Towards a Pattern Language for Colored Petri Nets

Nataliya Mulyar and Wil 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

Abstract. Experienced Petri net modelers model in terms of patterns, just like object-oriented programmers use the design patterns of Gamma et al. So far there is no any structured collection of patterns for Colored Petri Nets. We have empirically collected 34 patterns in Colored Petri Nets and documented them in the pattern format. The patterns focus on the interplay between data-flow and control-flow, (i.e. the essence of Colored Petri Nets), and have been modeled using CPN Tools. The goal of the patterns is to assist and train inexperienced modelers, and to serve as a domain language for communicating problems and solutions. In this paper, we give a summary of the CPN pattern language and give an overview of the patterns collected. In addition, we examine the clustering of patterns and the different types of relationships between the CPN patterns.

1 Introduction

Process-Aware Information (PAI) systems [16], 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 efficiently build a feasible model with the help of a PAI system (e.g. WFM software), 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 the advantages and disadvantages of possible solutions? Which solution to choose and how to realize the selected solution? These 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 often spend their time solving problems which may already have existing solutions.

A *pattern language* is one of the possible means to help developers to build their models efficiently, while avoiding reinvention of already existing solutions of problems. 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 at hand, a developer can look up a solution for the problem in the pattern catalog, while spending less effort on the development and also ensuring the soundness of a solution.

The work reported in this paper is part of the *Workflow Patterns Initiative* `www.workflowpatterns.com`. In the context of this initiative, we have developed control-flow patterns [6], data patterns [35], and resource patterns [34]. These patterns focus on the different perspectives [24] of PAI systems. In this paper we do not necessarily limit ourselves to workflow or PAI systems. Instead we *focus on the interplay between control flow and data flow*. To do this, we use a specific implementation language: *Colored Petri Nets* (CPNs). In our view, a good understanding of the interplay between control flow and data flow is foundational to PAI systems. Moreover, (colored) Petri nets have shown to be a solid basis for the modeling, analysis, and enactment of workflows [1, 3, 36].[1]

Based on the expert knowledge and an analysis of existing models and literature, we identified 34 patterns that focus on the interplay between control flow and data flow and can be represented in terms of Colored Petri Nets. On the one hand, the patterns we discovered are *implementation patterns*, i.e. they are mainly oriented on model developers who are working with CPN Tools. In particular, the CPN patterns support developers 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 the 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 visualized solutions by means of CPN. Similarly to the 23 design patterns of Gamma [19], the CPN patterns also systematically name, motivate, and explain solutions for generic 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 as implementation patterns.

The remainder of this paper is organized as follows. First, we introduce a set of concepts, central to the CPN patterns and define the scope of the patterns in the context of Petri Nets (Section 2.1). Next, in Section 2.2, we give a brief introduction into the world of patterns, and introduce the pattern format which we selected for the description of the 34 identified patterns. In Section 3, we give an example of a CPN pattern using this format. Then, we introduce the CPN pattern language (Section 4). We not only examine relationships between the discovered patterns to enable easy navigation through the CPN pattern catalog (Section 4.2), but we also classify patterns into clusters in order to simplify the selection of a suitable pattern (Section 4.3). We conclude the paper by discussing related work and future work (Section 5 and Section 6).

[1] Although (colored) Petri nets form a good foundation for workflow languages and PAI systems, one could argue that they are too low level as an end-user language [4]. Therefore, we propose Petri nets as a theoretical basis and use higher-level languages such as [5] as the end-user language.

2 Preliminaries

In this section we briefly introduce colored Petri nets and discuss existing patterns languages.

2.1 Colored Petri Nets

Colored Petri Nets (CPNs) [25, 26] extend the classical Petri Nets [15] with colors (to model data), time (to model durations), and hierarchy (to structure large models). Like in classical Petri Nets, CPNs use three basic concepts: transition, place, and token. We will use the terms "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 "token" or "object" we can refer to any of [22]:

- 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.

Whenever a pattern operates with a specific type of objects, we will specify the type (called color set in CPN) explicitly. For gathering CPN patterns, we concentrate on *discrete dynamic systems*, which are systems with a certain state at 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 workflow management systems, distributed databases, decision support systems, e-mail systems, payment systems, etc. Discrete systems are made out of actors, which are active components, and objects, which are passive components. Actors consume and produce objects [22]. 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 the basic rules of classical Petri nets are still valid in the context of CPNs, we do not elaborate on basic control-flow patterns and focus on the extensions of PN by *color* and *time* (in particular the interplay between control flow and data flow). Hence, we abstract from the extension with hierarchy (e.g., the substitution transitions). Figure 1 visualizes the scope of the pattern language presented in this paper.

