for every object a list of consumers, who requested a read lock. The latter one will store only an identifier of a single consumer, who requested an exclusive access to the object.
- Introduce an arc from the *shared locks* to transition *Acquire excl locks* with an inscription (*obj, []*). This will ensure that an exclusive lock may be granted if no other consumers access the object. This means that no actor may modify data as long as another actor reads or modifies it.

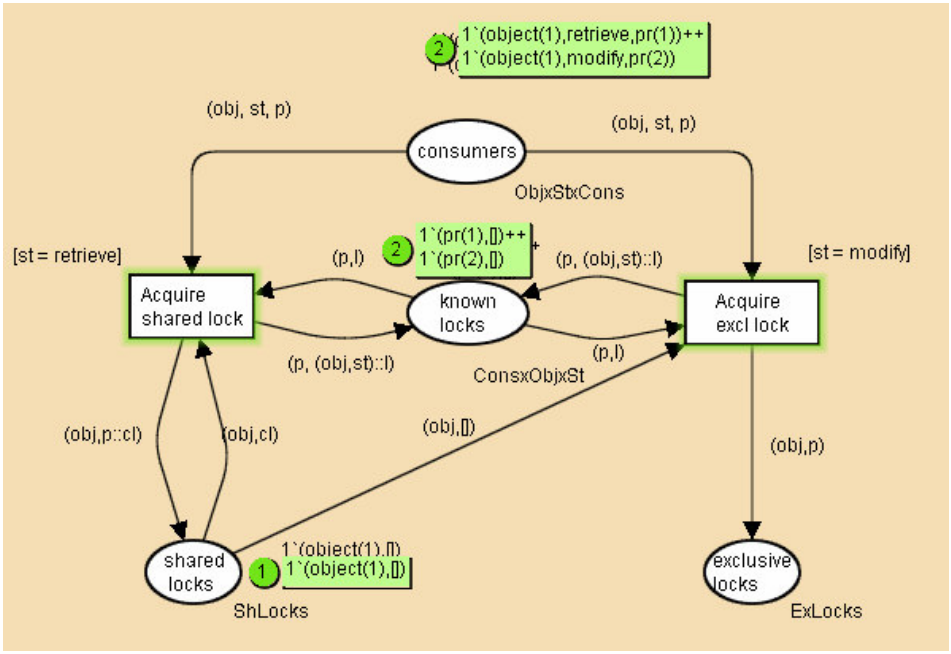


Figure 57 Bi-lock acquisition

Figure 58 shows how to incorporate the collective lock release, described in the LOCK MANAGER pattern, into the management of the shared and exclusive locks.
After an object, an exclusive lock on an object, which has been granted to a customer, becomes available, it is necessary to release the exclusive lock and enable both shared and exclusive access to this object. For this, add an arc from transition *release x lock* to place *shared locks*. (The rest of the functionality of the release mechanism is the same as described in the LOCK MANAGER pattern).

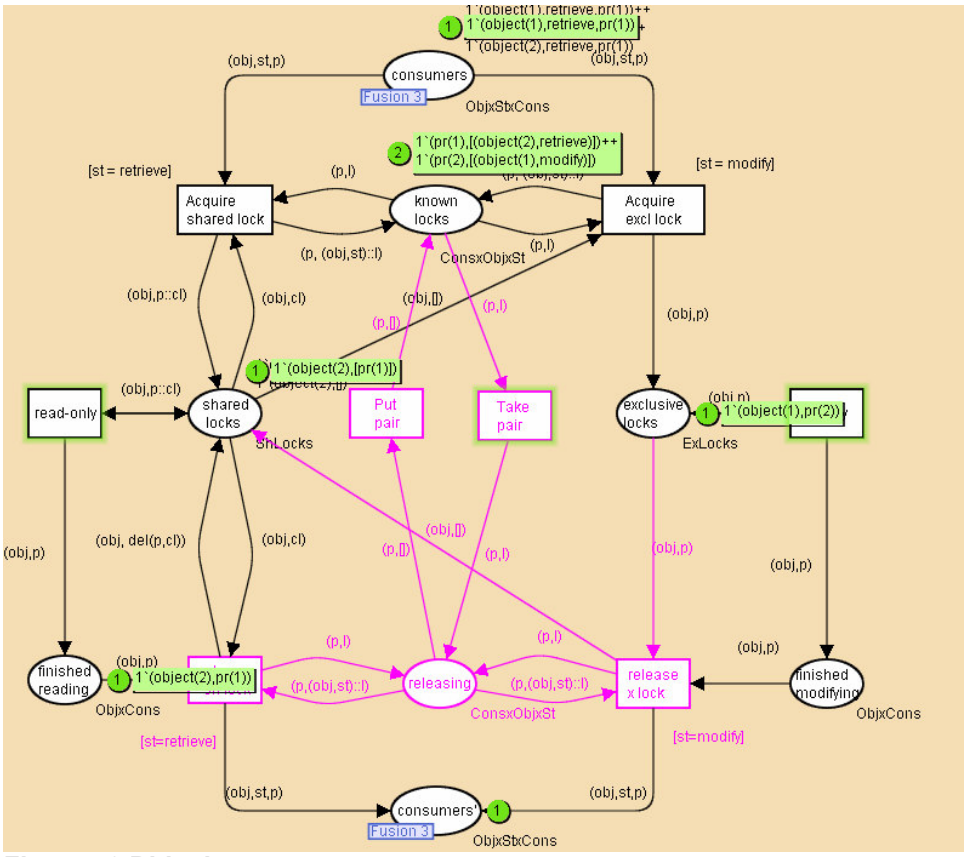


Figure 58 Bi-lock management

APPLICABILITY:
Apply this pattern in order to

- Ensure synchronization of shared and/or exclusive access to data shared between multiple actors, which need to read or modify the shared data respectively.

CONSEQUENCES:
The BI-LOCK MANAGER is an extension of the LOCK MANAGER pattern. This pattern can be applied in combination with a COPY MANAGER pattern, which replicates data stored in the central database into local copies, and local copies for data consistency purposes report their changes to the central database. To prevent overwriting the same data, the exclusive access is required, however for reading purposes the shared access is sufficient.

Similar to the LOCK MANAGER pattern, this pattern should be used in combination with shared databases described in the SHARED DATABASE pattern, in order to maintain consistency of data in the databases, while allowing multiple actors accessing it concurrently.

EXAMPLES:

- Account access by banks, credit card companies and insurance companies, which provide the shared access for reading but the exclusive access for writing.

RELATED PATTERNS: this pattern can be used in addition to the SHARED DATABASE and COPY MANAGER patterns. This pattern is an extension of the LOCK MANAGER pattern.

PATTERN 17: LOG MANAGER
ALSO KNOWN AS:

INTENT: to record the information about actual process execution by means of a data log

MOTIVATION:
In many situations, after executing a business process, a financial transaction, etc. it is necessary to record the information about actual process execution, i.e. what happened in between steps, what data was passed or modified, when and by whom. Based on such data it is possible to draw conclusions about the process efficiency, find out errors occurred or make data based analysis.

PROBLEM DESCRIPTION:
Consider a net presented in Figure 59. A chain of two transitions *Task A* and *Task B* are executed sequentially, i.e. one after another, processing data supplied from place *In* and storing the result in place *Out*. Although it is possible to gather the information about initial values of data provided by place *In* and final values stored in place *Out*, no information about actual process, i.e. which task modified which value and at what time, is available.

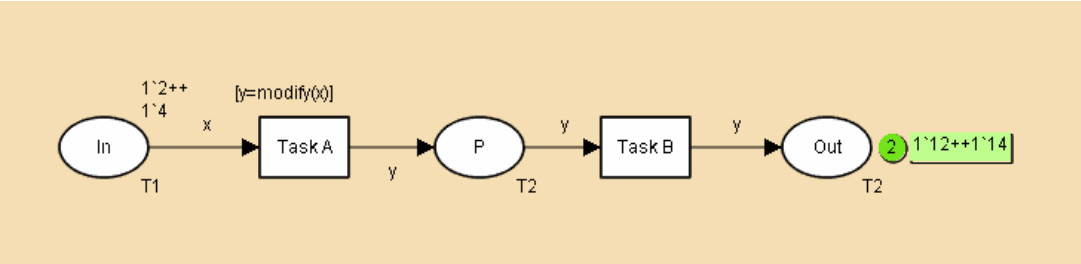


Figure 59

SOLUTION 1:
In order to record the information about actual process execution in a specific place, use an individual *log manager.* The individual log manager samples a transition, the information relevant to execution of which needs to be collected, and records this information into a separate log. Note that there is one-to-one correspondence between the sampled transition and the log place.

Implementation 1 of Solution:
Create a separate log for every transition, the information about execution of which needs to be recorded:

- Select a transition, the execution-related information of which needs to be recorded, (*Task A* in Figure 60).
- Create a new place *Log*, where the execution-relevant information will be stored. Define what data should be recorded, and correspondingly define the data format and type of place *Log.*

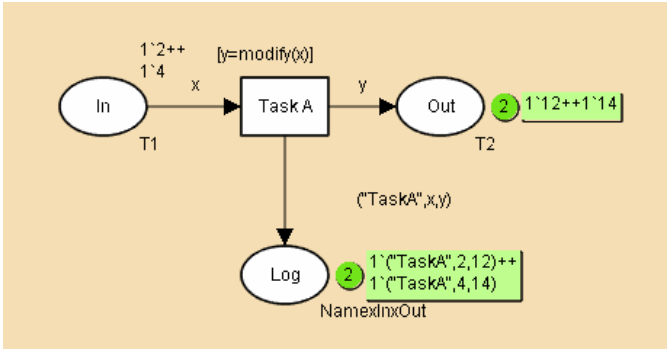


Figure 60

SOLUTION 2:
In order to record the information about actual process execution, use a collective *log manager.* The collective log manager samples every transition, the information relevant to execution of which needs to be collected, and records this information to a single log collection. Note that there is a many-to-one correspondence between the sampled transitions and the log place.

Implementation of Solution 2:
The list of instructions below describes how to implement Solution 2 of the LOG MANAGER pattern:

- Create a common place *Log* for logging the information related to execution of multiple transitions, and the AGGREGATE OBJECTS pattern to organize all logs into one collection. Note that all transitions must provide information in the same format selected for logging.
- In order to insert a data into the *Log*, take the current list of logs and add a new data to it. It is possible to sort data in the log, based either on the order of insertion or timestamp for instance. For this purpose, the AGGREGATE OBJECTS pattern can be combined with the QUEUE pattern or one of its variants.
- Note that if the time-relevant information is required for logging, in addition to a token value, each value may carry a time-stamp of type *Time.time*. The timestamp can be used as a parameter for sorting the data logs.

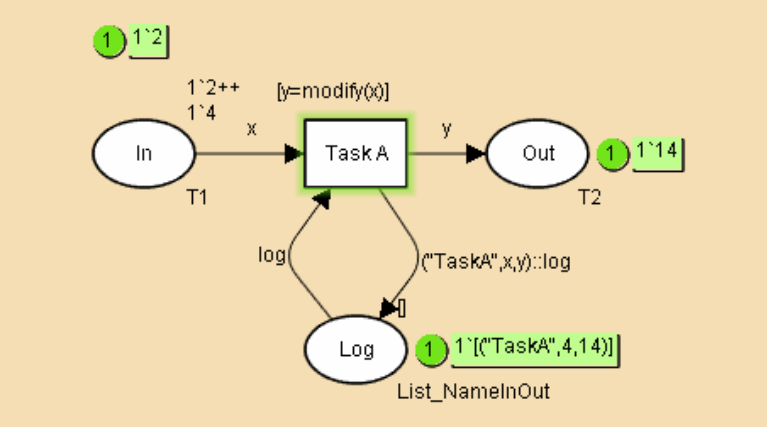


Figure 61

APPLICABILITY:
Apply this pattern to

- Record the execution-relevant information in order to get insight in intermediate steps of process execution for analysis purposes.

CONSEQUENCES:
The LOG MANAGER pattern provides two solutions for recording the information, based on which the flow of control and data in the modeled system/process can be analyzed.

Solution 1 is suitable for logging the information related to execution of a certain transition in a model. It allows to record different types of information for each specific transition. This solution can be used to test for instance critical points in a model. However, when the same type of information needs to be gathered from all transitions in a model, Solution 2 fits better. It centralizes data into one collection and allows sorting of the collection for structuring purposes.

EXAMPLES:

- Logging of the patients' state in the hospital on the daily basis for analysis of efficiency of treatment applied to a patient

RELATED PATTERNS: this pattern uses the AGGREGATE OBJECTS pattern in solution 2.

PATTERN 18: BLOCKING STATE-INDEPENDENT FILTER

ALSO KNOWN AS: BSI FILTER

INTENT: to prevent data, non-conforming to a certain property, from passing through

MOTIVATION:
In many different application examples, data intended for transferring from one source to another, or recording and processing afterwards, can be the scrambled, modified or flawed. Because of this, a target recipient of data instead of dealing with data of expected format, values and size, can be faced with incorrect, invalid or unknown data. Thus, there is a need in filtering incoming data depending on data-specific quality or characteristics.

PROBLEM DESCRIPTION:
Consider the mining of processes, which extracts workflow models from event-data logs. In many cases the data logs, which are provided for mining, include redundant information, i.e. repeating data, or so-called data noise, i.e. records of events, which are irrelevant to the process under mining. Mining of the process without distinguishing noise and redundancy in data logs can lead to discovering of the process, different from the desired one.

SOLUTION:
In order to prevent data, non-conforming to a certain set of properties associated with this data, from passing through, use a *state-independent filter*. The state-independent filter consumes data from the input place, analyses it against the set of specified conditions, and in case of properties fulfillment passes data to the output.

Implementation of Solution:
The list of instructions below describes how to implement a state-independent filter:

- Connect the place *In*, where non-filtered data is located to transition *Filter*.
- Specify the conditions for filtering in a guard of *Filter* transition. In the example in Figure 62 incoming integer data must be bounded from bottom by value 5, i.e. only data with values bigger then five will be passed to the output place *out.* Note that incoming data may be of any nature, i.e. lists, composite data structures, etc.
- Note that in this implementation alternative, the *Filter* has knowledge about the data bounds in which the incoming data should fit, and it does not handle the data, which do not fulfill the condition specified in the guard.

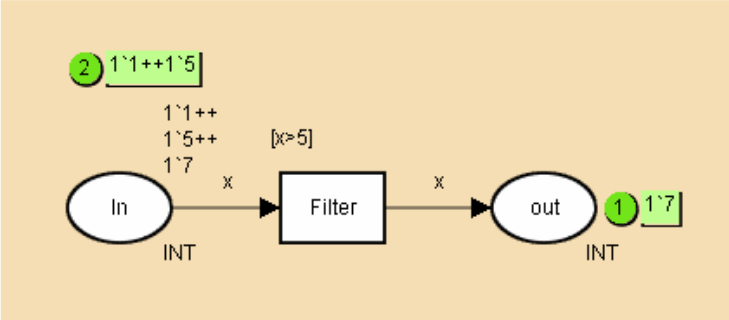


Figure 62 The blocking state-independent filter

APPLICABILITY:
Apply this pattern to

- Prevent transition of data non-compliant to a certain set of properties. This allows limiting the range, amount and types of incoming data based on the value of data itself and a state of an external data collection, if necessary.

CONSEQUENCES:
This pattern should be applied in situations, where filtering properties directly involve the value of incoming data. One of the characteristics of this pattern is a “blocking” feature, which allows only data fulfilling a certain property to pass through, but blocks the rest of data in the input place of the filter.

The drawback of this pattern is that it does not prevent from accumulation of objects, which did not fulfill the specified property in the input place of the filter. To address this problem, the DETERMINISTIC XOR-SPLIT pattern or an extension of this pattern, i.e. NON-BLOCKING STATE-INDEPENDENT FILTER, can be used instead.

In some situations, the filtering properties need to be based as on the value of incoming data as on the state of some externally available data structure. In this case, an extension of this pattern BLOCKING STATE-DEPENDENT FILTER can be applied instead.

EXAMPLES:

- From a stream of students only those ones who delivered homework are allowed to take a part in the exam

RELATED PATTERNS: this pattern is used in the DETERMINISTIC XOR-SPLIT pattern, and is extended by the BLOCKING STATE-DEPENDENT FILTER pattern and NON-BLOCKING STATE-INDEPENDENT FILTER pattern.

PATTERN 19: BLOCKING STATE-DEPENDENT FILTER

ALSO KNOWN AS: BSD FILTER

INTENT: to prevent data, non-conforming to a property, involving the state of an external data-structure, from passing through

MOTIVATION:
The BLOCKING STATE-INDEPENDENT FILTER pattern allows transitioning of data based on a property, directly involving the value of incoming data. In some situations, there is a need to incorporate a state of an external data structure into the filtering conditions. For instance, to allow transitioning of actors, identifiers of which are registered within a certain database, or to pass through only fresh data, which is not contained in a data collection. The state of such an external data structure may be static or variable, which changes over time.

PROBLEM DESCRIPTION:
For participating in a course, students must register in advance by submitting an application. Based on the received applications, a database of registered course participants is constructed. Both registered and non-registered students are allowed to attend the lectures. However, the certificates will be issued only to the registered students. From all requests on issuing the certificate, only the ones posted by the registered students must be handled.

SOLUTION:
In order to prevent data, non-conforming to a certain set of properties dependent on a state of an externally located data structure (i.e. list or multi-set), use a *blocking state-dependent filter*. The blocking state-dependent filter consumes data from an input place, checks the state of the external data collection, and in case of fulfillment the filtering conditions passes the data to the output.

