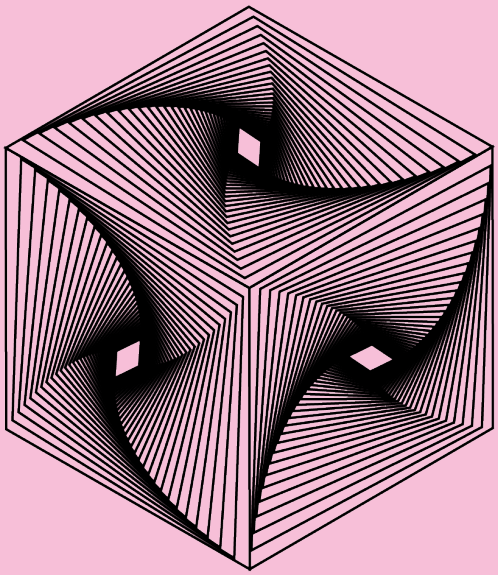
The Mathematical Specification of the Statebox Language



The Statebox Team

statebox.org

July 16, 2018

Contents

1 Introduction **3**
- 1.1 What to expect . . . 3
- 1.2 Prerequisites . . . 4
- 1.3 Synopsis . . . 4

I First concepts **6**

2 Petri nets **7**
- 2.1 Petri nets, informally . . . 7
- 2.2 Multisets . . . 9
  - 2.2.1 Operations on multisets . . . 10
- 2.3 Petri nets, formally . . . 11
  - 2.3.1 Markings, enabled transitions . . . 12
  - 2.3.2 Firing semantics for Petri nets . . . 12
- 2.4 Examples . . . 13
- 2.5 Further properties of Petri nets . . . 14
  - 2.5.1 Reachability, safeness and deadlocks . . . 14
- 2.6 Why is this useful? . . . 16

3 Introduction to category theory **18**
- 3.1 What is category theory? . . . 18
- 3.2 Functors, natural transformations, natural isomorphisms . . . 21
- 3.3 Monoidal categories . . . 24
- 3.4 String diagrams . . . 27
- 3.5 Monoidal functors . . . 29
- 3.6 Products, coproducs, pushouts . . . 31
- 3.7 Why is this useful? . . . 34

4 Executions of Petri nets **36**
- 4.1 Problem overview . . . 36
- 4.2 The category $\mathbf{Petri}^{\mathbb{N}}$ . . . 38
- 4.3 The Execution of a net . . . 40
- 4.4 The category $\mathfrak{F}(N)$ . . . 42
- 4.5 Functors between executions . . . 45
- 4.6 Beyond standard Petri nets . . . 47
- 4.7 Why is this useful? . . . 51

5 Folds **52**
5.1 Problem Overview . . . 52
5.2 Mapping executions . . . 53
5.3 Different folds, shared types . . . 56
5.4 Why is this useful? . . . 57

II Extending nets 58

6 Open Petri nets **59**
6.1 Open nets, first definitions . . . 59
6.2 Static operations on open nets . . . 62
6.2.1 Sequential composition of open nets . . . 62
6.2.2 Parallel composition of open nets . . . 65
6.2.3 Hiding of open Petri nets . . . 68
6.3 Runtime operations on open nets . . . 69
6.3.1 Quotienting . . . 70
6.3.2 Examples . . . 73
6.4 Relationship between static and runtime operations . . . 75
6.5 Why is this useful? . . . 76

Chapter 1

Introduction

This document defines the mathematical backbone of the Statebox language. In the simplest way possible, Statebox can be seen as a clever way to tie together different theoretical structures to maximize their benefits and limit their downsides. Since consistency and correctness are central requisites for our language, it became pretty clear from the beginning that such tying could not be achieved by just hacking together different pieces of code representing implementations of the structures we wanted to leverage: Some legit mathematics had to be employed to ensure both conceptual consistency of the language and reliability of the code itself. The mathematics presented here is what guided the implementation process, and we deemed very useful to release it to the public to help people wanting to audit our work to better understand the code itself.