There are many variants of colored Petri nets, i.e., Petri net models with color and time. Consider for example the different tools: Design/CPN, CPN Tools, ExSpect, ALPHA/Sim, Artifex, GreatSPN, PEP, Renew, etc. The patterns are tool independent. Nevertheless, we need to select a specific language/tool for the examples used to describe the patterns. For this purpose, we selected *CPN Tools* [14]. The language used by CPN Tools is the de facto standard. Moreover, all

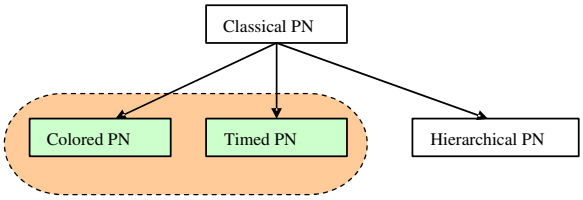

Fig. 1. Scope of CPN patterns: The focus is on color and time while abstracting from hierarchy.

the patterns that we have collected can be downloaded from [28] and executed using CPN Tools. Note that most pattern languages use a specific language to represent examples, for example in [19] both C++ and Smalltalk are used.

2.2 Patterns

Nowadays, there is a generic understanding of what a pattern is, i.e. it is a solution to a problem in a certain context. Note that originally the concept of a pattern was introduced by Christopher Alexander in [9], who wrote:

The pattern is, in short, at the same time a thing, which happens in the world and the rule which tells us how to create that thing, and when we must create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process that will generate this thing.

The notion of a pattern language, introduced by Alexander in [8], is similar to the notion of a language as recorded in the Merriam-Webster Dictionary as a "formal system of signs or symbols including rules for the formation and transformation of admissible expressions". If a word is a central entity of a language, then a pattern is a central entity of the pattern language. Similar to the rules describing the use of words in sentences, patterns also have rules associated with them. As such, pattern rules describe relations between the patterns and indicate how one pattern can be combined with other ones. Furthermore, a pattern language can serve as a systematic means of communicating problems and solutions between colleagues working in the same field/domain.

Since the definition of a pattern by Christopher Alexander, different types of patterns in different application domains have been described. This has resulted in a set of pattern languages each of which addresses different aspects of organization, software development, analysis, etc. Due to the differences in the types of problems and the solution means in different application fields and domains, there is an ongoing discussion concerning the suitability of the *pattern format* introduced by Alexander for documenting the patterns.

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 yet been achieved, we took as a basis the pattern format of Gamma [19], and adjuisted it in order to fit our purposes. Every CPN pattern adheres to the following pattern format:

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 alternatively used names for the *Pattern name*.
- *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 by 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 subsection.
- *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 not be 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 the 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 the possible advantages/disadvantages of using the pattern are. In case if the pattern supplies several solutions, this section elaborates on the differences between them.
- *Examples*. This section lists several examples demonstrating the use of the pattern in practice.
- *Related Patterns*. This section specifies relations of the pattern to other patterns.

3 Example: AGGREGATE OBJECTS Pattern

Before defining our pattern language that also relates patterns to one another, we present one of the 34 patterns. Like all the other patterns, it is described using the pattern format described in the previous section.

Pattern: AGGREGATE OBJECTS
Also Known As:

Intent: to allow the 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 2 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.

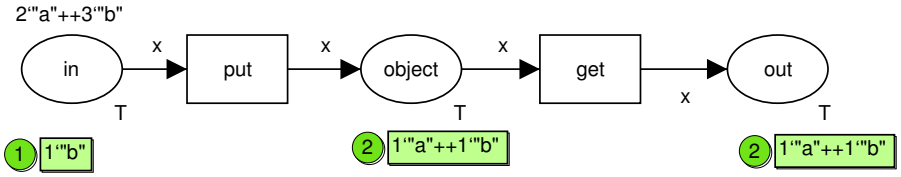

Fig. 2. Example used to explain the various problems.

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 than 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 memoryless 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 the manipulation of a set of information objects as a single entity, aggregate the objects into a single token of "collection type". [2]

[2] Note that we assume an interleaving semantics.

Implementation of Solution: The list of instructions below describes how to implement the AGGREGATE OBJECTS pattern (see Figure 3).