Implementation 1 of Solution:
The list below describes how to implement the BLOCKING STATE-DEPENDENT FILTER:

- Extend implementation of Solution 1 by connecting transition *Filter* to an external data collection place *State*, contemporary state of which is used in the conditions for filtering. Note that the state of this collection is always persistent, i.e. there is always one token of collection type in it. Such place may be representation of a shared database, for instance.
- Add a guard to transition *Filter*, i.e. a function *check(l,x)*, which takes as inputs an input data x and external collection *l,* and examines the fulfillment of a certain property. For instance, the filtering condition used in the net in Figure 64 is satisfied if no duplicated data has been sent.
- Collection *l* records all values of data that were passed to place *out*, and function *check()* examines if an input data *x* is contained in the list *l*. Similar, one can specify filtering conditions concerning non-containment of elements in the collection or non-compliance to a certain criteria.
- Note that a state of the data collection can be static (Figure 65), i.e. do not change during the whole process execution, or be dynamic (Figure 64), and vary under influence of the control flow.

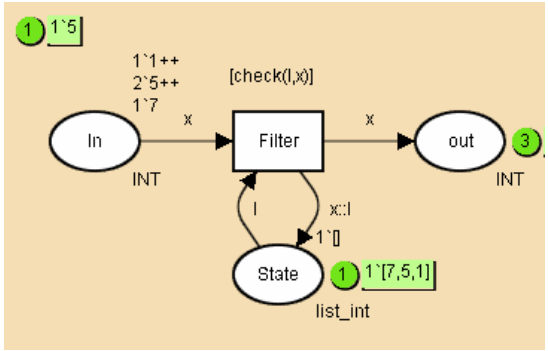


Figure 63

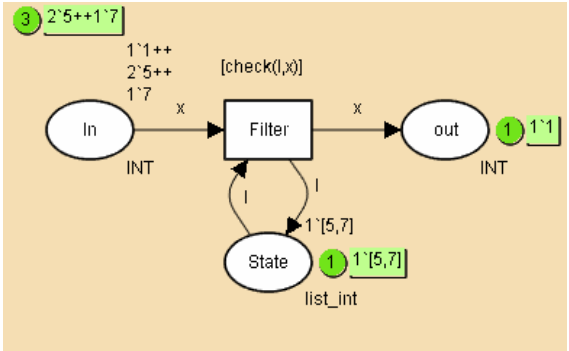


Figure 64

Implementation 2 of Solution:
This implementation alternative also incorporates the filter condition dependent on the state of external data. However, instead of using the state of data stored in a data collection, the state of data distinguished as a separate entity is used. Figure 65 presents the filter, which allows data non-contained in the external place *State* to pass through.

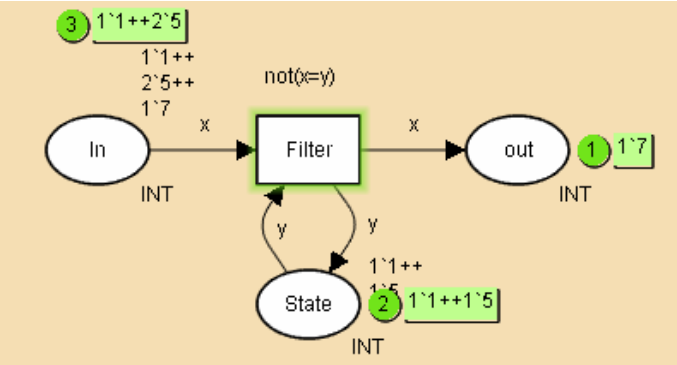


Figure 65

APPLICABILITY:
Apply this pattern to

- Prevent transition of data non-compliant to a set of properties, which involve the state of an external data-structure.

CONSEQUENCES:
This pattern is an extension of the BLOCKING STATE-INDEPENDENT FILTER. Similar to the latter, this pattern blocks the data, which does not fulfill a specified property, while not solving a problem of accumulation of this data in the input place of the filter. In order to address this problem, combine this pattern with the DETERMINISTIC XOR-SPLIT pattern or use the NON-BLOCKING STATE-DEPENDENT FILTER instead.

This pattern can be applied in combination with the AGGREGATE OBJECTS pattern, for incorporating the state of data in an external collection.

EXAMPLES:

- Assume that several hospitals are connected through a shared database of patients' records. For administrative purposes, patient statistics are kept in a master table that tracks data from all hospital. Nurses and doctors within each hospital need access to the patients table, but they need only the rows that contain the data of the patients in their hospital. To handle the needs of each hospital, a data filter that specifies the subset of data that each group can access is required.

RELATED PATTERNS: this pattern is an extension of the BLOCKING STATE-INDEPENDENT FILTER, it is extended by the DETERMINISTIC XOR-SPLIT pattern and NON-BLOCKING STATE-DEPENDENT FILTER pattern.

PATTERN 20: NON-BLOCKING STATE-INDEPENDENT FILTER

ALSO KNOWN AS: NBSI FILTER

INTENT: to filter out data fulfilling a certain property, while avoiding accumulation of non-conforming data in the filter input place

MOTIVATION:
The BLOCKING STATE-INDEPENDENT FILTER protects the data non-fulfilling a certain property from passing through. However, it does not handle the problem of accumulation of non-conforming data in the input place. In some situations, it is necessary to reroute or store somewhere else the data, which does not satisfy the filter property, rather then blocking and ignoring this data instead.

PROBLEM DESCRIPTION:
The production industry before delivering the items to its customers must perform a quality check. All produced items must be checked upon satisfaction the quality rules. Items which satisfy quality constraints will be supplied to the customers, while items of low-quality can be sent to a discount shop.

SOLUTION:
In order to prevent data, not fulfilling a certain set of properties associated with this data, from passing through, while avoiding an accumulation of non-conforming data in the input place, use a *non-blocking state-independent filter*. This filter consumes all data from the input place, analyses it against the set of the specified conditions, and passes it to an output for either data conforming or non-conforming the filter property.

Implementation of Solution:
The list of instructions below describes how to implement a non-blocking state-independent filter:

- Connect the place *In*, where non-filtered data is located, to transition *Filter*.
- Create two output places *Passed* and *Garbage*, where data conforming and non-conforming to the filter property will be placed correspondingly.
- Connect transition *Filter* to the output places. Specify on the output arcs mutually exclusive filtering conditions, i.e. the filtering conditions, which must be fulfilled by data in order to pass through the filter and negation of this condition in order to filter out the non-conforming data.
- Note that the obtained construct (Figure 66) is a specialization of the Solution 2 of the DETERMINISTIC XOR-SPLIT.

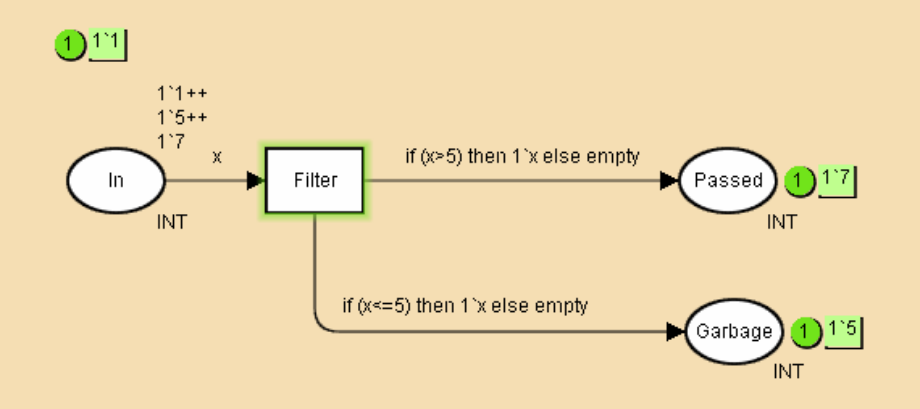


Figure 66 Non-blocking state-independent filter

APPLICABILITY:
Apply this pattern to

- Check data incoming to the filter against a set of the specified properties in order to filter out the non-conforming data, while avoiding over-accumulation of data non-conforming to the filter properties in the input place.

CONSEQUENCES:
In contrast to the BLOCKING STATE-INDEPENDENT FILTER, this pattern handles all incoming data allowing rerouting of data satisfying a filter property to one place, while all non-conforming data to another place. This filter works deterministically, i.e. it has knowledge about all possible data values and can be considered as a specialization of Solution 2 of the DETERMINISTIC XOR-SPLIT pattern.

EXAMPLES:

- Patients in the hospital are analyzed against the health problems they experience and are divided between the health specialists based on the characteristics of their health problems.

RELATED PATTERNS: this pattern is an extension of the BLOCKING STATE-INDEPENDENT FILTER and can be considered as a specialization of the Solution 2 of the DETERMINISTIC XOR-SPLIT pattern.

PATTERN 21: NON-BLOCKING STATE-DEPENDENT FILTER

ALSO KNOWN AS: NBSD FILTER

INTENT: to filter-out data non-conforming to a property, involving the state of an external data-structure, while avoiding accumulation of non-conforming data in the filter input

MOTIVATION:
The NON-BLOCKING STATE-INDEPENDENT FILTER protects the data non-fulfilling a property involving the state of externally available data from passing through. However, it does not handle the problem of accumulation of non-conforming data in the input place. In some situations, it is necessary to reroute or store somewhere else data, which does not satisfy the filter property, rather then blocking and ignoring this data instead.

PROBLEM DESCRIPTION:
The driving school handles requests of the applicants for participation in the theory-driving exam. A school officer checks in the database if an applicant subscribed for an exam. The subscribed applicants are rerouted to the exam room, while applicants without subscription are placed in the waiting queue. Requests of all applicants should be handled and none of them should be ignored.

SOLUTION:
In order to filter-out data, non-conforming to a certain set of properties, involving the state of an externally located data structure (i.e. list or multi-set), while avoiding accumulation of non-conforming data in the input place, use a *non-blocking state-dependent filter*. This filter consumes data from the input place, checks the state of external data collection, and in case of fulfillment the filtering conditions passes the data to one output, while in case of non-compliance to filtering conditions to other output.

Implementation of Solution:
The list below describes how to implement a NON-BLOCKING STATE-DEPENDENT FILTER:

- Extend the solution of the BLOCKING STATE-INDEPENDENT FILTER in the following way. Introduce two output places for transition *Filter* where data conforming and non-conforming the filter properties will be stored correspondingly.
- Place filtering condition on the corresponding output arcs, i.e. a condition which must be fulfilled for passing through the filter, and negation of this condition for filtering out non-conforming data.
- The net in Figure 67 incoming data if it not contained in the external data collection is passed through the filter, while repeating data, which is already contained in the collection, is filtered out.

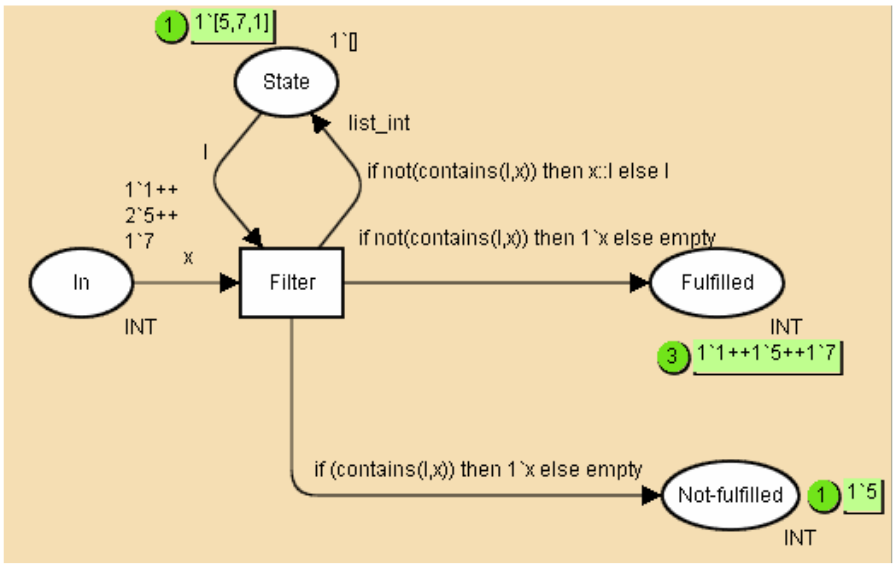


Figure 67 Non-blocking state-dependent filter

APPLICABILITY:
Apply this pattern to

- Check data incoming to the filter against a set of the specified properties in order to filter out the non-conforming data, while avoiding over-accumulation of data non-conforming to the filter properties in the input place.

CONSEQUENCES:
This pattern checks whether incoming data satisfies a filtering property depending on the state of an external database or a data collection. In case of property fulfillment the filter routes data to one place, while in case of non-compliance to another place. This pattern ensures that all incoming data are handled, thus avoiding accumulation of non-conforming data in the filter input place.

This pattern can be considered as an extension of the BLOCKING STATE-DEPENDENT FILTER pattern by means of applying Solution 2 of the DETERMINISTIC XOR-SPLIT pattern.

EXAMPLES:

- Handling of insurance claims, where all incoming claims should be analyzed and either worked out in a detail or rejected.

RELATED PATTERNS: this pattern extends the BLOCKING STATE-DEPENDENT FILTER pattern by using Solution 2 of the DETERMINISTIC XOR-SPLIT pattern.

PATTERN 22: TRANSLATOR

ALSO KNOWN AS: DATA MARSHALING

INTENT: to enable coordinated communication between two actors with originally different data formats

MOTIVATION:
When two actors (i.e. processes, applications, etc.) need to communicate to each other by means of data transfers, the information sent by one actor may be misinterpreted or denied due to difference in data formats. For successful data exchange, a common data format must be used.

PROBLEM DESCRIPTION:
For coordinated communication between two actors with originally different data formats (data types), the actors need to use a common data format. Changing the nature of an actor to support a unified type can be difficult or impossible.

SOLUTION 1:
In order to coordinate data exchange between two tightly coupled actors with originally distinct data formats, translate the format of data sender to the format of data receiver.

Implementation of Solution 1:
The list of instructions below describes how to implement Solution1 of the TRANSLATOR pattern:

- When sending a data from one actor to another actor, which has a different data format, keep the original data format of the sender. However, before sending the data, translate the format of the sender to the format of the receiver. For this, connect the sender and receiver by means of transition, specifying by connecting arcs the direction of data flow. Add the transition guard a function *translate(),* which takes as an input the data of the sender, and outputs the same data in the adjusted format.
- Such translation of data formats is possible only under the assumption that the knowledge about the data formats of sender and receiver is available. Note that the content of the function *translate()* is variable, and can be modified if the format of the receiver changes.
- In the example in Figure 68, an *Actor1* works with data of the following format (*data_name, data_value*). Assume that the *Actor2* requires the data in the following format (*header, data_name, data_value)*. Therefore, function *translate()* adds a header to the data provided by the *Actor1,* and only afterwards, transition *Send* supplies data to the *Actor 2*.
- The net in Figure 68 presents translation of all data supplied by *Actor1* to *Actor2*. Note that direct translation can be also done upon a request, so that not all available data is being translated, but only the data, request for translation of which has been obtained. In addition, there can be some differences in the moment of translation. In particular, the translation can be done on the moment of production or consumption.

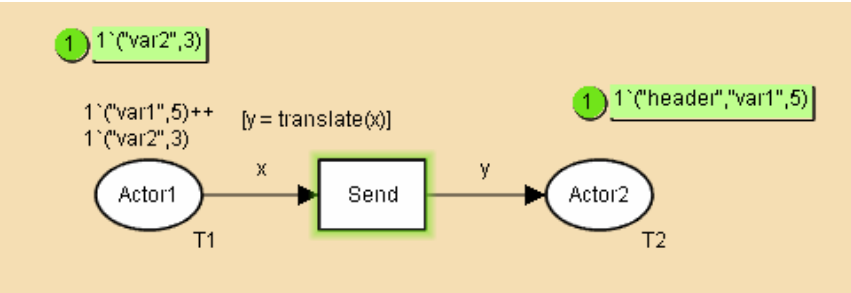


Figure 68

SOLUTION 2:
To coordinate communication between multiple actors with distinct data formats, while ensuring that the actors are loosely coupled and are not aware of details of the data formats of each other, translate all data formats to the common format, known to every actor.

Implementation of Solution 2:
The list of instructions below describes how to implement Solution 2 of the TRANSLATOR pattern:

- Implementation of this solution involves the ASYNCHRONOUS ROUTER, and ID MATCHING patterns.
- The ASYNCHRONOUS ROUTER pattern ensures loose coupling between data senders and receivers. Data sent by actors-senders is initially translated to the unified format and then placed to *Common* place, where it stays until an actor-receiver does not pick it up.
- Since data supplied by multiple senders is stored in *Common* place, the ID MATCHING pattern is applied to ensure that an actor-receiver does not consume data addressed to another actor.
- For each actor, which sends and receives data, two translation functions must be introduced: translation of the actor data format to the common data format when sending data, and translation of the common data format to the actor data format when receiving data. Translation functions are incorporated into the transition guards.

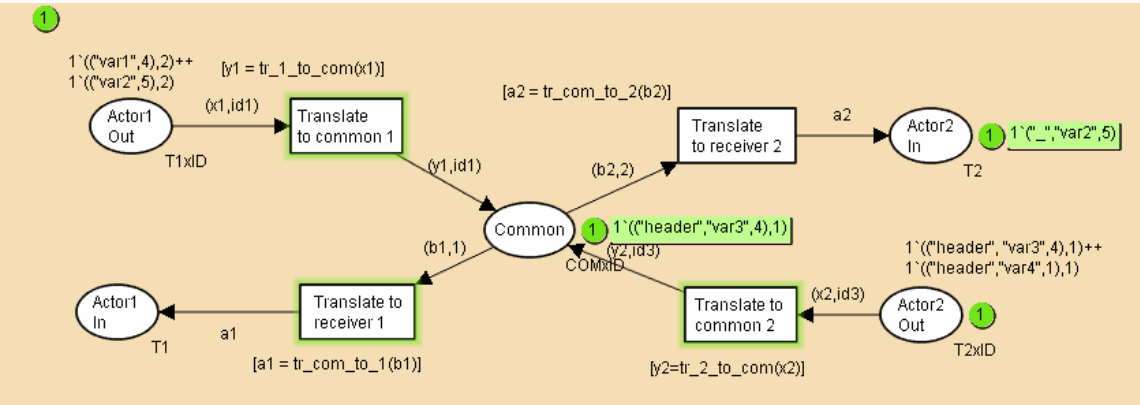


Figure 69

APPLICABILITY:
Apply this pattern to

- Establish communication between two parties, the format of data of which differs from each other.
- Exchange data between two tightly-coupled actors.
- Exchange data between multiple loosely-coupled actors.