1.1 What to expect

This document is a work in progess, and will be released together with each version of the Statebox language, suitably expanded to cover the new features we will gradually implement. Each version of it will contain more theoretical material than what will actually be implemented in the Statebox version it comes together with. This is intended, and serves the purpose of helping the audience understand what we are working on, and what to expect from the upcoming releases.

In this document there will be very little code involved, and quite a lot of mathematics. The maths will always be introduced together with intuitive explanations meant to clarify the ideas we are trying to formalize. Notice that here we care more about giving the bigger picture of the language itself and will focus on technical details only when strictly needed. There are a number of seminal papers that explain, with a great deal of precision, some of the theoretical material that we are employing to implement the Statebox language, and we will constantly refer the reader to them for details. When the material covered is genuinely new, details can be found in papers we published ourselves in peer reviewed venues, as for [11]. Again, in this case we will reference the audience to our own contributions for a thorough presentation of the concepts covered. The reader should then read this document as a high-level presentation of the concepts we are using and how they interact together, and should follow the references provided to understand the technicalities. All in all:

- The audience with a strong background in theoretical computer science can use this document to understand how we plan to use results in different research fields to create a new programming language, and how do we achieve consistent interaction between them, especially when they are expressed using very different formalisms. An exhaustive explanation of the concepts presented, if needed, will be found in the bibliographic references;
- The inexperienced reader will be able to understand the content of cutting-edge research that

would be otherwise difficult or impossible to access directly. Hopefully, reading this document will make the reader's attempt to read the papers firsthand easier - if they choose to do so.

It is also worth stressing that we did our best to keep the bibliography to a bare minimum, to help the reader who wants to dig deeper by focusing on a few, selected resources. In particular, when possible, we relied on works which are considered the standard reference in their field, as in the case of [18] for category theory.

1.2 Prerequisites

The Statebox team did their best to make this document as accessible as possible. This clearly required finding a trade-off between exhaustive presentation and conceptual accessibility. In general we assume very little previous knowledge. Our ideal reader knows some basic set theory and how to manipulate equalities – and, at least in principle, understands how coding works. This doesn't mean that it is necessary to be a programmer to understand this document. Everything we require is just having a vague idea of how, conceptually, humans instruct machines about how to perform tasks. This said, an inclination toward logical thinking and approaching problems rationally and in a pragmatic way is surely needed to understand this work properly.

Throughout the document, we will often make remarks and examples intended for a more experienced audience. These are marked with an *asterism superscript* (like this*) and can be safely ignored without undermining the general comprehension of the concepts exposed if too difficult to grasp.

We moreover tried as much as possible to stick to common mathematical notation to avoid any kind of discomfort, making exceptions only when ambiguity could arise.

1.3 Synopsis

We conclude this short introduction by presenting a synopsis of what we are going to do in each Chapter of this document. As we already said, this document is a work in progress, and its synopsis will be changed accordingly as the amount of released material grows.

- In Chapter 2 we will introduce Petri nets, one of the fundamental ingredients in our language. The emphasis in this chapter falls on why Petri nets make a great graphical tool to reason about complex infrastructure before giving formal definitions. We will moreover describe some of the most interesting properties that nets can have, and why it is important to study them;
- In Chapter 3 we will introduce category theory, the mathematical theory that will allow us to find a common ground to tie Petri nets with other theoretical structures, along with its corresponding diagrammatic formalism. This will ultimately enable us to export Petri nets from the realm of theoretical research to true software engineering, turning them into a great way of designing complex code while guaranteeing consistency and reliability. The diagrammatic formalism will serve the purpose of having the strength of mathematical reasoning backed up by visual, intuitive representation of concepts;
- In Chapter 4 we will give a first "categorification" of nets, expressing some of the concepts covered in Chapter 2 using category theory. We will show how this allows us to use Petri nets in a much more powerful way and to fine-tune our reasoning about them, for instance by allowing us to track the whole history of a token in a net. This will give us the needed tools to see nets as deterministic objects by defining their categories of executions, which is a fundamental step to make the implementation of Petri nets useful.