- Modify the type `T` of place `objects`, where multiple objects may reside, to the collection type `LT` (e.g., list, set, bag). 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.

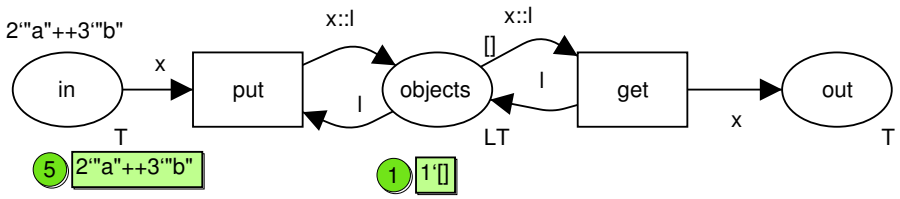

Fig. 3. Model after applying the pattern.

By introducing a collection type, it becomes possible to refer to the collection of objects as a single entity and perform operations on multiple objects contained in the collection at once. Several examples in Figures 4 and 5 show how to implement some operations from the ones mentioned in the Problem description section by extending the net presented in Figure 3.

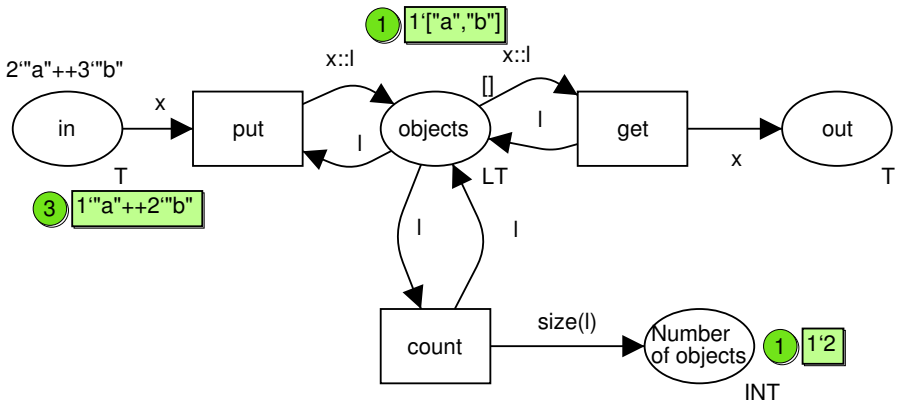

Fig. 4. Example illustrating how place `objects` can now be used.

Figure 4 shows how to calculate the size of the collection, i.e. the number of objects the collection contains. Note that there is always precisely one token in place `objects` representing all objects. 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 can be obtained by the transition `select` as shown in Figure 5.

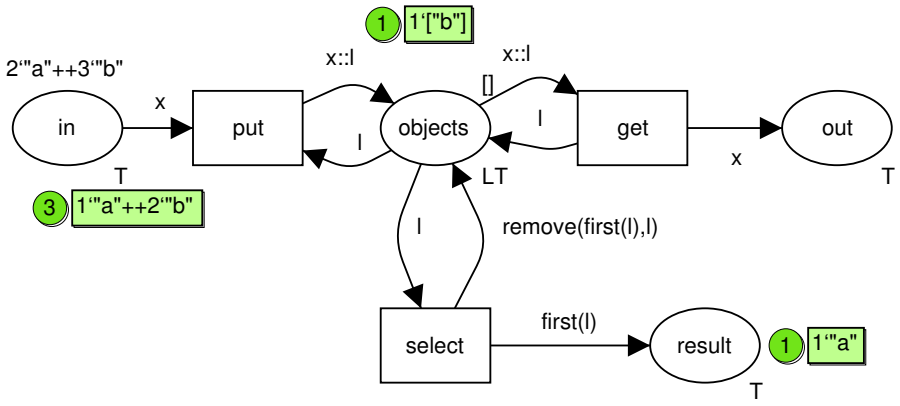

Fig. 5. Another example illustrating how place `objects` can now be used.