CONSEQUENCES:
The TRANSLATOR pattern allows establishing data exchange between multiple actors, providing compatibility of initially distinct data formats.

When multiple actors, each with different data types, are involved into data exchange, it might be necessary to marshal data from every sender to every receiver. This means that for every two actors with distinct data formats it is necessary to write two functions, translating one format to another back and forward, if these two actors are directly connected to each other, or writing two functions for every actor to translate its data format to a common format and backwards, if communicating actors are loosely coupled.

EXAMPLES:

- In order to post the request for the financial grand, the sender must package all information in a form requested by the financial committee.
- Two different types of processors, i.e. one processor working in the "Big Endian" format, and another in "Little Endian" format, need to establish a communication channel. Since their formats are mirror reflection of each other, i.e. 001 and 100, the format of data sender should be inverted.

RELATED PATTERNS: this pattern uses ID MATCHING and ASYNCHRONOUS ROUTER patterns.

PATTERN 23: ASYNCHRONOUS TRANSFER
ALSO KNOWN AS:

INTENT: to allow transportation of data from one location to another, while avoiding the sender to block

MOTIVATION:
In a distributed environment, several processes may proceed independently of each other until one process needs to interrupt the other process by transferring data to it. Although the processes are related to each other through a data channel, they should stay independent.

PROBLEM DESCRIPTION:
Assume that there are two processes running independently as Figure 70 illustrates. *Process 1* produces data for transferring to another process. *Process 2* processes data received from another process. Waiting for acceptance of the data by *Process 2* may cause blocking of *Process 1* and postpone preparation of data for the next data transfer.

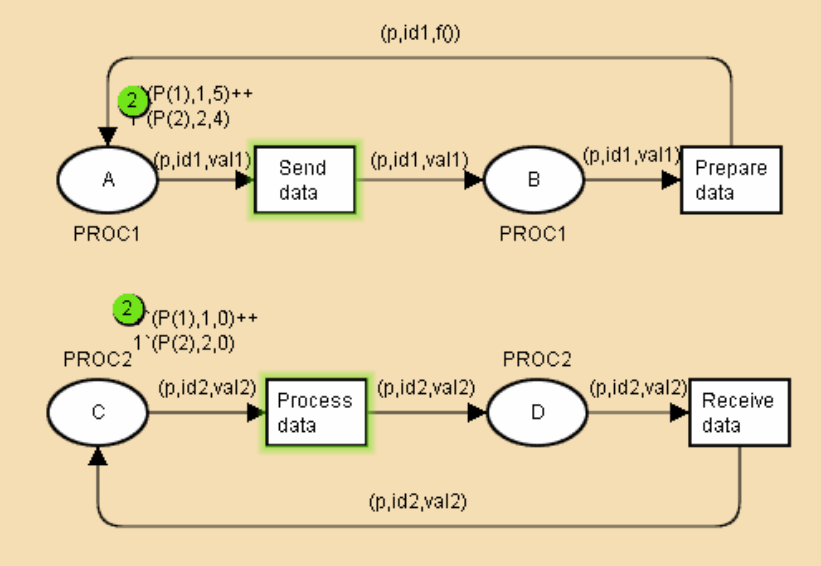


Figure 70

SOLUTION:
In order to send data from one place to another, while avoiding blocking of the sender, use an *asynchronous transfer.* The asynchronous transfer is established through a placeholder, which stores data arriving from the sender until the receiver picks it up.

Implementation of Solution:
The list of instructions below describes how to implement the ASYNCHRONOUS TRANSFER pattern:

- Introduce a new place *Request*, which will serve as a placeholder for data asynchronously transferred from transition *Send* to transition *Receive*. The format of data in the placeholder must incorporate data identifiers, known both to the sender and to the receiver.

- Transition *Send* puts the (*data_id,* data_value) into the placeholder *Request*. Transition *Ackn* uses the ID MATCHING pattern to withdraw a value of the data from the placeholder.

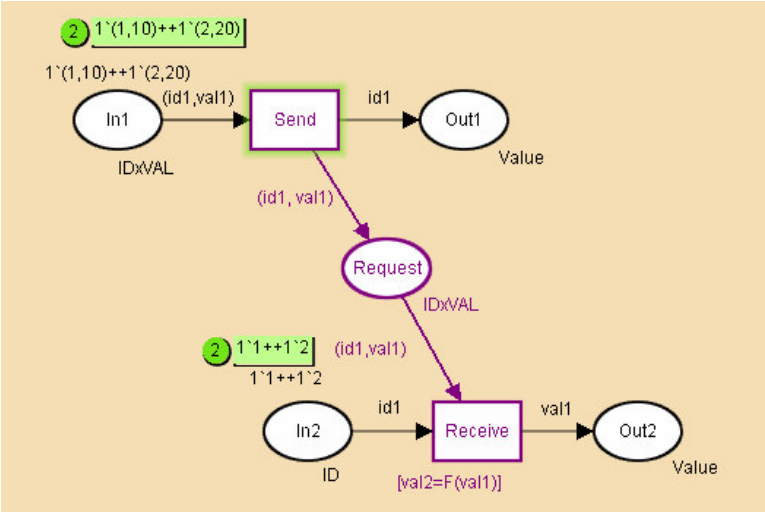


Figure 71 The Asynchronous Transfer

Figure 72 demonstrates how to incorporate the ASYNCHRONOUS TRANSFER pattern into the net presented in the *Problem description* section.

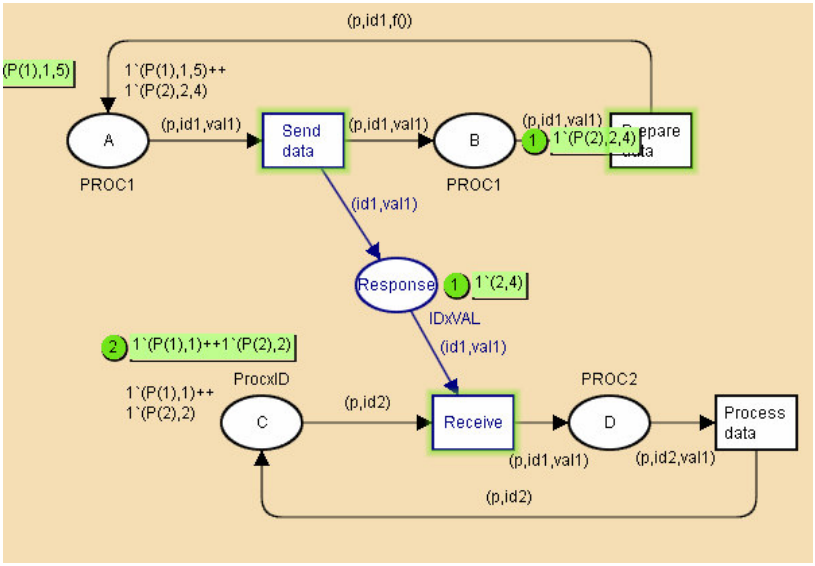


Figure 72

APPLICABILITY:
Apply this pattern to

- Establish communication channel between multiple parties, when the sender of data does not need an instant response on receipt of the data. This promotes independency of parties and avoids blocking of the sender.

CONSEQUENCES:
Application of the ASYNCHRONOUS TRANSFER pattern allows transferring data from one place to another, providing tight coupling between a sender and a receiver, while allowing them to work independently.

When one data source needs to transfer data to multiple targets asynchronously, the tight coupling between the sender and receiver introduced by the ASYNCHRONOUS TRANSFER pattern may be undesirable. In such case, instead of this pattern the ASYNCHRONOUS ROUTER pattern should be applied, which decouples a sender from receivers, thus making them loosely coupled.

An advantage of this pattern that it does not block the sender, and does not require a sender and a receiver always to be connected through a channel, since data sent can be queued.

This pattern is similar to the SYNCHRONOUS TRANSFER pattern and the RENDEZVOUS pattern, because they address similar kind of problems in the context of data transportation in the distributed environment.

EXAMPLES:

- An employee needs to send a letter. He does not wait until a mail carrier arrives to pick it up, but puts it in the mailbox. The mail carrier will pick it up on his own initiative.
- Participants, who are online at different times, use Web message boards, newsgroups, or e-mail, thus interacting asynchronously.

RELATED PATTERNS: this pattern is used in ASYNCHRONOUS ROUTER pattern. This pattern is similar to the SYNCHRONOUS TRANSFER pattern and the RENDEZVOUS pattern

PATTERN 24: SYNCHRONOUS TRANSFER

ALSO KNOWN AS: REQUEST/ REPLY, PING-PONG

INTENT: to allow transportation of data from one location to another, ensuring that an actor, which posted a request, is blocked until it does not receive the requested information

MOTIVATION:
In a distributed environment, several processes may proceed independently of each other until one process needs to interrupt the other process by transferring data to it. Data sender needs to get an instant respond on the data sent, is not allowed to proceed until an answer from the data receiver is obtained. Such request/respond communication is used, for example, by people gathering at the same time for chatting or instant messaging.

PROBLEM DESCRIPTION:
Assume that there are two processes running independently (Figure 73). *Process 1* produces data for transferring to another process. *Process 2* requires data produced by *Process 1* in its calculations. *Process 1* will provide data only if the request on transmission of this data from another process in received.

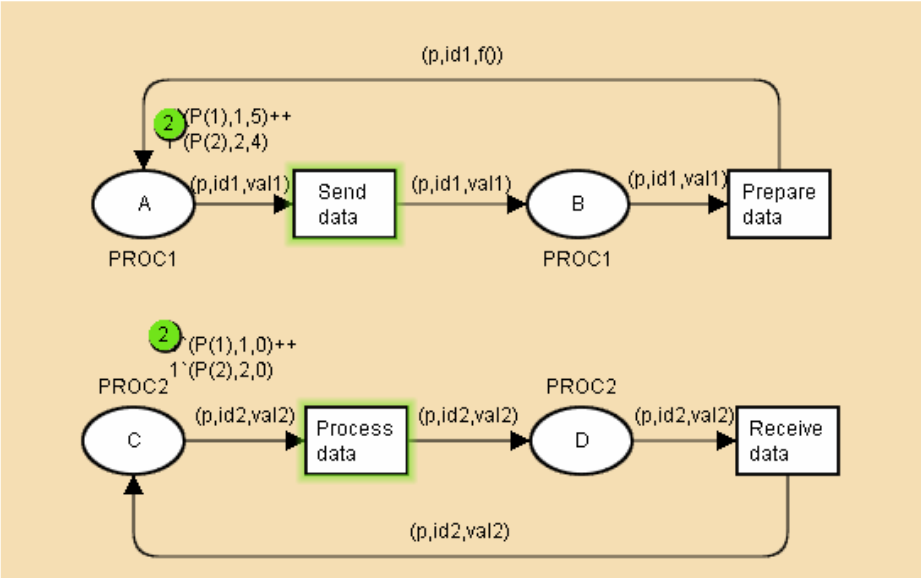


Figure 73

SOLUTION:
In order to allow transportation of data from one location to another, ensuring that the sender is blocked until it receives an answer from the receiver, use a *synchronous transfer.* The synchronous transfer is established through two placeholders, which temporally store data requests and responds. The sender is blocked until the requested data becomes available in the correspondent placeholder.

Implementation of Solution:
The list of instructions below describes how to implement the SYNCHRONOUS TRANSFER pattern:

- Define the transition, which initiates the data exchange. Break this transition into two parts: for sending the data request and receiving the requested data, i.e. *Send* and *Receive.*
- Connect transitions *Send* and *Receive* through a waiting place, where the initiator of the data exchange will reside until the requested data becomes available.
- Introduce place *Request* where the requests posted by the initiator of the data exchange will be stored until the receiver processes the request.
- Introduce place *Respond* into which the requested data will be placed.
- Connect transition *Send* to transition *Ackn* through placeholder *Requests*, indicating the flow of data (requests) by direction of the arrows.

- Similar, connect transition *Ackn* to transition *Receive* through placeholder *Respond,* indicating the flow of data (responds) by direction of arrows.
- Note that in such synchronous communication the exchange of data is possible only if the sender has knowledge about data available at the receiver side. For reference to the specific data, the identifiers associated with this data are used (see ID MATCHING pattern).

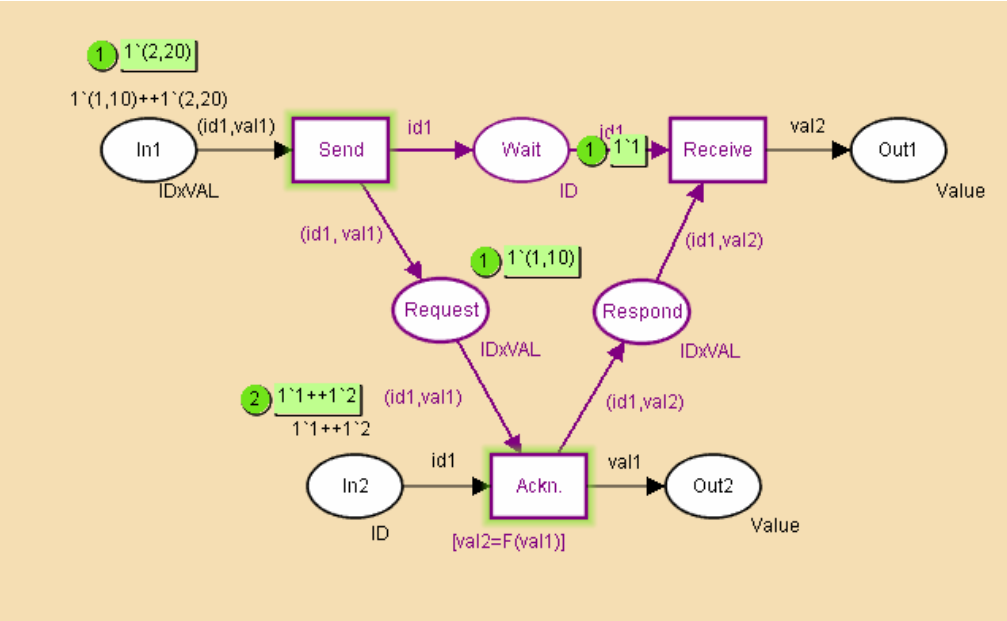


Figure 74 The Synchronous Transfer

Figure 75 demonstrates how to incorporate synchronous data transfer into the net presented in the *Problem description* section.

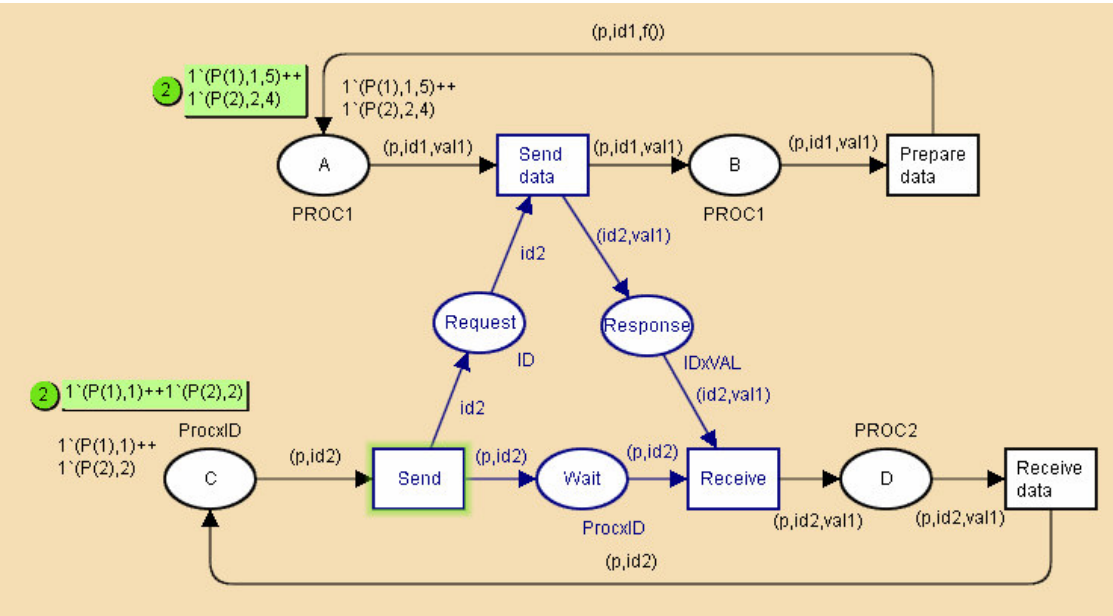


Figure 75

APPLICABILITY:
Apply this pattern to

- Provide data produced by one actor to another actor, which sent a data request, ensuring that the initiator of communication, i.e. the one that posted a data request, is blocked and will be resumed only when an answer on the data request becomes available.

CONSEQUENCES:
This pattern can be applied in situations when one actor (for instance, an application) sends a message and expects to receive the message back.

This pattern is similar to the ASYNCHRONOUS TRANSFER pattern and the RENDEZVOUS pattern, because they address similar kind of problems in the context of data transportation in the distributed environment.

In comparison to the ASYNCHRONOUS TRANSFER, the disadvantage of this pattern is that a sender is blocked until a receiver finished processing. Similar to RENDEZVOUS, this pattern needs to have synchronization between a sender and a receiver, however in this pattern it is done sequentially, while in the RENDEZVOUS pattern concurrently.

EXAMPLES:

- Any kinds of meetings, where an instant input from participants of the discussion is expected
- Subroutine calls from a program on one machine to library routines on another machine

RELATED PATTERNS: this pattern uses ID MATCHING pattern. This pattern is similar to the ASYNCHRONOUS TRANSFER pattern and the RENDEZVOUS pattern.