- In Chapter 5 we will elaborate on the results of Chapter 4, showing how we can map Petri nets to other programming languages to produce actual software in a conceptually layered fashion. This is achieved by a functorial mapping from net executions to semantic categories of functional programming languages, allowing us to achieve a separation between software topology and software meaning;
- In Chapter 6 we will extend our definition of nets to achieve the expressiveness we need to produce something meaningful. The nets defined in this chapter, that will be simply called *open nets*, will form the basis on which to build our infrastructure;

The language specification, which is mathematically documented by this work, can be found on the Statebox repository on Github. Links will be provided on our website.

Part I

First concepts

Chapter 2

Petri nets

Petri nets were invented by Carl Adam Petri in 1939 to model chemical reactions [22]. In the subsequent years, they have met incredible success, especially in computer science, to study and model distributed/concurrent systems [21], [23]. In this chapter, we will start explaining what a Petri net is, and why we chose this structure to be at the very core of Statebox.

We will start with an informal introduction, relying on the graphical formalism of nets to present concepts in an intuitive way. Then we will proceed by formalizing everything in mathematical terms. Finally, we will define some useful properties of nets which we will be interested in studying later on.

2.1 Petri nets, informally

A *Petri net* is composed of *places*, *transitions* and *arcs weighted on the natural numbers*. Any place contains a given number of *tokens*, which represent resources. Transitions are connected to places through the arcs, and can turn resources into other resources: A transition can *fire*, consuming tokens living in places connected to its input, and producing tokens living in places connected to its output. An example of a Petri net is shown in Figure 2.1, where:

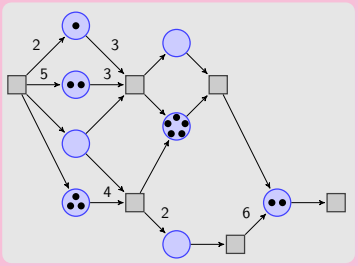


Figure 2.1: Example of a Petri net.

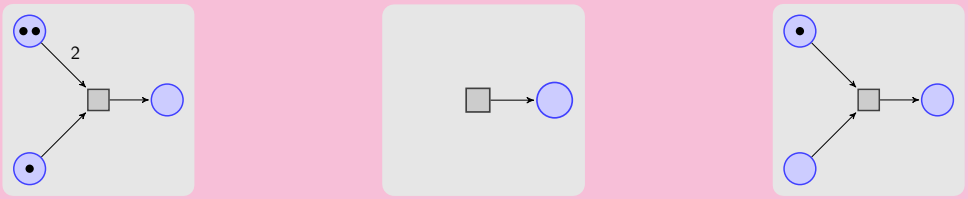
- Places are represented by blue circles;
- Tokens are represented by black dots in each circle;
- Transitions are represented by gray rectangles;
- A weighted directed arc going *from a place to a transition* represents the transition input; the weight signifies the number of consumed tokens. To avoid clutter, we omit the weights when they are equal to 1;

- A weighted directed arc going *from a transition to a place* represents the transition output; the weight signifies the number of produced tokens. To avoid clutter, we omit the weights when they are equal to 1.

A Petri net should be thought of as representing some sort of system. Tokens are resources, and places are containers that hold resources of a given type. Transitions are processes that convert resources from one type to another. Weights on the arcs identify how many resources of some kind a process needs to be executed, and how many resources of some other kind will be produced when the process finishes. With respect to this, we say that a transition can be in two states:

Enabled, if, in *all* the places having edges towards the transition, there is a number of tokens equal to the weight of the edge itself (see Figure 2.2a). Note that if a transition has no inbound edges (as in Figure 2.2b), then it is always considered enabled;

Disabled, otherwise (see Figure 2.2c).



(a) Enabled transition. (b) Trivially enabled transition. (c) Disabled transition.

Figure 2.2: Example of enabled and disabled Petri nets.

When a transition is enabled, then we say that it may *fire*. Firing represents the act of executing the process the transition represents. When a transition fires, a number of tokens are *removed* from each input place, according to the arc weight, and similarly a number of tokens are *added* to each output place, again according to the arc weight. Figure 2.3 shows an enabled transition before (left) and after (right) firing. As you can see in the Figure, we highlight the firing transitions in a net with a black triangle.

Figure 2.3: An enabled transition before (left) and after (right) firing.