Function `first` selects the right object while function remove is used to remove the object, i.e.,

```
fun f(x:T,l:LT) = if l = [] then x else if x< hd(l) then
    f(x,tl(l)) else f(hd(l),tl(l));
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 pattern).[3]

Nevertheless, if the problem of ordering is relevant, one should apply an extension of this pattern by adding 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 the writing of corresponding function(s) from scratch.

Examples:

- The salary administration of a university divided employees into different groups: students, PhD students, and 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 in one-step by increasing the salary of the whole group.
- The documents are collected and organized in one file. This allows the whole file to be taken and sent for processing elsewhere, keeping the documents structured and grouped.

Related Patterns: This pattern is extended by the QUEUE pattern.

4 CPN Pattern Language

In this section, we introduce the CPN pattern language by listing the names and intents of the discovered patterns. Next, we analyze relationships between the CPN patterns, and organize them into a relationship diagram, which allows navigation through the pattern catalog for identifying related patterns. Furthermore, we classify the patterns into categories in order to simplify the process of selecting a pattern from the CPN pattern catalog.

4.1 Overview of CPN Pattern Language

In this section we give an overview of the CPN patterns. In Tables 1 and 2 we present only the names and intents of the patterns; for further details the reader is referred to [27]. Patterns listed may belong to the same classes, and have similarities in their intents, problems and solutions. These relationships between patterns are not covered explicitly in the tables, but are discussed in Section 4.2.

[3] Note that this pattern refers to other patterns like the LIFO QUEUE pattern. These have not yet been discussed. An overview of all 34 patterns in given in Section 4.

Pattern name	**Intent**
ID Matching	to make identical information objects distinguishable
ID Manager	to ensure uniqueness of identifiers used for distinguishing identical objects
Aggregate Objects	to allow manipulation of a set of information objects as a single entity
Queue	to allow manipulation of the queued objects in a strictly specified order
FIFO Queue	to allow manipulation of objects from the collection in a strictly specified order such that an object which arrived first is consumed first
LIFO Queue	to allow manipulation of objects from the collection in a strictly specified order, such that the mostly recently added object is retrieved first
Random Queue	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
Priority Queue	to allow manipulation of objects from the collection in the order of the objects' priority
Capacity-bounding	to prevent over-accumulation of objects in a certain place
Inhibitor Arc	to support "zero"-testing of places
Colored Inhibitor Arc	to support "non-containment" property of places
Shared Database	to enable centralized storage of data shared between multiple transitions, supporting different levels of data visibility (i.e. local, group, or global)
Database Management	to specify the interface of accessing data, stored in a shared database for read-only and modification purposes
Copy Manager	to make data stored in the shared database available at other locations for local use, maintaining the consistency of data in all places
Lock Manager	to synchronize access to shared data by means of exclusive locks
Bi-lock manager	to synchronize access to shared data for reading and writing purposes by means of shared and exclusive locks
Log Manager	to record the information about actual process execution by means of a data log
BSI Filter	to prevent data non-conforming to a certain property from passing through
BSD Filter	to prevent data non-conforming to a property involving the state of an external data-structure, from passing through
NBSI Filter	to filter out data fulfilling a certain property while avoiding accumulation of non-conforming data in the filter input place
NBSD Filter	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
Translator	to enable coordinated communication between two actors with originally different data formats
Asynchronous Transfer	to allow transportation of data from one location to another, while avoiding the sender to block
Synchronous Transfer	to allow transportation of data from one location to another, ensuring that an actor, which posted a request, is blocked until it receives the requested information
Rendezvous	allow multiple actors to broadcast and discover data objects concurrently