PATTERN 25: RENDEZVOUS
ALSO KNOWN AS:

INTENT: to allow multiple actors to broadcast and discover data objects concurrently

MOTIVATION:
In some situations, it is necessary to model a channel, which only transfers data messages but does not store them, allowing sending and receiving of the messages at the same time.

PROBLEM DESCRIPTION:
Assume that there are two processes running independently (Figure 76). *Process 1* produces data for transferring to *Process 2*, based on the data provided by *Process 2*. *Process 2* correspondingly processes data received from *Process 1* and sends the results of processing back to the *Process 1*. In order to avoid unnecessary waiting, both *Process 1* and *Process 2* need to be able to exchange, i.e. send and receive, data concurrently.

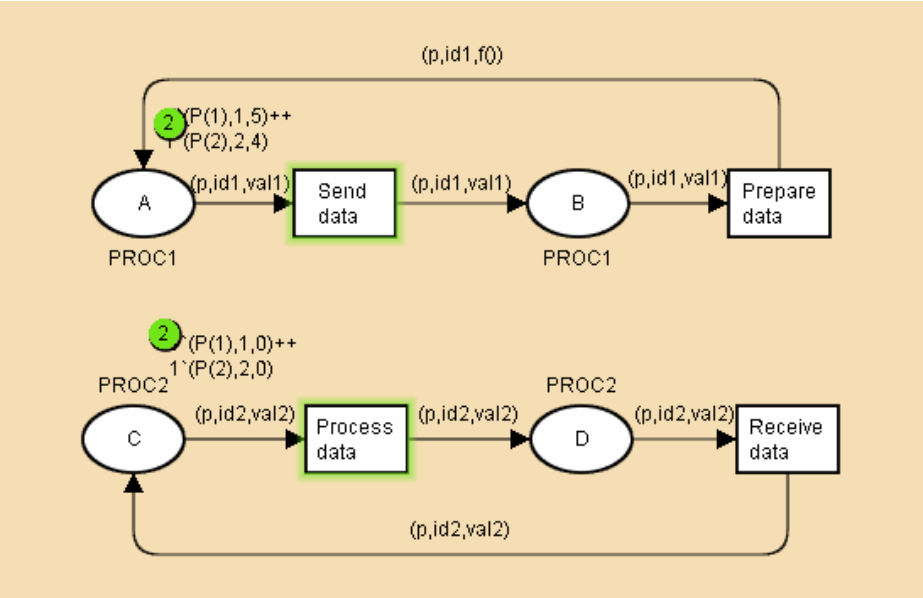


Figure 76

SOLUTION:
In order to allow concurrent exchange of data between multiple actors, use *rendezvous*. The rendezvous establishes concurrent exchange of data between multiple actors by connecting senders and receivers to a single transition, which will discover the data available for sending and broadcast it to the correspondent recipient(s).

Implementation of Solution:
The list of instructions below describes how to implement the RENDEZVOUS pattern:

- For each of the actors, participating in concurrent data exchange, define input and output places. The input places *In1* and *In2* provide data for broadcasting to the recipients, and the output places *Out1* and *Out2* store the data received from a sender.
- Connect input places to the transition *Exchange,* and connect transition *Exchange* to the output places. As soon as both senders provide the data for broadcasting, transition *Exchange* will fire putting the correspondent data to the output places.

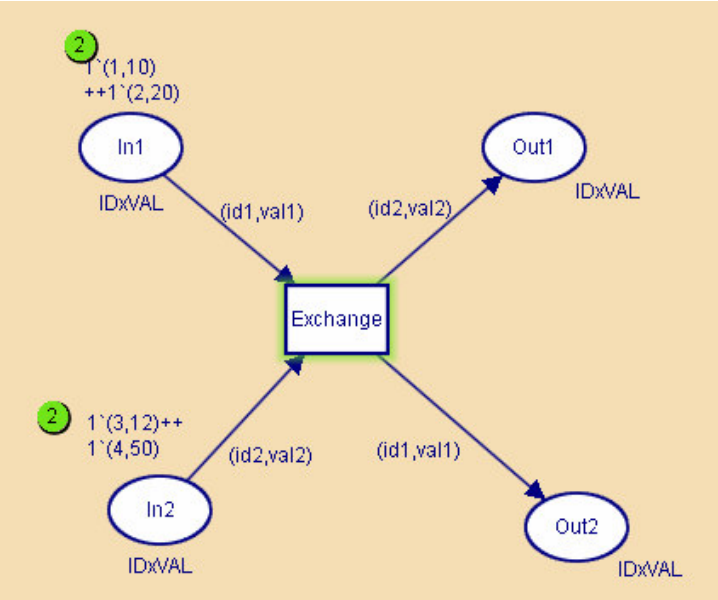


Figure 77 The Rendezvous

Figure 78 illustrates how to implement concurrent data exchange in the example presented in the Problem description section. Note that transitions *Send data* and *Process data* are merged into one transition *Send data*, which synchronizes concurrent communication of the process 1 and process 2.

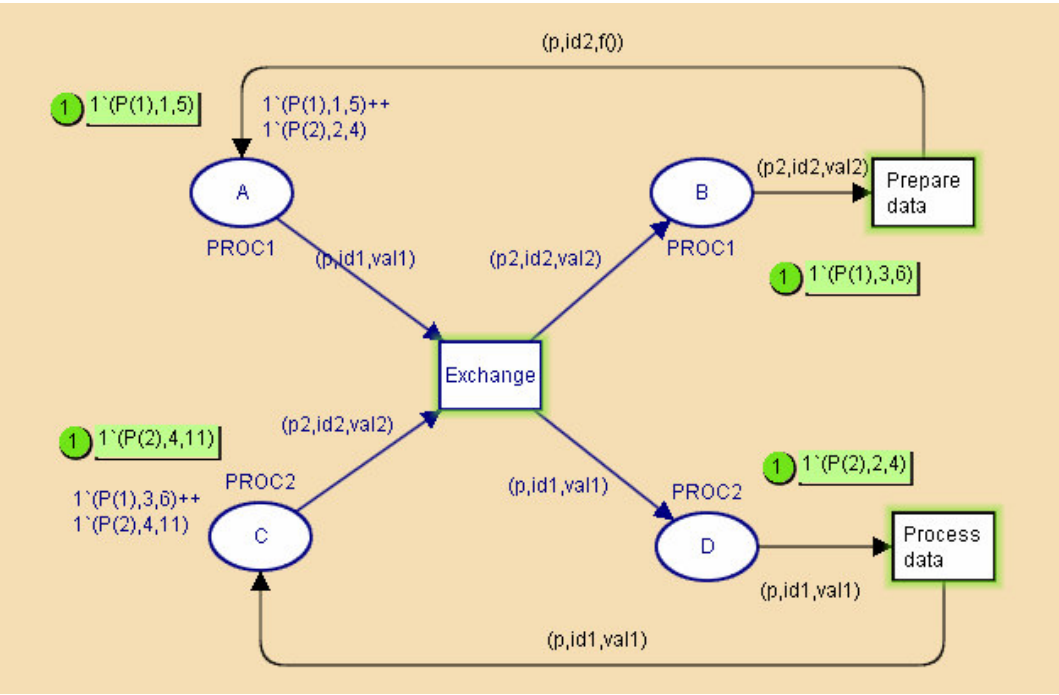


Figure 78

APPLICABILITY:
Apply this pattern to

- Facilitate concurrent exchange of data between two or more actors.

CONSEQUENCES:
This pattern is similar to the SYNCHRONOUS TRANSFER pattern and the ASYNCHRONOUS TRANSFER pattern, because they address similar kind of problems in the context of data transportation in the distributed environment. Although an advantage of this pattern over the SYNCHRONOUS TRANSFER that it more efficient, the data producers and consumers are tightly coupled and must run simultaneously for data delivery to occur.

The RENDEZVOUS pattern can be applied for broadcasting the same data to multiple recipients. The disadvantage of this pattern is that a sender and receivers are tightly coupled to each other. If it is necessary to avoid dependency between sender and receivers, but to be able to broadcast the same data to every recipient, instead of this pattern the BROADCASTING pattern can be applied. The drawback of using the BROADCASTING pattern is that it is based on the ASYNCHRONOUS TRANSFER and does not ensure that targets will get data simultaneously.

EXAMPLES:

- two or more objects meeting at a preconceived time and place

RELATED PATTERNS: this pattern is similar to the SYNCHRONOUS TRANSFER, ASYNCHRONOUS TRANSFER and BROADCASTING patterns.

PATTERN 26: ASYNCHRONOUS ROUTER
ALSO KNOWN AS:

INTENT: to enable asynchronous transfer of data from a single source to a dedicated target, providing loose coupling between the source and targets connected to it

MOTIVATION:
In some situations, there is a need to transfer data from a source to any of available targets asynchronously. For this purpose the ASYNCHRONOUS TRANSFER pattern can be applied, which allows the source and targets working independently, however requires tight coupling between them. This means that the source delivers data directly to the target, while requiring the data source to know all details of target recipients. Due to the tight coupling between a source and targets, changes in any of the targets may directly affect the source, thus providing low flexibility in targets manipulating.

PROBLEM DESCRIPTION:
Figure 79 presents the problem of direct addressing data messages between source and targets. Transition *source* when sending a message *mes* knows exactly who will be the receivers of the message, i.e. *target1, target2* or *target3* respectively. Meanwhile targets have no knowledge about how the source selects a target and what information it uses for this purpose.

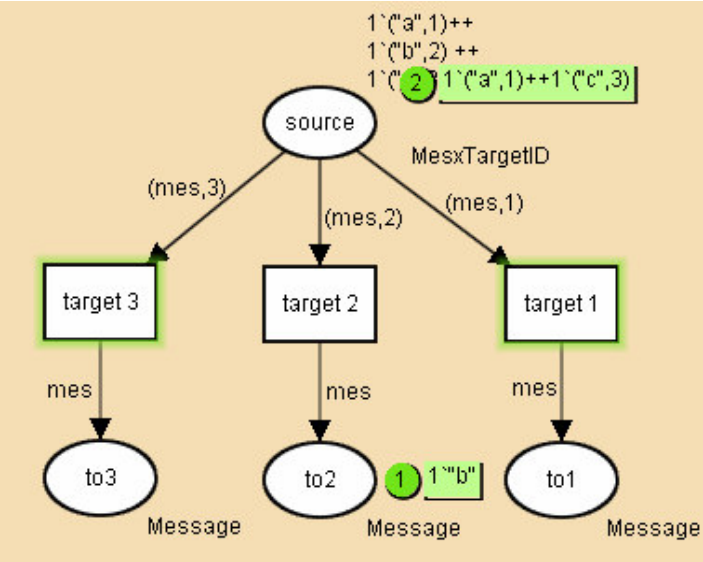


Figure 79

For instance, adding of a new target, the information about which (for instance, the target address) is not known yet, is not possible in this diagram. In addition, changes in any of the targets directly affect the *source*, and may even influence the connection between the source and the rest of the targets.

SOLUTION 1:
In order to decouple connection between a data source and a set of targets, which communicate asynchronously, while ensuring that data sent by the source is received by a target, to which it was dedicated, introduce an *asynchronous router*. The asynchronous router will direct all data received from the source to a place, where it will be stored until the dedicated target picks it up on its own initiative.

Implementation of Solution 1:
The list of instructions below describes how to implement Solution 1 of the ASYNCHRONOUS ROUTER pattern:

- Introduce transition *Route,* and connect place *source* to it, providing the composite data *(mes,t)*, which includes an information object to be sent to a target and an identifier of the target.

- Introduce place *temp*, which will serve as a temporary storage and will store all data *(mes,t)* sent by the source.
- Connect place *temp* to the targets. Provide the data to a target, by drawing an arc from the router to a target. Note that data from the *source* to the *temp* is sent asynchronously, as the ASYNCHRONOUS TRANSFER pattern describes.
- Add a transition guard to each of the targets to ensure that a target gets data addressed specifically to it. For this, target itself must have an identifier and should be aware of it. In Figure 80 target identifiers are encapsulated into the variables *t1, t2,* and *t3.* The transitions guards examine the second element of the data available in *temp*, i.e. the destination of the data, and compare it with own identifier (the ID MATCHING pattern). Since targets identifiers are different, no two targets can consume the same data (the pattern DETERMINISTIC XOR-SPLIT is used) at once.
- Note that the order in which data, which is routed from the *source* to *temp,* consumed by a *target* transition is non-deterministic. In order to ensure that targets consume data in the order of arrival, the data in place *temp* can be aggregated into a collection by applying the AGGREGATE OBJECTS pattern, while manipulating data in a strictly specified order can enabled by applying the QUEUE pattern.

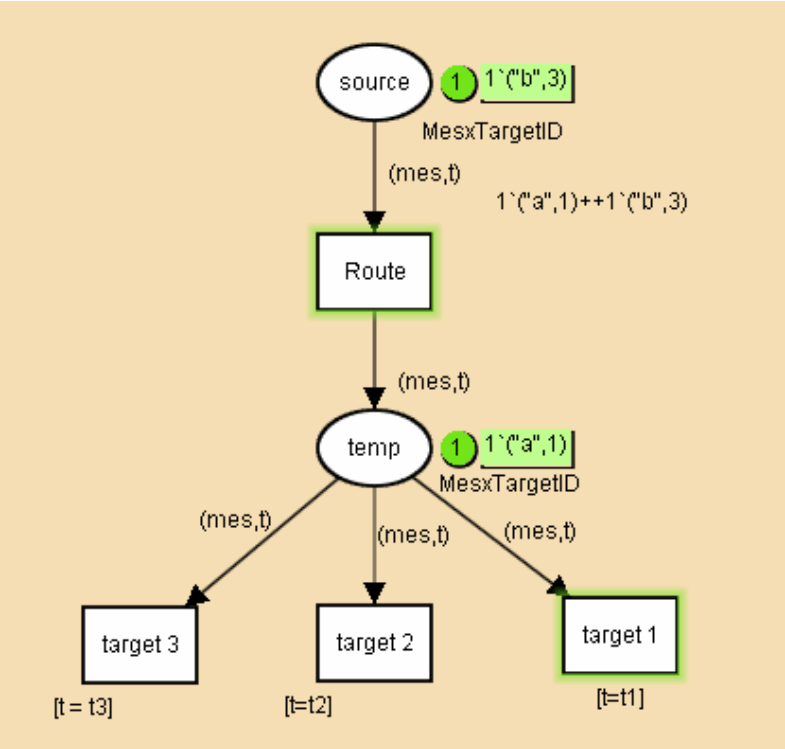


Figure 80

SOLUTION 2:
In order to decouple connection between a data source and a set of targets, which communicate asynchronously, while ensuring that data sent by the source is received by a target, to which it was dedicated, introduce an *asynchronous router*. The asynchronous router will direct a data directly to a dedicated target.

Implementation of Solution 2:
The list of instructions below describes how to implement Solution 2 of the ASYNCHRONOUS ROUTER pattern:

- Introduce transition *Route,* and connect place *source* to it, providing the composite data *(mes,t),* which includes an information object to be sent to a target and an identifier of the target.
- Introduce for each of the target an input place, where the router will place a dedicated data. As such for targets 1,2 and 3 introduce places *temp1, temp2, and temp3.*
- On the arcs connecting transition *Route* to target input places add a filtering condition, which would examine data provided by the router and define whether it is dedicated to the target, to which the input place corresponds. For instance, to

examine whether a data supplied by the router was dedicated to a *target1*, compare the identifier of dedicated target to the identifier of *target1,* i.e. *if t=t1 then 1`mes else empty.* The obtained construct incorporates Solution 2 of the DETERMINISTIC XOR-SPLIT pattern.

- Note that data from the *source* to every target input place *temp* is sent asynchronously, as the ASYNCHRONOUS TRANSFER pattern describes.

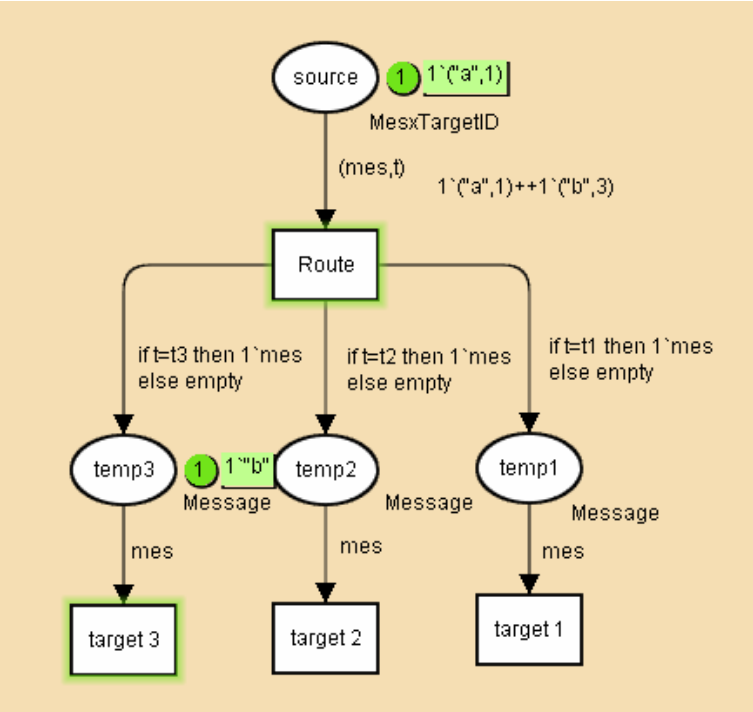


Figure 81

APPLICABILITY:
Apply this pattern to

- Increase flexibility in manipulation with targets, to which a data source sends data asynchronously.
- Decouple a source from targets, avoiding direct dependency between them.

CONSEQUENCES:
Application of this pattern allows decoupling a source from targets in two different ways. Solution 1 stores all data routed from the source in a temporary data storage. However, this has a consequence that a source must change the format of data to specify explicitly the intended data recipient. From the target point of view, if initially the delivery of data was organized and initiated by a source, then after applying this solution, targets should become more active and take own initiative to get the data.

Solution 2 routes data from a source to an input place of a target, without involving the target in the procedure of selecting a dedicated data. However, the drawback of this solution is that the router must change each time a new target is added.