Remark 2.1.1 (Generalized nets)**.** Note that the behavior of Petri nets can be generalized much further than this, for example by annotating the arcs with logical conditions that have to be satisfied to consider a transition enabled, or introducing transitions that – a bit counterintuitively – fire only when there are no tokens in one of their input places. *For Statebox, though, we will stick to the rules we defined above, since they are expressive enough to define everything we need.* Working in a greater degree of generality, in fact, makes much more difficult – or even impossible – to answer questions pertaining reachability and absence/presence of deadlocks, which are important concepts that will be formally introduced later. In Statebox, the fundamental requirement is that we should always be able to tell what's going on in our processess, viz. to answer those questions. This is the reason why we do prefer working with a restricted set of rules. We will briefly consider generalizations of nets towards the end of Chapter 4.

2.2 Multisets

The first concrete goal of this Chapter is to state the intuitive concepts presented above in mathematical terms. Before we can introduce Petri nets formally, we need a way to formalize *multisets*. Intuitively, a multiset is just a *set with repetition*, meaning that each element is allowed to occur multiple times in the same set. To make things easier to understand, consider the following writings:

$$\{a, b, d, e, k\} \qquad \{a, b, b, d, e, e, e, k\} \qquad \{a, b, b, d, e, e, k, k\} \tag{2.2.1}$$

When seen as sets, the ones above denote the same thing, since sets "do not distinguish" between repeated elements. The reason why we are interested in the concept of a multiset is precisely because, in our case, we want to be able to consider the three sets above as distinct. The experienced reader will have already noticed how the need for multisets naturally arises when dealing with Petri nets. Specifically, multisets will be useful in:

- Describing the transitions of a Petri net, since we can represent how many tokens a transition consumes (produces) from (in) a place as the number of occurrences of that place in a multiset;
- Describing the state of a Petri net, since we can represent the number of tokens in each place as the number of occurrences of that place in a multiset.

Without further ado, let us introduce the first mathematical definition of this document.

Definition 2.2.1 (Multiset)**.** *A* multiset on X *is a function* $X^{\mathbb{N}} : X \to \mathbb{N}$*, where* X *is a set. A multiset is called* finite *when there is only a finite number of* $x \in X$ *such that* $X^{\mathbb{N}}(x) > 0$*. Finite multisets will usually be denoted with a* $\mathbb{N}$ *used as superscript. For instance,* $X^{\mathbb{N}}$ *represents a multiset on* X*.*

Remark 2.2.2 (Non-finite multisets)**.** In this work, we will be only interested in finite multisets. To avoid clutter, *we will refer to finite multisets just as multisets*.

Example 2.2.3 (Multisets are functions)**.** As we said, multisets have to be interpreted as sets where the same element can be repeated a finite number of times. If we go back to the sets displayed in Equation 2.2.1, we readily see how these can indeed be expressed as functions $f, g, h : \{a, b, d, e, k\} \to \mathbb{N}$, taking values:

$f(a) = 1$	$f(b) = 1$	$f(d) = 1$	$f(e) = 1$	$f(k) = 1$
$g(a) = 1$	$g(b) = 2$	$g(d) = 1$	$g(e) = 3$	$g(k) = 1$
$h(a) = 1$	$h(b) = 2$	$h(d) = 1$	$h(e) = 2$	$h(k) = 2$

Where f is the function representing the first multiset, g the function representing the second, and h the function representing the third, respectively.

Remark 2.2.4 (Same multiset, different functions)**.** Notice that our definition of $\mathbb{N}$ includes 0, and hence if we have a function $g' : \{a, b, c, d, e, k\} \to \mathbb{N}$ defined as:

$$g'(a) = 1 \qquad g'(b) = 2 \qquad g'(c) = 0 \qquad g'(d) = 1 \qquad g'(e) = 3 \qquad g'(k) = 1$$

This also defines the multiset $\{a, b, b, d, e, e, e, k\}$, like g. Also note that subsets of X correspond to functions $f : X \to \{0, 1\}$, and can thus be seen as particular multisets on X where each element is mapped to 0 or 1.