Asynchronous Router	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

Table 1. Summary of CPN patterns

Pattern name	Intent
Asynchronous Aggregator	to provide a holistic view of data, produced by multiple unrelated sources through asynchronous data aggregation
Broadcasting	to allow broadcasting of data from a single source to multiple targets, while avoiding direct dependency between them
Redundancy Manager	to prevent transfer of duplicated data between loosely-coupled actors who communicate asynchronously
Data Distributor	to support parallel data processing by distributing data between several independent actors
Data Merger	to compose a single information object out of several smaller ones when all parts required for composition become available
Deterministic XOR-split	to allow at most one transition out of several possible to execute, based on fulfillment of mutually excluding data conditions
Non-deterministic XOR-split	to allow any transition out of several possible, but satisfying the same data condition, to execute
OR	to allow any number of tasks to be selected for execution based on the fulfillment of a certain data condition

Table 2. Summary of CPN patterns (Cont.)

4.2 CPN Pattern Relationships

The 34 CPN patterns that we have identified, together with the relationships between them, form a pattern language. In order to classify the CPN patterns we examined the nature of relationships between the patterns. We used three types of primary relations: *specialization of a problem*, *use in a solution*, and *extension of an implementation*; and two types of secondary relations: *problem similarity*, and *combination of solutions* to describe the pattern relationships. Some of the relationship types are based on Zimmer's classification [39].

The main purpose of this classification is to provide a holistic view on the catalog of patterns, providing a means for a user to select a number of patterns and to determine how the patterns can help in solving a given problem. The selected types of relationships can help to trace other patterns related to a chosen pattern, thus allowing the estimation of an overall problem complexity, the tradeoffs made, and compare the chosen pattern with other similar patterns, in order to select an optimal solution for a problem in the given context.

Figure 6 shows some of the relationships between the various patterns. The graphical representation and the text depict the type of a relationship. To understand the diagram, we first need to define the different types of relationships.

Primary relations

Problem-oriented relation
Pattern A is a *specialization* of the more generic pattern B. Specific pattern A, which deals with a specialization *of the problem* that generic pattern B addresses, has a similar but more specialized solution than pattern B. Pattern A includes

all the properties of pattern B, but adds further restrictions by adding some specialized characteristics. Note that a specialization often adds more context to the problem thus making it less generic.

Solution-oriented relation
Pattern A *uses* pattern B *in its solution*. When building a solution for a 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.[4] All instantiations of pattern A use pattern B. Some example relationships: Lock Manager uses ID Matching, Asynchronous Router uses Asynchronous Transfer.

Solution implementation-oriented relation
Pattern A *syntactically extends* pattern B. Pattern A addresses a set of requirements to have more or slightly different functionality than the pattern B addresses. However, this is the implementation of B, which is syntactically extended by A, rather than a problem or a solution. For example, the implementation of Non-Blocking State-Independent Filter extends implementation of Blocking State-Independent Filter.

Secondary relations

Problem similarity
Pattern A is *similar* to pattern B. Pattern A addresses a *problem similar* to the one addressed in pattern B. Patterns A and B can be considered as alternatives of each other; therefore, one can compare them and select the one which fits the problem best.

Combinable solutions
Pattern A can be *combined* with pattern B. Neither of the patterns is a part of the other. Combining the solution of pattern B with the solution of pattern A can help to solve a more complex problem than a single pattern solves in isolation. Use this relation to find out other patterns, which can be used in addition to pattern A. For example, the Shared Database can be combined with Copy Manager; Asynchronous Aggregator can be combined with Aggregate Objects.