This pattern includes the ASYNCHRONOUS TRANSFER pattern to ensure that data is sent asynchronously, and the DETERMINISTIC XOR-SPLIT pattern, which guarantees that every data will be consumed by one-and-only-one dedicated target.

The major characteristic of this pattern is that a source, producing data for multiple targets, sends data asynchronously so to a single dedicated target, but does not broadcast the same to data to all targets. If it is necessary to broadcast data from a source to a set of targets, so that every target receives the same data, apply the BROADCASTING pattern.

Since this pattern is based on asynchronous communication, the data source does not know whether a target got the data sent to it. If for example the connection was broken, but the target got the data, the source may try to retransmit the same data again. Therefore, the

drawback of this pattern is that it does not guarantee that no duplicated data is transferred. In order to address this problem this pattern can be combined with Solution1 of the REDUNDANCY MANAGER pattern.

EXAMPLES:

- The secretary of a department is responsible for distribution of holiday cards. Instead of delivering a card to every employee of the department, the secretary puts the cards to the post-box. The employee will pick up the card on his own initiative.

RELATED PATTERNS: this pattern includes the DETERMINISTIC XOR-SPLIT , ASYNCHRONOUS TRANSFER and ID MATCHING patterns. This pattern is similar to the BROADCASTING pattern, and can be combined with the REDUNDANCY MANAGER pattern.

PATTERN 27: ASYNCHRONOUS AGGREGATOR
ALSO KNOWN AS: XOR-join

INTENT: to provide the holistic view on data, produced by multiple unrelated sources, through asynchronous data aggregation

MOTIVATION:
Often in situations involving the ASYNCHRONOUS ROUTER pattern, i.e. when a data source produces data dedicated to a certain target and transfers it asynchronously, while ensuring the loose coupling between the data source and targets correspondingly, there is a need to get an overview of data produced by the targets. Sometimes it is desirable to get the holistic view on all data produced by the targets, rather then sampling data from every target directly.

PROBLEM DESCRIPTION:
Figure 82 presents a solution of the ASYNCHRONOUS ROUTER pattern. The targets *target1, target2* and *target3* receive the data sent asynchronously from the *source* through the *Router.* Assume that after processing the received data by the correspondent targets it is necessary to show these data to an external actor, which does not want communicate to the targets directly, but have an access to all information stored in a single location.

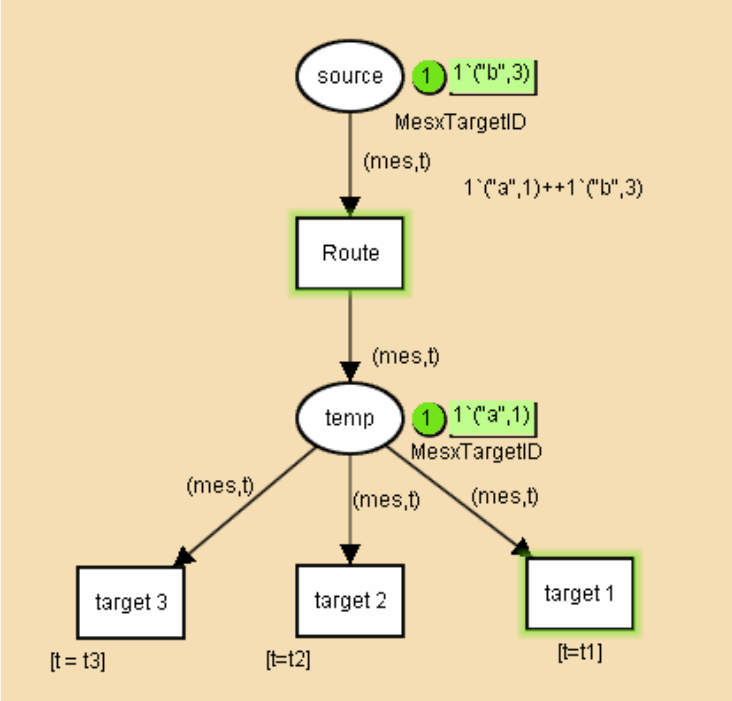


Figure 82

SOLUTION:
In order to provide the holistic view on data, delivered asynchronously from multiple unrelated sources, and abstract from sources' details, use an *asynchronous aggregator.* The asynchronous aggregator is a placeholder, which aggregates data delivered asynchronously from different sources and serves as a mediator between the data consumer and the data sources.

Implementation of Solution:
The list of instructions below describes how to implement the ASYNCHRONOUS AGGREGATOR pattern:

- Introduce an *Aggregate* place, where data provided as a result of processing by *target 1,2 and 3* will be stored until an external *Process* takes initiative to consume the data available at a moment.
- Connect the available *target*(s) to place *Aggregate* and supply together with data produced by a target an identifier of the correspondent target, if required. Note that any *target* may deliver data to the placeholder *Aggregate* any time, without being

dependent on other targets. This means that although *Aggregate* placeholder collects the data from the targets, it does not synchronize the targets and does not put a require targets to deliver data concurrently.

- Figure 83 illustrates asynchronous aggregation of data delivered by *targets*. Note that every new data delivered by a target is represented by a separate token. If there is a need to organize data into a collection, the AGGREGATE OBJECTS pattern can be applied. The collection achieved in this way can by ordered by applying the QUEUE pattern.

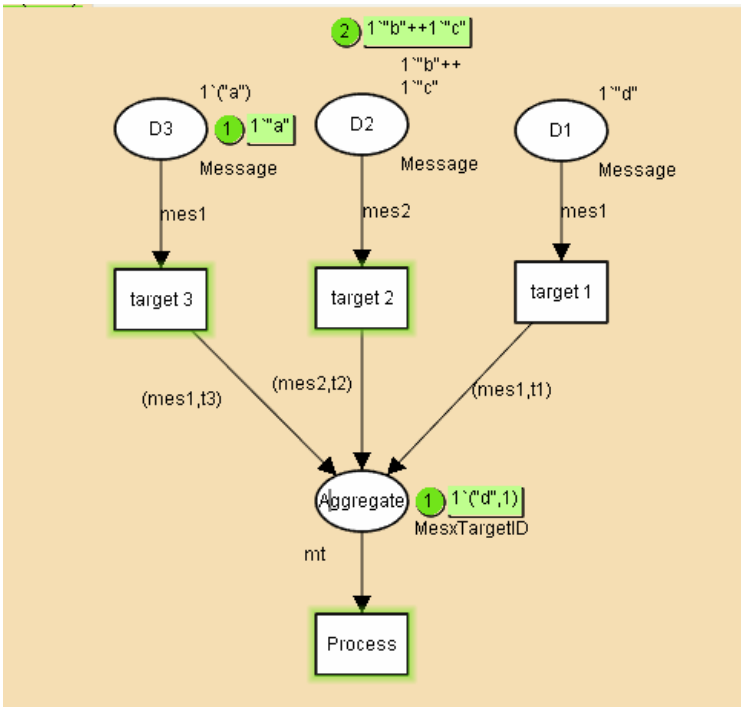


Figure 83

APPLICABILITY:
Apply this pattern in combination with the ASYNCHRONOUS ROUTER pattern to

- Aggregate data asynchronously delivered from multiple sources into a single place
- Isolate a data consumer from details of data sources.

CONSEQUENCES:
The ASYNCHRONOUS AGGREGATOR pattern corresponds to the ASYNCHRONOUS ROUTER pattern, and is based on the principles of the ASYNCHRONOUS TRANSFER. In this pattern, all data sources can work independently of each other and may deliver data any time without waiting for other sources to be ready.

The data aggregator serves as a sort of mediator between data producers and data consumer, hiding the details of data producers from the data consumer, and thus loosing out connection between them.

EXAMPLES:

- In order to pass a course, students must finish a personal assignment. Instead of allowing students to contact the course instructor directly, all assignments should be placed on the web-site, from which the instructor can pick them up when it is suitable for the instructor.

RELATED PATTERNS: this pattern corresponds to the ASYNCHRONOUS ROUTER pattern, and can be combined with the AGGREGATE OBJECTS pattern.

PATTERN 28: BROADCASTING
ALSO KNOWN AS:

INTENT: to allow broadcasting of data from a single source to multiple targets, avoiding direct dependency between them

MOTIVATION:
In some situations, there is a need to transfer some data from a single source to a set of targets, so that all targets receive the same information. This can be done using the RENDEZVOUS pattern, which allows distributing data to multiple targets concurrently. However, the data exchange achieved in this way requires tight coupling between the source and targets, resulting in low flexibility. Moreover, direct addressing from a source to targets can be cumbersome, when the number of targets and other target-related information is not known in advance.

PROBLEM DESCRIPTION:
Figure 84 presents the problem of direct addressing the data messages between source and targets. In order to broadcast the same data *mes* to several targets, the *source* needs to be connected to each of the targets, thus providing data directly to each of them. If the number of target data recipients varies, the source will be directly affected.

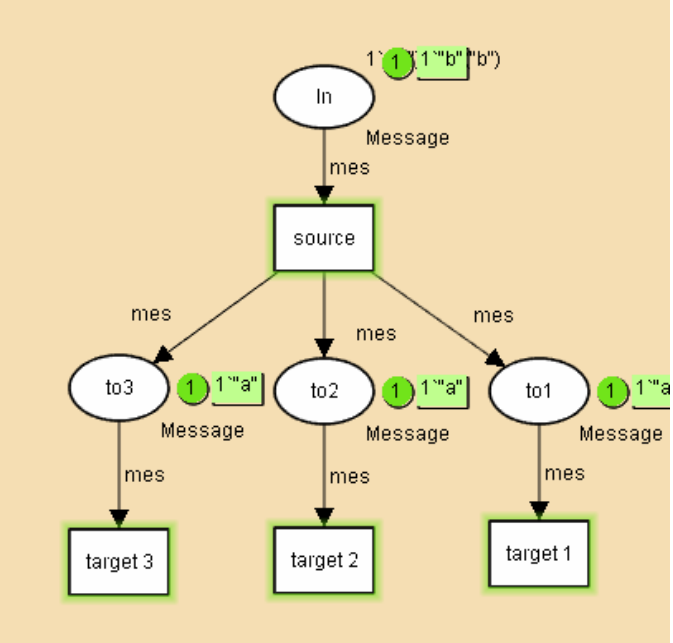


Figure 84

In terms of programming, this would lead to recompiling the source each time a new component is added or removed. Thus, the tight coupling between a source and targets gives no flexibility in manipulation of targets.

SOLUTION:
In order to loose out the connection between a source and targets, ensuring that the same data is broadcasted to each of the targets, decouple the source from the targets by introducing an intermediate placeholder, where data transferred from the source to the targets will be collected and stored. The source will provide data to the placeholder, and targets will take the data from this place on their own initiative.

Implementation of Solution:
The list of instructions below describes how to implement the BROADCASTING pattern:

- Merge the places connecting the source and the targets in one place *Router*. Note that it is possible to aggregate all data into a collection as it is described in the AGGREGATE OBJECTS pattern.
- Connect the merged place *Router* to the targets. Provide the data *mt* to a target, by drawing an arc from the router to a target. Return the data back to the router, so that other targets can get the same information.

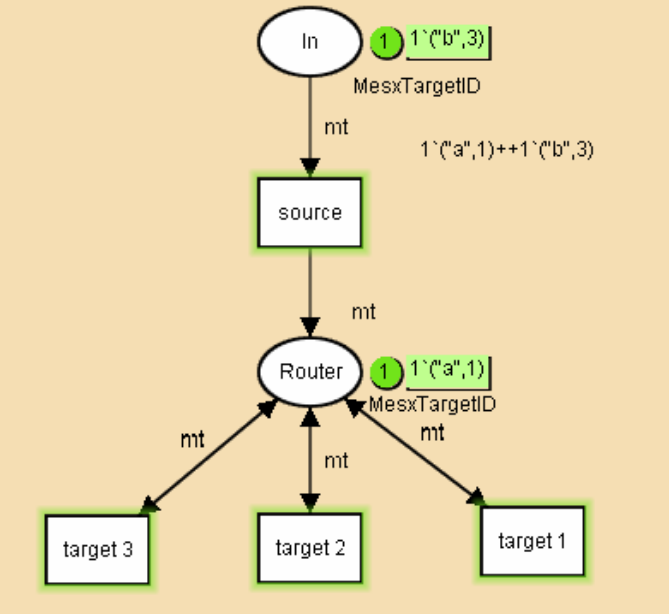


Figure 85

APPLICABILITY:
Apply this pattern to

- Broadcast data from a single source to multiple targets, so that every target receives the same data.
- Increase flexibility in manipulation of targets by decoupling them from the source.

CONSEQUENCES:
The BROADCASTING pattern provides a loose coupling between a source and targets, ensuring that every target gets the same data. The advantage of this pattern is that the source does not need to care any more about the delivery of data to targets, but only to the router. Targets themselves take care of consuming the data from the router.

Although BROADCASTING ensures that all targets get the same data, it is not guaranteed that every target will consume the data only once. In order to solve this problem, this pattern can be combined with Solution 2 of the REDUNDANCY MANAGER pattern.

EXAMPLES:

- The proceeding of a conference by decision of a chief of the department must be distributed to all employees. One exemplar of proceedings is placed in the secretary office. In order to track that all employees processed the material, a list with the names of the employees is attached to the proceedings. Employees take the proceedings on their own initiative and to indicate that they already processed them, put their signature on the list.

RELATED PATTERNS: this pattern can be combined with the AGGREGATE OBJECTS pattern and the REDUNDANCY MANAGER pattern.

PATTERN 29: REDUNDANCY MANAGER

ALSO KNOWN AS:

INTENT: to prevent transfer of duplicated data between loosely-coupled actors, communicating asynchronously

MOTIVATION:
When sending an information object from one place to another, it is sometimes undesirable that the sender retransmits the same data again. In the business context, for instance, it is undesirable to send the same bill to the customer twice.

PROBLEM DESCRIPTION:
Figure 86 presents the solution of the ASYNCHRONOUS ROUTER pattern. In this net, the source does not know if a target received the data message, and how many of such messages have been sent. Assume that during the data transfer an interrupt occurred, but a target did process the data. Since the communication is asynchronous, the source does not know if the target received the data, and can transfer the same data again.

Figure 87 presents the solution of the BROADCASTING pattern. In this net, the same data should be transferred through the router to all targets, therefore after sampling data from the *Router* a target returns the data back. However, it is not guaranteed that every target will consume the data only once.

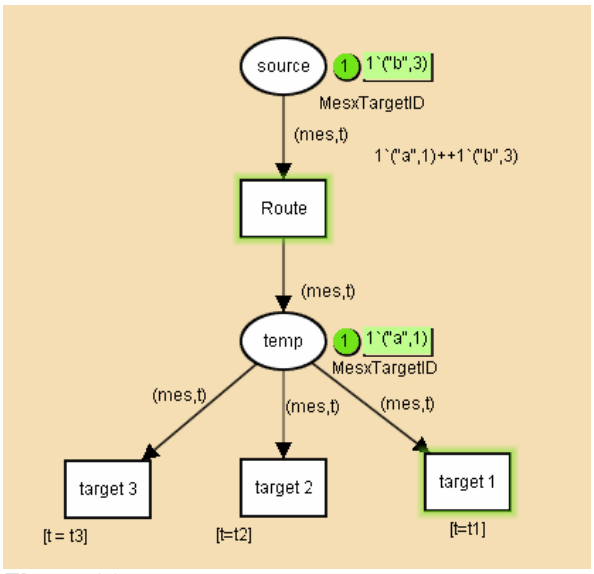


Figure 86

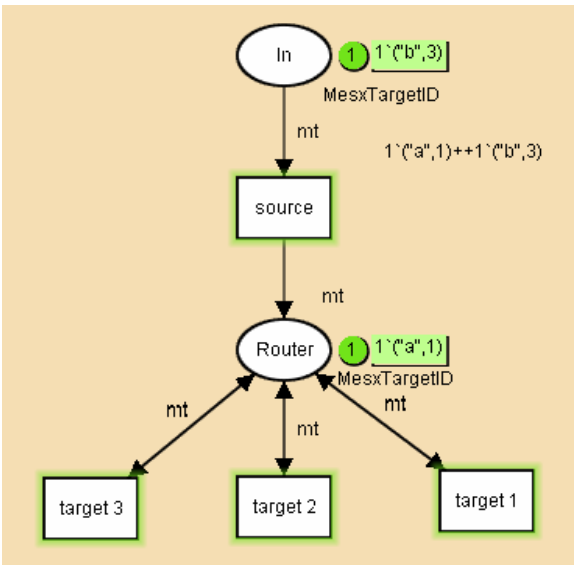


Figure 87

SOLUTION 1:
In order to avoid redundant data to be asynchronously transferred from a single source to a dedicated target, apply a state-dependent FILTER. The state, associated with a number of times a data was transmitted, defines whether the data will be sent to a target or will be filtered out.

This solution is valid under an assumption that every data message has a unique identifier associated with it.

Implementation of Solution 1:
The list of instructions below describes how to implement Solution 1 of the REDUNDANCY MANAGER pattern:

- Ensure that data elements available in *source* are coupled to a unique identifier, for instance (*data_id, data_value*).
- Create a list of identifiers *lid* of type *ListID*, which stores each data identifier only once. Before sending a data element to the router, check whether an identifier associated with the data element is not a member of the list *lid.* If the identifier is not

in the list, transfer data message to the router, otherwise discard the data message, so that no new attempt to send a duplicated data message happens. Note that the choice between execution of transitions *discard* and *send* is made according to the DETERMINISTIC XOR-SPLIT pattern.