Definition 2.2.5 (Set of multisets over X)**.** $\mathscr{M}_X^{\mathbb{N}}$ *denotes the set of all possible multisets over* X*, that is,*

$$\mathscr{M}_X^{\mathbb{N}} := \{f : X \to \mathbb{N},\ f \mid f(x) > 0 \text{ for a finite number of } x \in X\}$$

2.2.1 Operations on multisets

To be able to proficiently use multisets to formalize Petri nets, we need to understand what we can do with them. Given two multisets $X_1^{\mathbb{N}}, X_2^{\mathbb{N}}$ on X, we can generalize many operations from sets to multisets, as inclusion, union and difference using point-wise definitions.

Definition 2.2.6 (Operations on multisets)**.** *Let $X_1^{\mathbb{N}}, X_2^{\mathbb{N}} \in \mathscr{M}_X^{\mathbb{N}}$. Set* inclusion *generalizes easily setting, for all $x \in X$,*

$$X_1^{\mathbb{N}} \subseteq X_2^{\mathbb{N}} := X_1^{\mathbb{N}}(x) \leq X_2^{\mathbb{N}}(x)$$

Similarly, union *can be generalized to multisets $X_1^{\mathbb{N}}$ and $X_2^{\mathbb{N}}$, setting:*

$$\begin{aligned} &\cup : \mathscr{M}_X^{\mathbb{N}} \times \mathscr{M}_X^{\mathbb{N}} \to \mathscr{M}_X^{\mathbb{N}} \\ &(X_1^{\mathbb{N}} \cup X_2^{\mathbb{N}})(x) := X_1^{\mathbb{N}}(x) + X_2^{\mathbb{N}}(x) \end{aligned} \tag{2.2.2}$$

When $X_1^{\mathbb{N}} \subseteq X_2^{\mathbb{N}}$, we can moreover define their multiset difference, *that unsurprisingly is just:*

$$\begin{aligned} &- : \mathscr{M}_X^{\mathbb{N}} \times \mathscr{M}_X^{\mathbb{N}} \to \mathscr{M}_X^{\mathbb{N}} \\ &(X_1^{\mathbb{N}} - X_2^{\mathbb{N}})(x) := X_1^{\mathbb{N}}(x) - X_2^{\mathbb{N}}(x) \end{aligned}$$

Another intuitive operation that can be defined on multisets is the one of scalar multiplication, *that is similar in concept to scalar products for vector spaces. For each $n \in \mathbb{N}$ and $x \in X$, we set:*

$$\begin{aligned} &\cdot : \mathbb{N} \times \mathscr{M}_X^{\mathbb{N}} \to \mathscr{M}_X^{\mathbb{N}} \\ &(n \cdot X_1^{\mathbb{N}})(x) := n\, X_1^{\mathbb{N}}(x) \end{aligned}$$

Denoting with $X \sqcup Y$ the disjoint union of sets, that we recall being defined as:

$$X \sqcup Y := \{(x,0) \mid x \in X\} \cup \{(y,1) \mid y \in Y\}$$

We can moreover define the analogous disjoint union of multisets, *setting for all $z \in X \sqcup Y$:*

$$\begin{aligned} &\sqcup : \mathscr{M}_X^{\mathbb{N}} \times \mathscr{M}_Y^{\mathbb{N}} \to \mathscr{M}_{X \sqcup Y}^{\mathbb{N}} \\ &(X^{\mathbb{N}} + Y^{\mathbb{N}})(z) := \begin{cases} X^{\mathbb{N}}(x) \text{ iff } z = (x,0) \\ Y^{\mathbb{N}}(y) \text{ iff } z = (y,1) \end{cases} \end{aligned}$$

Finally, for each set X we denote with $\emptyset_X$ the multiset in $\mathscr{M}_X^{\mathbb{N}}$ with the following property:

$$\forall x \in X,\ \emptyset_X(x) = 0$$

Remark* 2.2.7 (Multisets are free commutative monoids)**.** The reader fluent in algebra will have noticed that multiset union defined an operation in the algebraic sense, that makes $\mathscr{M}_X^{\mathbb{N}}$, for each X, *the free commutative monoid generated by X*, where the unit is the multiset $\emptyset_X$.