Figure 6 shows the five types of relationships. The listing of the primary relationships is intended to be complete while the secondary relationships depicted only represent typical examples. The solid arrows represent primary relationships while the dashed lines represent secondary relationships. A problem-oriented relation is labelled "is specialization of". A solution-oriented relation is labelled

[4] Since a pattern may have multiple solutions, the relationship may need to refer to a specific solution. For the sake of clarity we use the mnemonics "s1", "s2" and "s3" as identifiers for referring to the first, second, and third solution and to make relations between pattern solutions explicit.

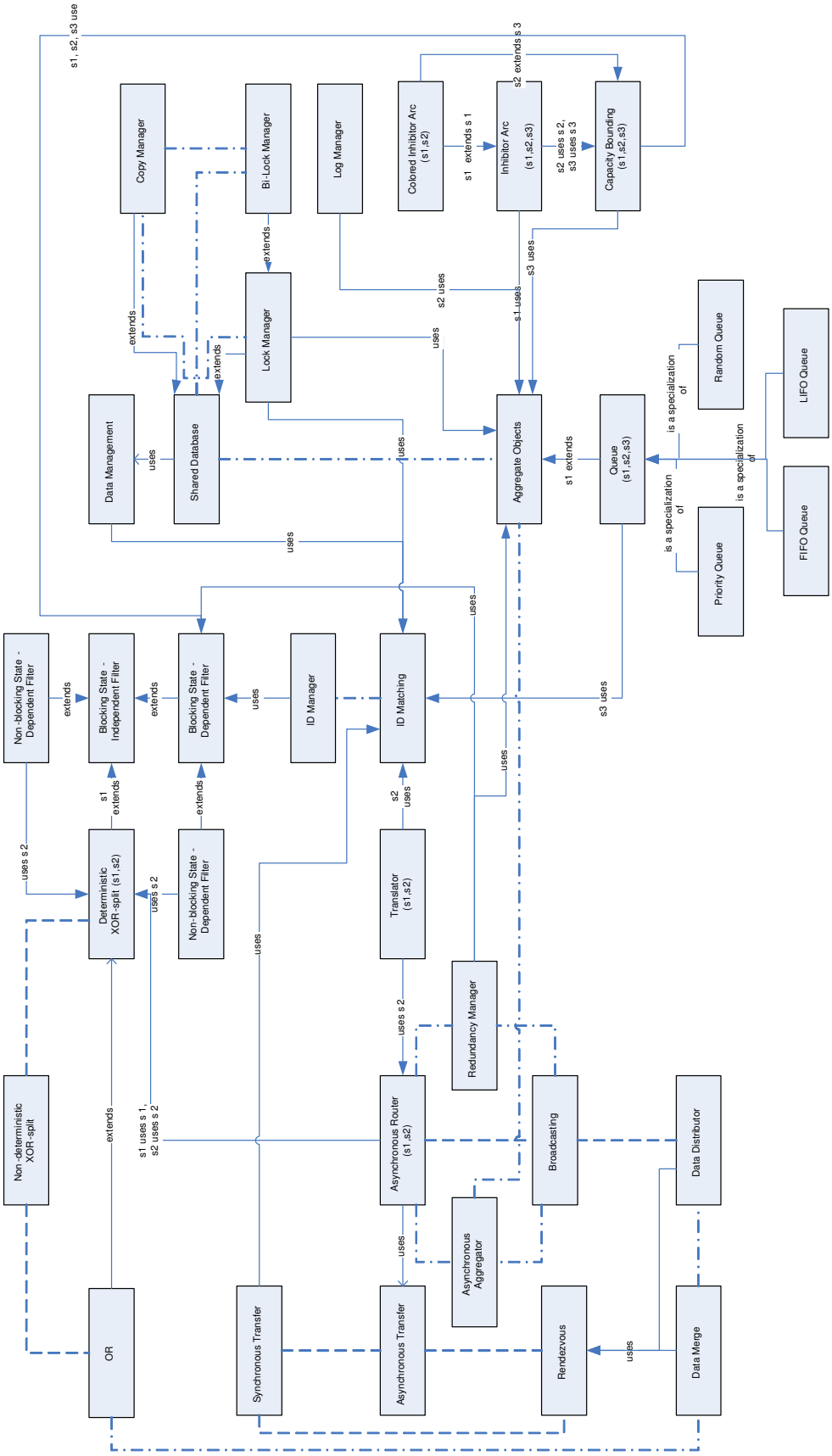

Fig. 6. CPN pattern relationship diagram.

"uses". A solution implementation-oriented relation is labelled "extends". Problem similarity is denoted by dashed lines (without dots) while combinable solutions are denoted by dashed lines with dots. Note that the details regarding the combination of one pattern with another one, or similarities between patterns are not indicated in the relationship diagram, but can be found in the *Consequences* and *Related patterns* sections of a chosen pattern.

In Section 3, we defined the Aggregate Objects pattern. As shown in Figure 6, the Queue pattern extends the first solution of the Aggregate Objects pattern. Moreover, many patterns use the Aggregate Objects pattern (e.g., the Log Manager, Inhibitor Arc, Lock Manager, and Capacity Bounding patterns).

4.3 Classification of CPN Patterns

Although the CPN pattern relationship diagram presented in Figure 6 allows the navigation through the catalog of the CPN patterns, it is not sufficient to classify the patterns precisely and unambiguously.

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

- patterns where the 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 both data perspectives and control-flow perspectives are important and involved.

However, this classification turns out to be not very meaningful and is rather subjective.

In order to provide a more useful means for selecting an appropriate pattern, we adopt the classification presented in [21] to categorize the CPN patterns. This classification is based on the *intent* of each pattern. The intent of every pattern has been analyzed according to a structure where *common components* contain *diagnostic elements*, and in turn diagnostic elements contain *supplementary components*. This structure will be represented using the following format.

Common component
- ◦ Diagnostic component
  - • Supplementary component

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

Supplementary components address additional features for extended definitions of meanings. This components address 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.

Using the nested format described above (i.e., common components, diagnostic elements, supplementary components), we are able to classify the 34 CPN patterns:

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)
    indirectly 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)

The Aggregate Objects pattern defined in Section 3 is classified under common component "Assemble", diagnostic component "Information objects", and supplementary component "by aggregating into a collection". It is interesting to see how the Queue pattern, although it extends Aggregate Objects pattern, is classified completely different. This can be explained by the clear difference in intent.

In this section we briefly introduced the set of 34 patterns. Clearly, we cannot list the patterns in full and instead refer to [28, 27]. Then we showed two ways to compare and relate patterns. We identified three primary and two secondary pattern relationships and classified the patterns using the classification of Hasso and Carlson [21].

5 Related Work

It is impossible to give a complete overview of the different types of patterns described in literature. The 23 design patterns by Gamma et al. [19] triggered the development of many more patterns in the object-oriented software community. Some of it successors include: the patterns for knowledge and software reuse by Sutcliffe [37], the design patterns in communication software by Linda Rising [33], and the framework patterns by Wolfgang Pree [32].

Aside from 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 [29], has been discovered and documented.