- The function of checking the inclusion in the list is shown below:
  fun elt(mid,[])=false| elt(mid, h::queue)= if (mid=h) then true else elt(mid, queue);

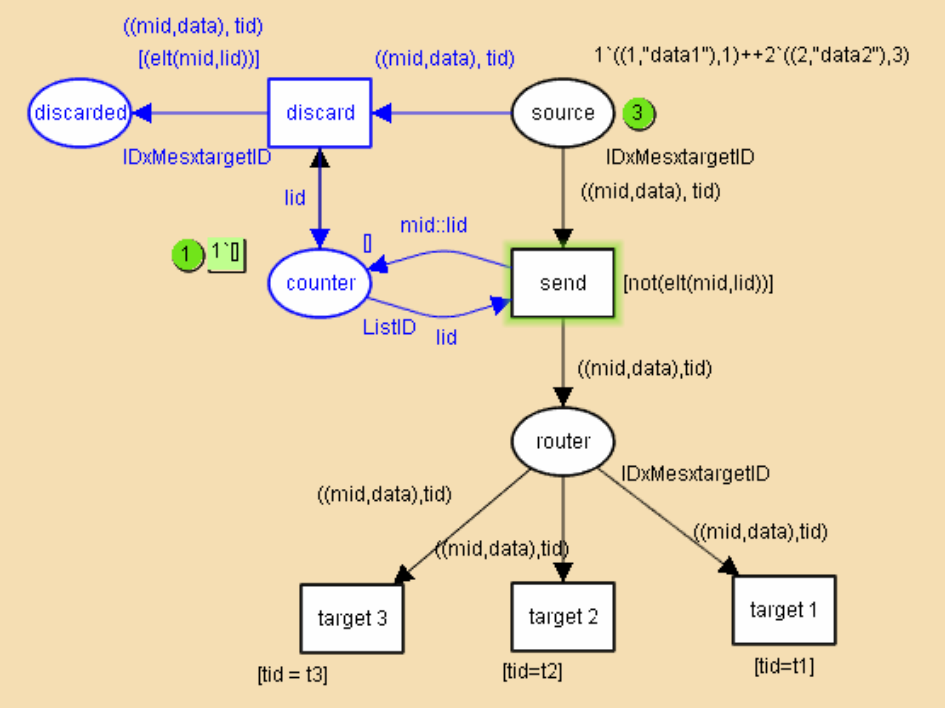


Figure 88

SOLUTION 2:
In order to avoid redundant data to be asynchronously broadcasted from a single source to multiple targets, apply the Solution 1 of the DETERMINISTIC XOR-SPLIT pattern. This will ensure that every target will process only new data, and no data read by the target will be read multiple times.

Implementation of Solution 2:
The list of instructions below describes how to implement Solution 2 of the REDUNDANCY MANAGER pattern.

- Modify the format of data stored in place *router* in order to associate every data with a list of targets, by which this data was received. When data was broadcasted to all targets, the list of targets associated with the data will contains identifiers of all targets.
- When providing a data to the transition, provide also a correspondent targets' list. Check if the data was not yet received by a target, by adding a transition guard *[not(elt(target_id,list_of_targets))]*.
- If a target is not an element of the targets' list, then this target did not receive this data yet. In this case, the target may receive the data, send it back to the router, while updating the target's list. The condition in the transition guard ensures that the target will not read the same data again.

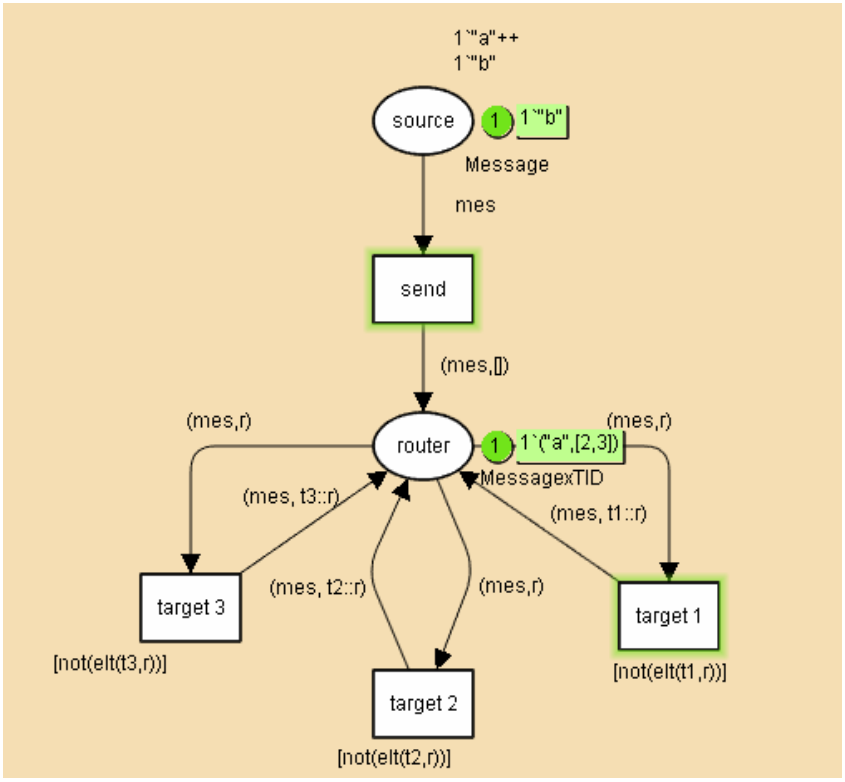


Figure 89

Because of the synchronous communication between the router and the targets, the data, consumed by all targets, may accumulate in the *Router*. To avoid this, combine this pattern with the BLOCKING STATE-INDEPENDENT FILTER pattern for checking the size of a list of targets, associated with every data and removing the correspondent data from the *router* place in case of condition fulfillment. Assuming that the total number of targets is known in advance, it is possible to withdraw data, which was broadcasted to all targets. Figure 90 demonstrates this functionality by means of the *Withdraw data* transition and arcs connected to it.

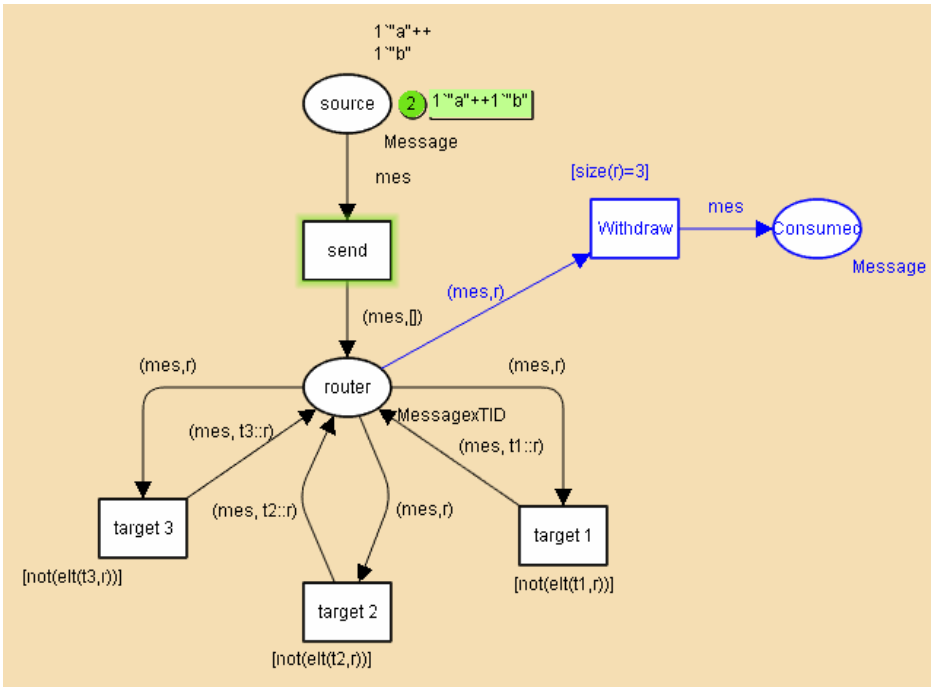


Figure 90

APPLICABILITY:
Apply this pattern in combination with the BROADCASTING and ASYNCHRONOUS ROUTER patterns to avoid transmission of duplicated data to decrease the redundancy of data transmitted asynchronously from one place to another.

CONSEQUENCES:
This pattern helps to avoid the duplicates of data to be occasionally retransmitted or processed. In its solutions, this pattern uses the FILTER and the DETERMINISTIC XOR-SPLIT patterns.

EXAMPLES:
- Money transactions, where the same bill should not be paid multiple times.
- Subscription requests, where the same user should be registered only once.

RELATED PATTERNS: this pattern uses the DETERMINISTIC XOR-SPLIT pattern and can be combined with BLOCKING STATE-INDEPENDENT FILTER.

PATTERN 30: DATA DISTRIBUTOR

ALSO KNOWN AS: DATA DECOMPOSITION, DATA SPLIT

INTENT: to support parallel data processing by distributing data between several independent actors

MOTIVATION:
In most business processes and information systems, where continuously growing amount of data for processing is involved, there is a need to improve the processing efficiency by introducing the data parallelism, which relates to both flow and structure of the information. Depending on the processing context, the nature and complexity of data, it might be necessary to involve several either identical or specialized actors in processing data concurrently, rather then letting a single actor doing all work sequentially instead.

PROBLEM DESCRIPTION:
Consider an automobile factory, which produces car components required for car assembly. The sequential production of four wheels by a single actor, for instance, results in much longer waiting time before the process of car assembling can start, then single-wheel production by four independent actors. Similar, a bunch of other car units, required for assembly, can be produced faster by several specialized actors then by a singe generic one.

SOLUTION:
In order to scale the throughout of processing, use a *data distributor*. The data distributor takes the compound data unit as an input and distributes it between several concurrent processing streams, either specializing or performing the same set of operations.

Depending on the data structure, the complexity of data, and the context of data processing, the data distributor either spreads the input data equally or divides it among several different places for independent processing.

***Implementation of Solution*:**
The list of instructions below describes how to implement the DATA DISTRIBUTOR pattern:

- Introduce a new transition *Distribute,* which will distribute the data received from the source place *In* between a set of other locations.
- Define the distribution rules, which specify what data is to be provided for each of the outgoing places of transition *Distribute*. Note that two types of data distribution can be applied, i.e. replication of input data and correspondent distribution of replicas between all outgoing places, or decomposition of input data into smaller parts. Figure 92 and Figure 93 illustrate data replication and data decomposition correspondingly.
- Specify the distribution rules on the arcs, connecting transition *Distribute* and outgoing places, either explicitly or encapsulate them into functions.

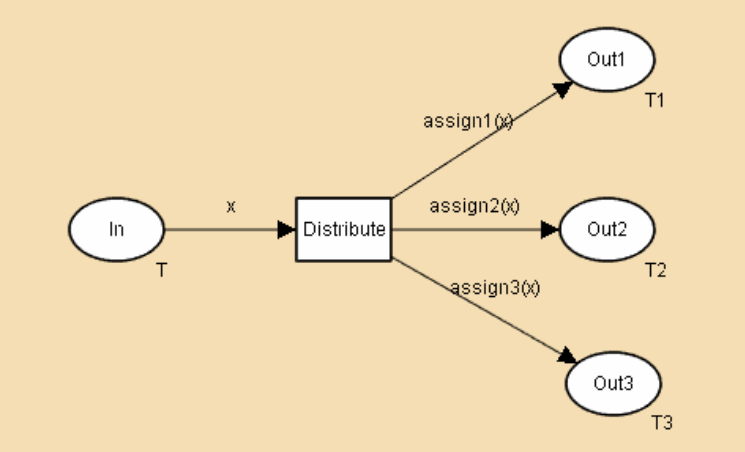


Figure 91

Figure 92 illustrates distribution of data replicas *(a,b,c)* received from place *In* between the outgoing places *Out1, Out2* and *Out3.*

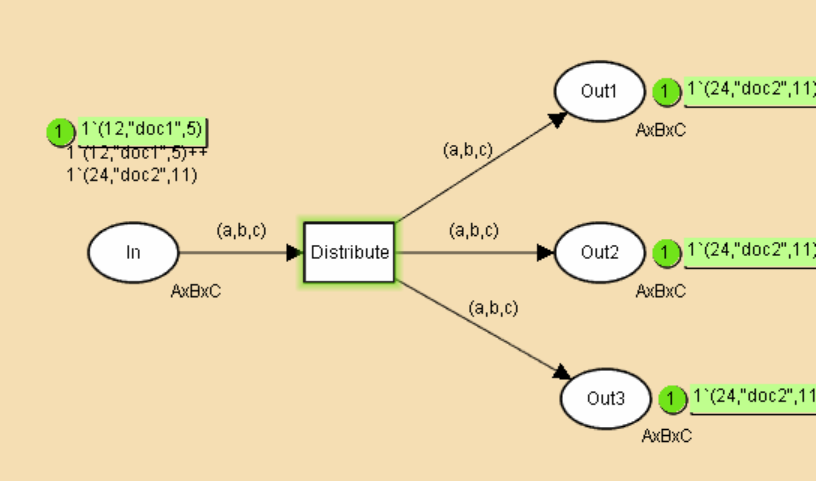


Figure 92 Distributing replicated data

Figure 93 illustrated distribution of data *(a,b,c)* received from place *In* and decomposed into smaller parts, between the outgoing places *Out1, Out2* and *Out3.*

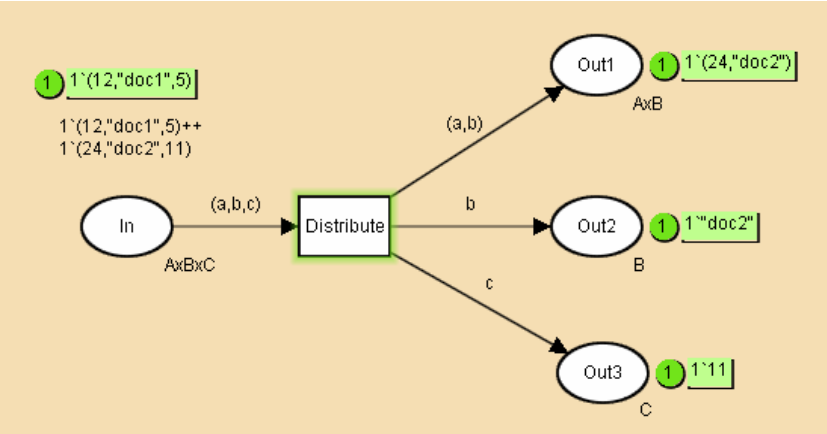


Figure 93 Distributing decomposed data

APPLICABILITY:
Apply this pattern to

- Support parallel processing of data.
- Decompose a single compound request from a client into several simpler requests and distribute them for parallel processing.

CONSEQUENCES:
This pattern supports distribution of data over multiple places for independent processing. Typically, after processing the data, it is necessary to combine the results of processing back into a single data unit. This can be done by applying the DATA MERGE pattern.

For concurrent distribution of data between several places, this pattern uses communication of the RENDEZVOUS type.

The DATA DISTRIBUTOR pattern assumes a direct dependency between the source of data and targets processing it. This dependency can be removed by applying the BROADCASTING pattern for transferring the same data to all targets or by applying the ASYNCHRONOUS ROUTER pattern for transferring data for dedicated processing to a specific target.

EXAMPLES:

- To distribute the load between several processors a complex process is divided into several threads, which are distributed over available processors.

- Parallel servers and data mining applications are examples of distributing the function that the application is performing among the tasks. Each task operates on the same data but does something different.
- Transferring products from an organization to customers

RELATED PATTERNS: this pattern is similar to the BROADCASTING pattern and the ASYNCHRONOUS ROUTER pattern. This pattern should be used in combination with the DATA MERGE pattern. It uses RENDEZVOUS pattern in its solution.

PATTERN 31: DATA MERGE
ALSO KNOWN AS: DATA COMPOSITION

INTENT: to compose a single information object out of several smaller ones, when all parts required for composition become available

MOTIVATION:
The DATA DISTRIBUTOR pattern is applied to scale the throughput of processing based on ability to decompose compound data into independent processing streams, performing the same or distinct types of operations. Typically, after the results of processing become available there is a need to merge them back into one entity. Since composite data parts are produced in parallel, but it is not guaranteed that they all are delivered at the same time, it is necessary to synchronize the moment of data merge.

PROBLEM DESCRIPTION:
Consider the process of car assembling. The assembling of a car may start only when all required parts, i.e. an engine, wheels, a car frame, etc. produced by independent manufactures, become available.

SOLUTION:
In order to merge data units produced by independent actors into a single compound data entity, synchronize the outputs of actors by means of the *data merge* transition. The data merge transition will wait until all actors finished their processing, and produce a single compound data entity when all composite parts become available.

***Implementation of Solution*:**
The list of instructions below describes how to implement the DATA MERGE pattern:

- Introduce a new transition *Merge*, which will synchronize the delivery of data units *a, b,* and, *c* from input places *In1, In2,* and *In3.*
- Connect transition *Merge* to place *Out*, where a single compound data entity (*a,b,c*) formed out of transition inputs will be transferred.
- Note that the concurrent data exchange in this solution incorporates the RENDEZVOUS pattern.

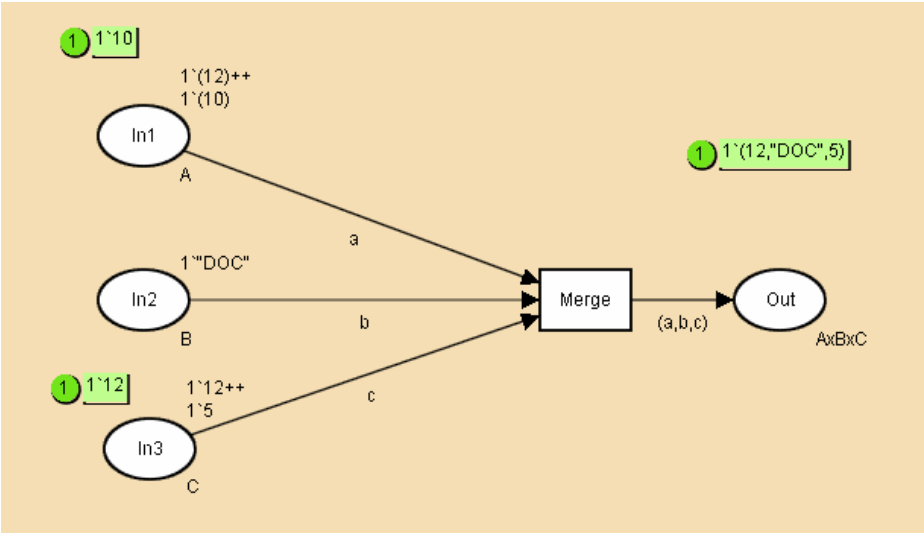


Figure 94

APPLICABILITY:
Apply this pattern to

- Synchronize several parallel branches, each of which delivers a data unit required for merging into a compound data entity.

CONSEQUENCES:
This pattern should be used in combination with the DATA DISTRIBUTOR pattern in order to form data decomposition-reconstruction construct.

Construction of composed data entity requires all data units from existing inputs to be available, which poses limitation on usage of this pattern in combination with the ASYNCHRONOUS ROUTER or BROADCASTING patterns. These patterns are based on the ASYNCHRONOUS TRANSFER, rather then the RENDEZVOUS. Improper combination of these patterns may lead to deadlocks and make models non-operational.

EXAMPLES:

- Information from multiple accounts is summarized to provide a single unified portfolio view to the customer
- Synchronized delivery of requests from multiple channels to a back-end application

RELATED PATTERNS: this pattern corresponds to the DATA DISTRIBUTOR pattern; it uses RENDEZVOUS pattern in its solution.

PATTERN 32: DETERMINISTIC XOR-SPLIT
ALSO KNOWN AS:

INTENT: to allow at most one transition out of several possible to execute, based on fulfillment of data conditions, which mutually exclude each other

MOTIVATION:
In many systems, there are data structures, which can be accessed by several tasks concurrently. However, it is not allowed that several tasks execute at the same time, i.e. only one task out of several possible need to be selected. In terms of safety requirements for event-driven systems, it is allowed that after a certain event only one of two possible events happen, but not both. Similar, in some situations there is a need to ensure that at most one process can be engaged in a specified activity at a time.

PROBLEM DESCRIPTION:
Assume that a place *data* and two transitions *Activity1* and *Activity2,* which have access to data stored in this place, are given. Depending on the value of data supplied by place *data,* either the value of data *d* is bigger than 5 or not more then 5, only one of the transitions *Activity1* and *Activity2* may execute. In the net in Figure 95 both transitions are enabled, however the choice of an activity is not explicitly defined.

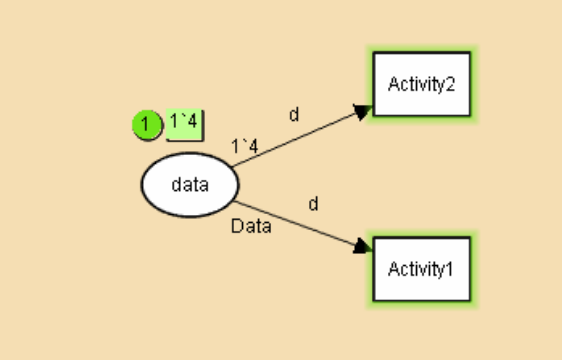


Figure 95

SOLUTION 1:
In order to define explicitly which transition out of several possible can be selected for execution, associate all transition with data conditions, which mutually exclude each other. Evaluation of mutually excluding data conditions results in selection of at most one transition.

Implementation of Solution 1:
The list of instructions below describes how to implement Solution 1 of the DETERMINISTIC XOR-SPLIT pattern:

- Connect place *data*, where data for execution of activities is stored, to the transitions *Activity 1* and *Activity2*, specifying the flow of data *d* by direction of connecting arcs.
- Add data conditions into the transition guards, ensuring that they cover all possible data values, but mutually exclude each other. In Figure 96, a guard of transition Activitiy2 evaluates to *true* and enables the execution of this transition if the data *d* is bigger then 5. A guard of transition *Activity 1* evaluates to true in all the rest cases, i.e. if the value of data *d* is at most 5. Note, that since the data included in guard can be of composite nature or more complex type, several conditions may be specified in a guard respectively.

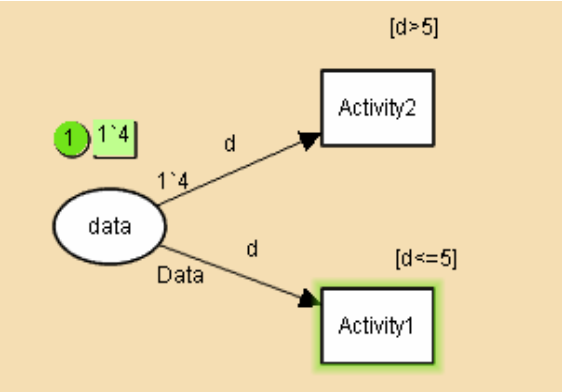


Figure 96

SOLUTION 2:
In order to define explicitly which transition out of several possible can be selected for execution, but to make data conditions transparent for the transitions, associate all transition with data conditions, which mutually exclude each other, and make evaluation of these condition in advance. Evaluation of mutually excluding data conditions results in selection of at most one transition.

Implementation of Solution 2:
The list of instruction below describes how to implement Solution 2 of the DETERMINISTIC XOR-SPLIT pattern:

- Decouple place *data,* which provides data inputs to transitions *Activity1 and Activity2*, by introducing in between two intermediate places *Data Act1* and *Data Act2*. Places *Data Act1* and *Data Act2* store only data that satisfies criteria of *Activity1* and *Activity2* respectively.
- Introduce transition *Define branch* in order to move the responsibility for evaluation of data conditions and selection of an activity away from the activities.
- Connect place *data* to transition *Define branch*, providing the input data *d.* Connect this transition to places *Data Act1* and *Data Act 2*, and specify on the arcs data conditions *"if d>5 then 1`d else empty"* and "*if d<=5 then 1`d else empty*", which must be satisfied for selecting the corresponding branch. Note that the data conditions mutually exclude each other.
- As a result of evaluation of condition "*if d>5 then 1`d else empty"* data *d* will be placed into the *Data Act2* if the value of data d is bigger then 5, otherwise the condition leading on the arc to *Data Act1* will evaluate to true, and data *d* will be correspondingly placed for consumption by *Activity 1.*

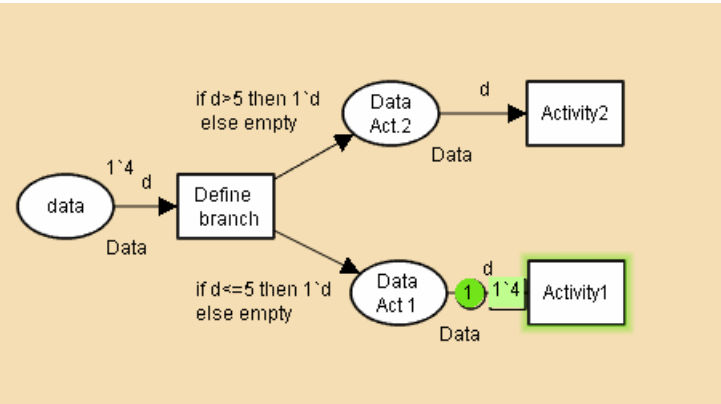


Figure 97

APPLICABILITY:
Apply this pattern to

- Make explicit selection of at most one activity, event, task, etc. modeled by means of transition based on the characteristics of data provided as an input.

CONSEQUENCES:
The DETERMINISTIC XOR pattern provides two solutions. Apply Solution 1 if it is necessary to involve the transitions, selection of one from which needs to be made, into the evaluation of data conditions directly. Solution 1 extends the BLOCKING STATE-INDEPENDENT FILTER pattern, which allows input data to come through only if a data condition specified in a transition guard is satisfied. In contrast to the BSI FILTER pattern, this pattern covers all range of values associated with input data, ensuring that all input data can be unambiguously categorized according to mutually excluding data conditions.

In its turn, Solution 2 allows transitions, at most one of which can be selected, to abstract from evaluation of data conditions involved in the selection procedure, thus making data transparent to transitions.

From the control-flow point of view, Solution 1 and Solution 2 differ in the moment of taking a decision for selecting a transition for execution. In Solution 1, this decision is made on the latest moment, while in the Solution 2 as early as possible.

This pattern addresses problem of scheduling nature, similar to the ones addressed in the NON-DETERMINISTIC XOR and OR patterns.

EXAMPLES:
- Computer resources that can only be manipulated by one task at a time

RELATED PATTERNS: this pattern extends the BLOCKING STATE-INDEPENDENT FILTER pattern. This pattern is similar to the NON-DETERMINISTIC XOR-SPLIT and OR patterns.

PATTERN 33: NON-DETERMINISTIC XOR-SPLIT
ALSO KNOWN AS:

INTENT: to allow any transition out of several possible, but satisfying the same data condition, to execute

MOTIVATION:
The DETERMINISTIC XOR-SPLIT pattern is often applied for allowing at most one transition out of several possible to execute, based on fulfillment of data conditions, which mutually exclude each other. In some situations, there is a need to make such mutually excluding data conditions less strict. For example, two types of resources need to handle tree types of tasks. Each group of resources specializes in carrying out only one specific task; however, a task of more generic type can be executed by any kind of resource. Being able to relax mutually exclusive data conditions can result in more flexible work distribution.

PROBLEM DESCRIPTION:
Figure 98 presents solution 1 of the DETERMINISTIC XOR-SPLIT pattern. In this net, there is a strict separation of data conditions on basis of mutual exclusion. As such, *Activity1* may execute if the value of data *d* provided by place *data* is bigger than 5, and *Activity2* may execute if the value of data does not exceed 5. Now assume that we want to specify that data, the value of which is bigger then 5 but less then 10, can be handled by any of the activities, still ensuring that only one activity may execute at a time.

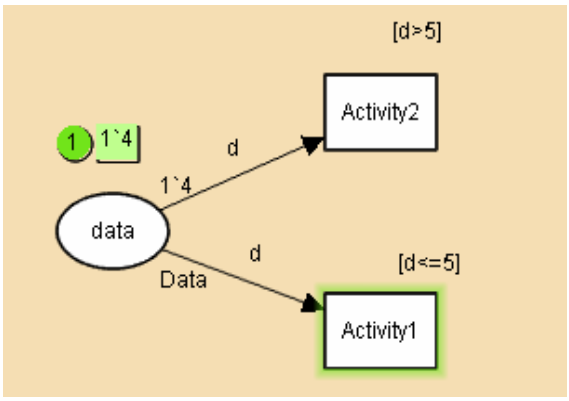


Figure 98

SOLUTION:
In order to allow non-deterministic selection of at most one transition out of several possible, associate every transition with overlapping data condition.

Implementation of Solution:
The list of instructions below describes how to implement Solution of the NON-DETERMINISTIC XOR-SPLIT with overlapping data conditions pattern:

- Connect place *data*, where data for execution of activities is stored, to the transitions *Activity 1* and *Activity2*, specifying the flow of data *d* by direction of connecting arcs.
- Add data conditions into transition guards, ensuring that they overlap in the part common to both transitions. As Figure 99 illustrates, data, values of which vary between 5 and 10 can be processed by any of the activities, while all other values out of the specified range can be handled by only one activity. Note that the choice of activities, when common condition is satisfied, is non-deterministic. It is not possible to predict which of the activities will process values satisfying the range of overlapping values.

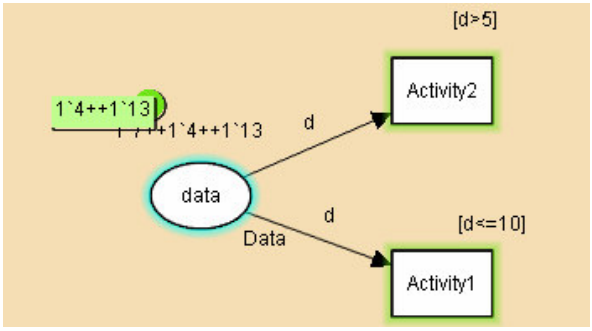


Figure 99

Figure 99 contains three ranges of data values, i.e. d<=5 for transition *Activity 1,* d>10 for transition *Activity 2*, and 5< d <=10, which represents the overlapping condition for both transitions respectively. Note however that the range of data values only for overlapping data conditions should be considered for realizing non-deterministic choice.

Similar, if only one of two transitions should be selected non-deterministically in the range of all possible data values, the data conditions can be omitted and the net presented in Figure 100 can be used instead.

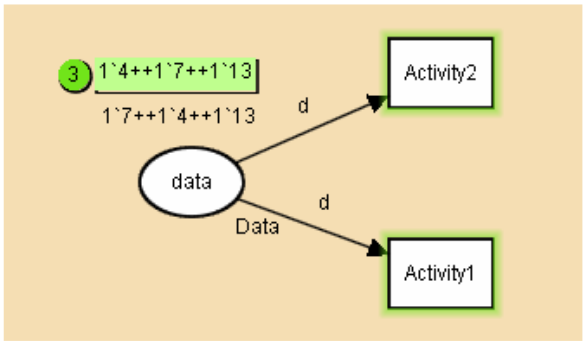


Figure 100

APPLICABILITY:
Apply this pattern to

- Realize non-deterministic selection of a task from a set of possible tasks, while ensuring that only one task may execute at a time.

CONSEQUENCES:
This pattern addresses the problem of scheduling nature, similar to the ones addressed in the DETERMINISTIC XOR-SPLIT and OR patterns.

In comparison to the mutually excluding data conditions, overlapping conditions in some way extend the way of handling data. Note that overlapping and mutually excluding data conditions can be combined. Opposite to overlapping conditions, one can specify insufficient conditions, i.e. when all range of data except a certain part is being handled in handled deterministically. For instance, if one task handles data with value that is smaller then 5, and another task handles data values that bigger then 10, then the data in range from 5 to 10, won't be handled at all; this may cause undesirable blocking or even deadlock. Insufficient conditions can be applied if one can ensure that only data in specified range is considered, however usage of insufficient data conditions is not desirable.

EXAMPLES:

- Dynamic assigning of generic tasks between two specialized groups of resources

RELATED PATTERNS: this pattern is similar to the DETERMINISTIC XOR-SPLIT and OR patterns

PATTERN 34: OR
ALSO KNOWN AS:

INTENT: to allow any number of tasks to be selected for execution based on fulfillment of a certain data condition

MOTIVATION:
The DETERMINISTIC XOR-SPLIT pattern allows only one of two possible tasks execute based on the data conditions mutually excluding each other. In some situations, however, there is no need to pose strict conditions on the number of tasks, which may execute at a time, but allow it vary and execute concurrently if a certain condition is fulfilled. For instance, when distributing work items, an employee gets a work item if he specializes in the requested type of work, however generic requests for attending the meetings must be executed by all employees.

PROBLEM DESCRIPTION:
Solution of the NON-DETERMINISTIC XOR-SPLIT, presented in Figure 101, allows at most one task, does not matter which, from two available tasks to execute if a condition common to both tasks is satisfied. Although this pattern allows selection of multiple tasks, it does not allow multiple tasks to execute concurrently.

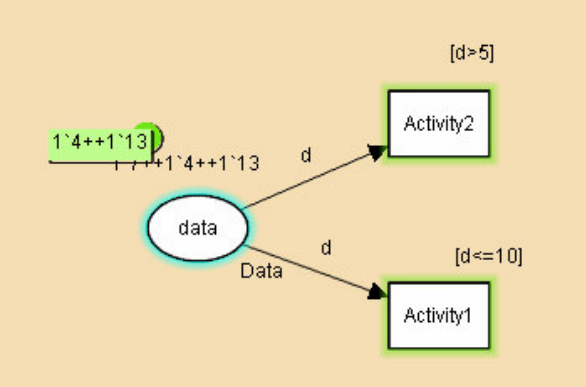


Figure 101

SOLUTION:
In order to enable execution of all from several available transitions in case of fulfillment a common to all transitions data condition, but at most one task in case of fulfillment of mutually excluding conditions, extend Solution 2 of the DETERMINISTIC XOR-SPLIT with overlapping data conditions.

Implementation of Solution 1:
The list of instructions below describes how to implement the OR pattern:

- Modify Solution 2 of the DETERMINISTIC XOR-SPLIT pattern by introducing the overlapping data conditions on the arcs from transition *Define branch* to places *Data Act1* and *Data Act2,* which provide input data to *Activity1* and *Activity2* respectively.
- Based on the rule of transition firing, specified data will be placed in the outgoing place if the condition on the arc connecting transition and the outgoing place is satisfied. As such, if the value of data supplied by place *data* is in the range between 5 and 10, then this data will be provided as an input to both transitions *Activity1* and *Activity2.* However, if data value is less then 5 then only transition *Activity 2* will execute, and if data value is bigger then 10, then it will be provided as an input to transition *Activity2*.

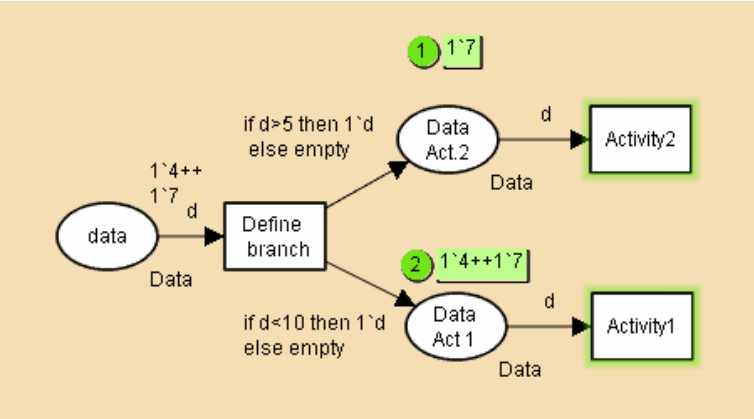


Figure 102

APPLICABILITY:
Apply this pattern to

- Allow one or more tasks to execute concurrently depending on fulfillment of data conditions associated with the tasks.

CONSEQUENCES:
Similar to the Solution 2 of the DETERMINISTIC XOR-SPLIT pattern, in this pattern tasks processing data, are not aware of conditions based on which the supplied data was selected. This allows data conditions be transparent and tasks to concentrate on actual processing of data.