Furthermore, some work has been done on formalizing the organization, process, analysis, and business-related patterns. Among them are the analysis patterns by Martin Fowler [18], the enterprise architecture patterns by Michael Beedle [12], the framework process patterns by James Carey [13], the patterns for e-business [7] (which focus on Business patterns, Integration patterns, and Application patterns), the business patterns at work [17] (which use UML to model a business system), and the process patterns [10]. Other interesting patterns collections focusing on the process-side of things are the enterprise integration patterns by Hophe and Woolf [23] and the service interaction patterns by Barros et al. [11].

However, the real starting point for this work has been the Workflow Patterns Initiative (cf. `www.workflowpatterns.com`). To capture the functionality of PAIS in term of patterns, control-flow patterns [6], data patterns [35], and resource patterns [34] have been uncovered. We consider the CPN patterns foundational for the further development of this initiative.

In the context of Petri nets some initial attempts to capture patterns have been made. Earlier work by Kurt Jensen [25], Wil van der Aalst [2], and Kees van Hee [22], provides some patterns in an implicit and/or fragmented manner. In [31], Petri nets are used to represent workflow and communication patterns in the context of webservices. One of the few papers, linking CPN to patterns is [30]. However, here CPNs are merely used as an underlying representation of the dynamic object-oriented architecture and the real focus is on patterns found in concurrent software designs. The paper that is probably most related to our work is [20] by Matthias Griess et al. They define 3 patterns in terms of classical Petri nets using pattern language similar to ours. For a given example they analyze the use of these patterns.

Our work differs from these papers in at least two ways. First of all, we use CPNs (rather than classical nets) and focus on the interplay between control flow and data flow. Second, our set of patterns is more mature as is illustrated by the number of patterns, classification, and relationships.

6 Conclusion and Future Work

In this paper, we described a pattern language for CPNs. We collected a set of 34 patterns focusing on the interplay between control flow and data flow. Although

expressed in a specific language, the patterns can be applied to model and design of any kind of dynamic systems with elements of data and concurrency.

The language and the patterns have been developed in an explorative manner. This means that we applied empirical methods to gather information, such as observation, content analysis, and simulation. In order to discover patterns we used application models, publications, tutorials, workshop materials, opinion of experts, and feedback from model developers. We applied the content analysis technique for extracting the patterns from the literature sources, and verified the correctness of models, which represent solutions for certain problems, by simulating them in CPN Tools.

We do not claim that the CPN patterns we gathered are complete, since they are the result of explorative work and were not derived in a systematic manner. We have made the implementations of CPN patterns available to the CPN community in the form of a pattern library [28]. We want to encourage members of the CPN community to extend the catalog of patterns by including the ones not covered here. Moreover, these patterns can serve as a language enhancing communication between developers, allowing to communicate problems and solutions unambiguously.

Further research will include the development of a patterns repository and empirical research into the actual use of patterns. We have developed a prototype patterns repository allowing people to navigate patterns based on various relationships. This needs to be improved to be really useful. The empirical research into the actual use could involve the analysis of student projects and/or the analysis of papers describing CPN models. In [20] this approach was used for a single example and a small set of basic Petri net patterns. In [38] a similar approach was applied to workflow projects of Atos Origin using the workflow patterns [6].

Acknowledgments

We would like to thank Kurt Jensen for contributing to the work reported in this paper. His experience in modeling using Colored Petri Nets has been vital for collecting and describing the patterns presented. We also thank Maurice Hendrix and Alex Norta for working on the initial prototype of the patterns repository.

References

1. W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
2. W.M.P. van der Aalst. *Process Modeling, Lecture Notes*. Eindhoven University of Technology, Eindhoven, The Netherlands, 2003.
3. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT press, Cambridge, MA, 2002.
4. W.M.P. van der Aalst and A.H.M. ter Hofstede. Workflow Patterns: On the Expressive Power of (Petri-net-based) Workflow Languages. In K. Jensen, editor,