Depending on which data conditions are satisfied, this pattern is able to behave as the ASYCNHRONOUS ROUTER or DATA DISTRIBUTOR pattern. However, the selection of the correspondent pattern, able to merge multiple branches present in this pattern, is not straightforward. Selection of the DATA MERGE pattern would lead to blocking if this pattern behaves as ASYNCHRONOUS ROUTER, or would miss synchronization point if the ASYNCHRONOUS AGGREGATOR is used in combination with the DATA DISTRIBUTOR. Therefore, if this pattern is applied, use the DATA MERGE pattern in combination with Booleans variables, indicating whether a correspondent branch was selected.

EXAMPLES:

- Scientific researches working in the same department work on individual research tasks separately, however they all attend the group meetings.

RELATED PATTERNS: this pattern extends Solution 2 of the DETERMINISTIC XOR-SPLIT pattern. This pattern can be combined with the DATA MERGE pattern.

CPN PATTERN RELATIONSHIPS

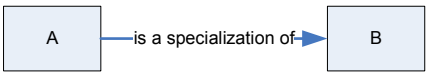
The 34 CPN patterns presented in this report, together with relationships between them, form a pattern language. In order to classify the CPN patterns we examine the nature of relationships between the patterns. We will use three types of primary relations, i.e. *specialization of a problem, use in a solution,* and *extension of an implementation,* and two types of secondary relations, i.e. *problem similarity,* and *solutions' combination,* to describe the pattern relationships. Some of the relationship types are based on the Zimmer classification [12].

The main purpose of this classification is to provide a holistic view on the catalog of patterns listed in this report, providing a means for selecting a number of patterns and determining how they can help in solving a problem under hand. The selected types of relationships can help to trace other patterns related to a chosen pattern, thus allow estimating an overall problems complexity, the tradeoffs made, and compare the chosen pattern with other patterns similar to it in order to select an optimal solution for a problem in the specific context.

Note that the types of the selected pattern relations are used in the description of every pattern. The details regarding combining one pattern with another one, or similarities between patterns, which are not indicated in the relationship diagram (Figure 103), can be found in the *Consequences and Related patterns* sections of a chosen pattern.

Primary relations

Problem-oriented



Pattern A is a *specialization* of more generic pattern B. Specific pattern A deals with a specialization of the *problem* the generic pattern B addresses, has a similar but more specialized solution. Pattern A includes all properties of pattern B, but adds some more restrictions by adding some specialized characteristics.

Example: patterns Priority Queue and FIFO Queue are both specializations of the Queue pattern

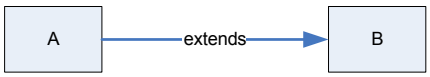
Solution-oriented

Pattern A *uses* pattern B in its solution. When building a solution for problem addressed by pattern A, one sub-problem is similar to the problem addressed by pattern B. Thus, the *solution of pattern* B is a composite part of the *solution of pattern* A. Whenever pattern A is used, pattern B should also be considered, since it makes a part of A. All instantiations of pattern A use pattern B.

Example: Lock Manager uses ID Matching, Asynchronous Router uses Asynchronous Transfer.

Since many CPN pattern have multiple solutions, for a sake of clarity we will use mnemonics "s1", "s2" and "s2" as identifiers for referring to Solution 1, 2, and 3 correspondingly, to make relations between pattern solutions explicit.

Solution implementation-oriented



Pattern A *syntactically extends* pattern B. Pattern A addresses a set of requirements to have more or slightly different functionality then the pattern B addresses. However, this is the implementation of B, which is syntactically extended by A, rather then a problem or a solution.

Example: implementation of Non-Blocking State-Independent Filter extends implementation of Blocking State-Independent Filter.

Secondary relations

Pattern A is *similar* to pattern B. Pattern A addresses a problem similar to the one addressed in pattern B. See description of a pattern to find out in what sense the pattern relates to the other pattern. Patterns A and B can be considered as alternatives of each other; compare them and select the one which fits the best.

Example: Asynchronous Transfer is similar to Rendezvous; Asynchronous Router is similar to Broadcasting.

Pattern A can be combined with pattern B. None of the patterns is a part of each other. Combining solution of pattern B with solution of pattern A can help in solving a more complex problem, then a single pattern solves in isolation. Use this relation to find out other patterns, which can be used in addition to pattern A.

Example: Shared Database can be combined with Copy Manager, Asynchronous Aggregator can be combined with Aggregate Objects

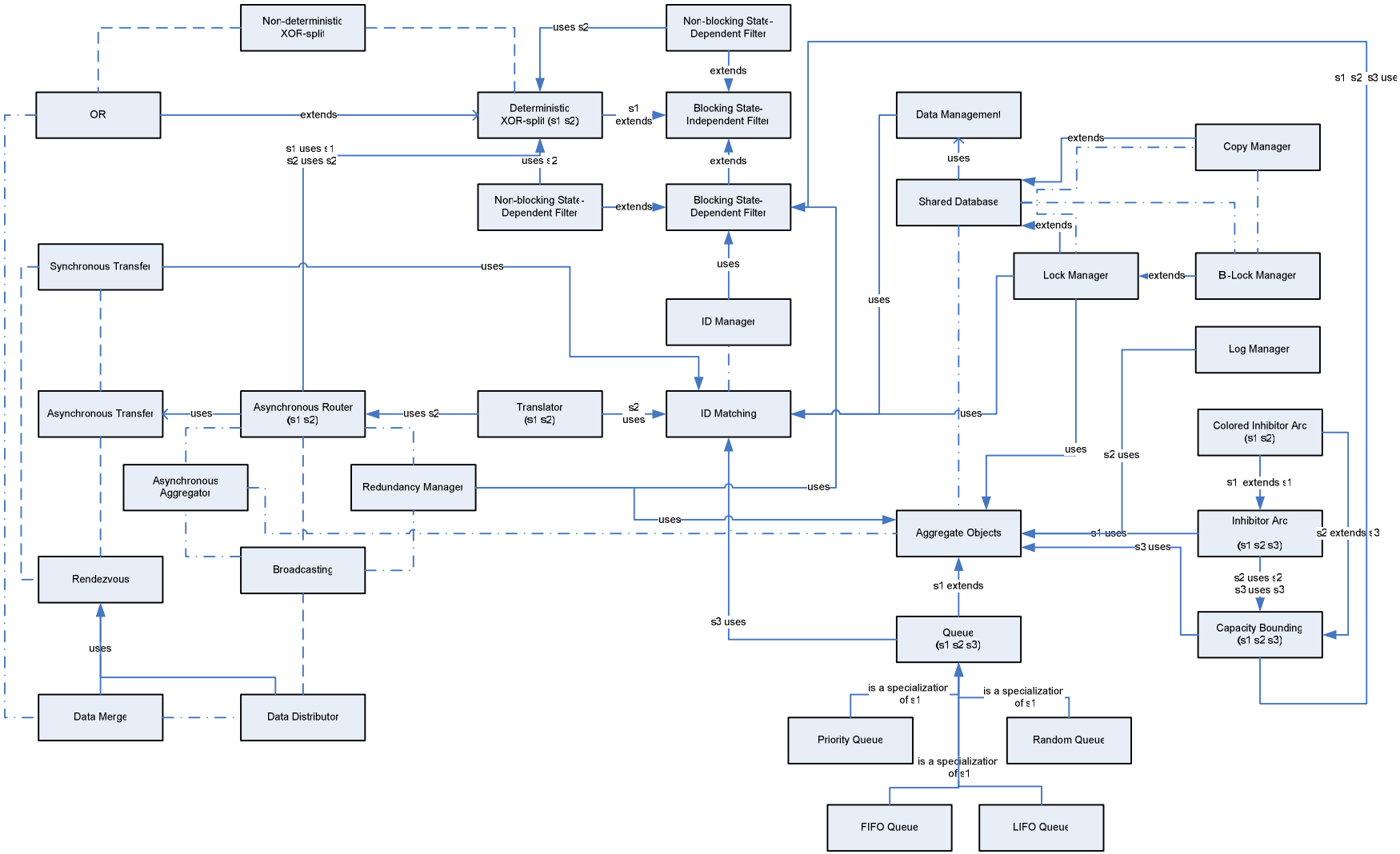


Figure 103 CPN pattern relationship diagram

CLASSIFICATION OF CPN PATTERNS

Although the CPN pattern relationship diagram in Figure 103 allows navigating through the catalog of the CPN patterns, it is not sufficient for classifying the patterns precisely and unambiguously. In order to provide the means for selecting an appropriate pattern, we adopt classification presented in [13] to categorize the CPN patterns.

As it was mentioned in the introduction, the CPN patterns aim to cover problems in the domain where data and control-flow perspectives interplay. In this domain, three pattern groups can be distinguished:

- patterns, where data perspective dominates, but which must be considered in the context of the control-flow;
- patterns, where the control-flow perspective dominates, but which are data-based;
- patterns, where data perspectives and control-flow perspectives are equally important and involved.

The intent of every pattern has been analyzed according to the following structure:
<*common components, diagnostic elements, supplementary components*>.

Common components define the set of related meanings, by which different patterns can be placed into one group. For instance, patterns addressing problems of creation new elements or entities, belong to the same group with a common component *create.* Thus, this is the intent of a pattern from the process (functionality) point of view.

> Example: patterns, whose main intent is to manage or control something, will be combined into the group with a common component *control.*

Diagnostic elements define the contrastive features, which distinguish the patterns belonging to the same common component. For instance, patterns belonging to the same common component *control*, i.e. control patterns, can involve different participants or differ by control parameters.

> Example: patterns, whose main purpose is to control such features as the order, the throughput, the quantity, belong to the same group with a common component *control* and can be distinguished by diagnostic elements *Order, Throughput, Quantity* respectively.

Supplementary components address additional features for extended definitions of meanings. This component addresses special circumstances of applying a pattern. This feature could be applied to distinguish the pattern from other patterns belonging to the same common component with the same diagnostic elements; however, multiple patterns may have the same supplementary component.

The classification list below presents characteristics of patterns in the following format:

Common component
- **Diagnostic component**
  - **Supplementary component**

Control
- Order of information objects (Queue)
  - by predefined scheduling policy (FIFO Queue, LIFO Queue, Random Queue)
  - by objects priority (Priority Queue)
- Availability/Consistency of information objects
  - by regular replication (Copy Manager)
- Concurrent access to information objects
  - by means of exclusive locks (Lock Manager)
  - by means of shared and exclusive locks (Bi-Lock Manager)
- Throughput of information objects
  - by inspecting content (Blocking State-Independent Filter, Non-blocking State-Independent Filter)

        - by inspecting state (Blocking State-Dependent Filter, Non-Blocking State-dependent Filter, Redundancy Manager)
    - Number of objects in place
        - by bounding the place capacity (Capacity-Bounding)

Discern
- Information objects
    - by identities (ID Matching)
    - by visibility (Shared Database)

Choose
- 1 branch deterministically (Deterministic XOR-split)
- 1 branch non-deterministically (Non-deterministic XOR-split)
- 1 or more branches deterministically (OR)

Create
- Information objects
    - by unique generation (ID Manager)
    - by decomposing into parts (Data Distributor)

Assemble
- Information objects
    - by aggregating into a collection (Aggregate Objects)
    - by synchronizing composite parts (Data Merge)
    - by asynchronous merging (Asynchronous Aggregator)

Access
- Information objects
    - by read/write operations (Data Management)

Inspect
- "Non-containment" property of place (Colored Inhibitor Arc)
- "Zero"-property of place (Inhibitor Arc)

Monitor
- Process execution-relevant information
    - by data logs (Log Manager)

Transform
- Information objects
    - by adjusting the data format (Translator)

Transfer
- Information objects
    - Asynchronously
        - directly
            - from a source to a target: 1-to-1 (Asynchronous Transfer)
        - indirectly
            - from a source to one of several targets: 1-to-1 (Asynchronous Router)
            - from a source to multiple targets: 1-to-N (Broadcasting)
    - Synchronously
        - between two actors: 1-to-1 (Synchronous Transfer)
    - Concurrently
        - from N sources to M targets: N-to-M (Rendezvous)

RELATED WORK

The concept of patterns has been originated by Christopher Alexander, who defined them in the architectural context. Alexander was the first one who proposed a format for documenting patterns and combined patterns into a pattern language [14]. The pattern initiative was supported and resulted in a set of significant milestones, i.e. pattern languages, in other application fields and domains.

In the past years, an idea of patterns became popular in object-oriented software community. As an evidence of this, we refer to the 23 design patterns by Gamma [4], and their numerous successors, such as:

- Patterns for knowledge and software reuse by Sutcliffe [15];
- Design patterns in communication software by Linda Rising [18];
- Framework patterns by Wolfgan Pree [19];
- etc.

Alternatively to the generic patterns, a set of language-specific pattern languages (UML, Smalltalk, XML, Python, etc.), links to which can be found in the pattern digest library [20], has been discovered and documented.

Furthermore, some work has been done on formalizing the organization, process, analysis, and business-related patterns. Among them:

- Analysis patterns by Martin Fowler [21];
- Enterprise Architecture Patterns by Michael A. Beedle [22];
- Framework Process Patterns by James Carey [23];
- Patterns for e-business [16], which focus on Business patterns, Integration patterns, and Application patterns;
- Business patterns at work [15], which use UML to model a business system;
- Process patterns [24].

In a line with an initiative (cf. www.workflowpatterns.com) to capture the functionality of PAIS in term of patterns, workflow [1], data [2], and resource patterns [3] have been discovered. No work on discovering patterns combining several perspectives have been done yet. Our initiative to discover patterns combining data flow and control flow, resulted in 34 patterns listed in this report. We selected Colored Petri Nets as an implementation language. As far as we know, no effort to formalize patterns in Colored Petri Nets, except the ones fragmentally listed by Kurt Jensen in [5], [6], [7], and [9], Wil van der Aalst in [8], and Kees van Hee in [11] have been made.

FUTURE WORK

We do not claim the completeness of the implementation patterns in CPN, listed in this document, since they are the result of an explorative work and are not derived in a systematic manner. One of the intents of the CPN patterns is to make them available to the CPN community in a form of a pattern library in order to share sound solutions, proven by experience, between developers. We want to encourage members of the CPN community to extend the catalog of patterns by the ones not covered here. Moreover, these patterns can serve as a language enhancing communication between developers, allowing communicate problems and solutions unambiguously. Note that although CPN patterns are language-specific, they can be applied for modeling and design of any kind of dynamic systems with elements of concurrency.

On the other hand, since the CPN patterns are language-specific, they cannot be applied in the scope of PAIS, based on languages different from CPN. Therefore, this work is considered as a background work for discovering tool-independent patterns combining different perspectives.

ACKNOWLEDGEMENTS

We would like to thank Kurt Jensen for contributing to the work reported in this paper. This paper is a spin-off of the PhD course on Colored Petri Nets he gave in Eindhoven. His experience in modeling using Colored Petri Nets has been vital for collecting and describing the patterns presented.

REFERENCES

[1] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. *Workflow Patterns*. *Distributed* and Parallel Databases, 14(1):5-51, 2003.

[2] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. *Workflow Data Patterns*. QUT Technical report, FIT-TR-2004-01, Queensland University of Technology, Brisbane, 2004.

[3] N. Russell, A.H.M. ter Hofstede, D. Edmond, W.M.P. van der Aalst. *Workflow Resource Patterns*, 2004, BETA Working Paper Series, WP 127, Eindhoven University of Technology, Eindhoven

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, 1995.

[5] K. Jensen: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts. Monographs in Theoretical Computer Science, Springer-Verlag, 2nd corrected printing 1997.

[6] K. Jensen: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 2, Analysis Methods. Monographs in Theoretical Computer Science, Springer-Verlag, 2nd corrected printing 1997.

[7] K. Jensen: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 3, Practical Use. Monographs in Theoretical Computer Science, Springer-Verlag, 1997.

[8] W.M.P. van der Aalst. *Lecture notes on Process-modeling*. Eindhoven University of Technology, 2003.

[9] Official site of CPN Tools http://wiki.daimi.au.dk/cpntools-help/cpntools-help.wiki

[10] Proceedings of Fifth Workshop and Tutorial on Practical Use of Colored Petri Nets and the CPN Tools. October 8-11, 2004

[11] K.M. van Hee. *Information Systems Engineering:A Formal Approach.* Cambridge University Press,1994

[12] W. Zimmer. "Relationships between Design Patterns" in Pattern Languages of Program Design, J. O. Coplien and D. C. Schmidt (eds.), Reading, MA: Addison-Wesley, 1995.

[13] S. Hasso and C. R. Carlson. *Linguistics-based Software Design Patterns Classification.* In Hawaii International Conference on Computer Sciences, Honolulu, HI, Jan 2004.

[14] C. Alexander. *A Pattern Language: Towns, Building and Construction*. Oxford University Press, 1977.

[15] A. Sutcliffe. Patterns for Knowledge and Software Reuse. Lawrence Erlbaum Associates, Inc. March 2002.

[16] J.Adams, S.Koushik, G.Vasudeva, G.Galambos. *Patterns for e-Business. A Strategy for Use.* IBM Press, 2001

[17] H. Eriksson, M. Penker. *Business Modeling with UML. Business patterns at work.* Wiley, John & Sons, Incorporated, 1998.

[18] L. Rising. *Design Patterns in Communication Software*. Cambridge University Press, 2000.

[19] W. Pree. *Framework patterns*. SIGS Bks, 1996.

[20] Pattern digest library http://patterndigest.com/books/otherlang.jsp

[21] M. Fowler. *Analysis Patterns.* Addison Wesley Longman, 1995.

[22] M.A. Beedle. *Enterprise Architecture Patterns.* Cambridge University Press, 1998.

[23] J. Carey, B. Carlson. *Framework Process Patterns*. Addison Wesley Longman, Inc., 2001.

[24] S.W. Ambler. *Process Patterns,* Cambridge University Press,1998.