Proceedings of the Fourth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2002), volume 560 of *DAIMI*, pages 1–20, Aarhus, Denmark, August 2002. University of Aarhus.
5. W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.
6. 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.
7. J. Adams, S. Koushik, G. Vasudeva, and G. Galambos. *Patterns for e-Business. A Strategy for Use*. IBM Press, 2001.
8. C. Alexander. *A Pattern Language: Towns, Building and Construction*. Oxford University Press, 1977.
9. C. Alexander. *Timeless Way of Building*. Oxford University Press, 1979.
10. S.W. Ambler. *Process Patterns*. Cambridge University Press, 1998.
11. A. Barros, M. Dumas, and A.H.M. ter Hofstede. Service Interaction Patterns: Towards a Reference Framework for Service-based Business Process Interconnection. QUT Technical report, FIT-TR-2005-012, Queensland University of Technology, Brisbane, 2005.
12. M.A. Beedle. *Enterprise Architecture Patterns*. Cambridge University Press, 1998.
13. J. Carey and B. Carlson. *Framework Process Patterns*. Addison Wesley Longman, 2001.
14. CPN Group, University of Aarhus, Denmark. CPN Tools Home Page. http://wiki.daimi.au.dk/cpntools/.
15. J. Desel, W. Reisig, and G. Rozenberg, editors. *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2004.
16. M. Dumas, W.M.P. van der Aalst, and A.H.M. ter Hofstede. *Process-Aware Information Systems*. Wiley & Sons, 2005.
17. H. Eriksson and M. Penker. *Business Modeling with UML. Business Patterns at Work*. Wiley, John and Sons, 1998.
18. M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, Reading, Massachusetts, 1997.
19. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison Wesley, Reading, MA, USA, 1995.
20. M. Gries, J.W. Janneck, and M. Naedele. Reusing Design Experience for Petri Nets Through Patterns. In *Proceedings of High Performance Computing HPC'99*, pages 453–458, San Diego, CA, USA, 1999.
21. S. Hasso and C.R. Carlson. Linguistics-based Software Design Patterns Classification. In *Proceedings of the Thirty-Seventh Annual Hawaii International Conference on System Science (HICSS-37)*. IEEE Computer Society Press, 2004.
22. K.M. van Hee. *Information System Engineering: a Formal Approach*. Cambridge University Press, 1994.
23. G. Hohpe and B. Woolf. *Enterprise Integration Patterns*. Addison-Wesley Professional, Reading, MA, 2003.
24. S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture, and Implementation*. International Thomson Computer Press, London, UK, 1996.
25. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1*. EATCS monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1997.

26. L.M. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2):98–132, 1998.
27. N. Mulyar and W.M.P. van der Aalst. Patterns in Colored Petri Nets. BETA Working Paper Series, WP 139, Eindhoven University of Technology, Eindhoven, 2005.
28. N. Mulyar. CPN Patterns Home Page. http://is.tm.tue.nl/staff/nmulyar.
29. Pattern digest library. http://patterndigest.com/books/otherlang.jsp.
30. R.G. Pettit and H. Gomaa. Modeling Behavioral Patterns of Concurrent Software Architectures Using Petri Nets. In *WICSA '04: Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, page 57, Washington, DC, USA, 2004. IEEE Computer Society.
31. S.K. Prasad and J. Balasooriya. Fundamental Capabilities of Web Coordination Bonds: Modeling Petri Nets and Expressing Workflow and Communication Patterns over Web Services. In *HICSS '05: Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05) - Track 7*, page 165.2, Washington, DC, USA, 2005. IEEE Computer Society.
32. W. Pree. *Framework patterns*. SIGS Books, 1996.
33. L. Rising. *Design Patterns in Communication Software*. Cambridge University Press, 2000.
34. N. Russell, W.M.P.van der Aalst, A.H.M. ter Hofstede, and D. Edmond. Workflow Resource Patterns: Identification, Representation and Tool Support. In O. Pastor and J. Falcao e Cunha, editors, *Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05)*, volume 3520 of *Lecture Notes in Computer Science*, pages 216–232. Springer-Verlag, Berlin, 2005.
35. 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.
36. Software-Ley. *COSA 3.0 User Manual*. Software-Ley GmbH, Pullheim, Germany, 1999.
37. A. Sutcliffe. *Patterns for Knowledge and Software Reuse*. Lawrence Erlbaum Associates Inc., 2002.
38. K. de Vries and O. Ommert. Advanced Workflow Patterns in Practice (1): Experiences Based on Pension Processing (in Dutch). *Business Process Magazine*, 7(6):15–18, 2001.
39. W. Zimmer. Relationships between Design Patterns. In *Pattern languages of program design*, pages 345–364, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.