Remark* 2.2.8 (Injections of multisets)**.** The multiset $\emptyset_X$ can be used cleverly to inject $\mathscr{M}_Y^{\mathbb{N}}$ into $\mathscr{M}_{X \sqcup Y}^{\mathbb{N}}$, as follows:

$$\begin{aligned} &\mathscr{M}_Y^{\mathbb{N}} \hookrightarrow \mathscr{M}_{X \sqcup Y}^{\mathbb{N}} \\ &Y^{\mathbb{N}} \mapsto \emptyset_X \sqcup Y^{\mathbb{N}} \end{aligned}$$

The set X may be embedded into $\mathscr{M}_X^{\mathbb{N}}$ via a function $\delta : X \to \mathscr{M}_X^{\mathbb{N}}$, defined as:

$$\delta_x(y) := \begin{cases} 1, \text{ iff } x = y \\ 0, \text{ iff } x \neq y \end{cases}$$

Given a function $f : X \to \mathscr{M}_Y^{\mathbb{N}}$, we can abuse notation and consider f as a function of multisets $\mathscr{M}_X^{\mathbb{N}} \to \mathscr{M}_Y^{\mathbb{N}}$, by defining

$$f : X^{\mathbb{N}} \in \mathscr{M}_X^{\mathbb{N}} \mapsto \bigcup_{x \in X} X^{\mathbb{N}}(x) \cdot f(x) \in \mathscr{M}_Y^{\mathbb{N}}$$

2.3 Petri nets, formally

Now that we have some intuition about how Petri nets work and have introduced multisets, it is time to define Petri nets formally.

Definition 2.3.1 (Petri net)**.** *A* Petri net *is a quadruple*

$$N := (P_N, T_N, {}^\circ(-)_N, (-)^\circ_N)$$

Where:

- P_N *is a* finite *set, representing places;*
- T_N *is a* finite *set, representing transitions;*
- P_N *and* T_N *are disjoint: Nothing can be a transition and a place at the same time;*
- ${}^\circ(-)_N : T \to \mathscr{M}_{P_N}^{\mathbb{N}}$ *is a function assigning to each transition the multiset of* P *representing its* input *places;*
- $(-)^\circ_N : T \to \mathscr{M}_{P_N}^{\mathbb{N}}$ *is a function assigning to each transition the multiset of* P *representing its* output *places.*

We will often denote with $T_N, P_N, {}^\circ(-)_N, (-)^\circ_N$ *the set of places, transitions and input/output functions of the net* N*, respectively.*

Example 2.3.2 (Input and output places)**.** In Figure 2.4 we highlighted the action of ${}^\circ(t)_N$ in red for two different transitions, denoted with t. We did the same for $(t)^\circ_N$, highlighted in green.

Figure 2.4: Examples of input/output places of transitions.

Remark* 2.3.3 (Generalized input and output)**.** Given a Petri net N, we can generalize ${}^\circ(-)_N$ and $(-)^\circ_N$ to functions of multisets $\mathscr{M}_{T_N}^{\mathbb{N}} \to \mathscr{M}_{P_N}^{\mathbb{N}}$ using the procedure explained in Remark 2.2.8, that is, we can extend them so that they act on multisets of transitions, as follows:

$${}^\circ(-)_N : U^{\mathbb{N}} \in \mathscr{M}_{T_N}^{\mathbb{N}} \mapsto \bigcup_{t \in T_N} U^{\mathbb{N}}(t) \cdot {}^\circ(t)_N \in \mathscr{M}_{P_N}^{\mathbb{N}}$$

$$(-)^\circ_N : U^{\mathbb{N}} \in \mathscr{M}_{T_N}^{\mathbb{N}} \mapsto \bigcup_{t \in T_N} U^{\mathbb{N}}(t) \cdot (t)^\circ_N \in \mathscr{M}_{P_N}^{\mathbb{N}}$$

2.3.1 Markings, enabled transitions

Up to now, we still didn't formalize the concept of a marking. At the moment, our Petri nets are empty, meaning that we don't have a way to populate places with tokens. This can be readily expressed using multisets again.

Definition 2.3.4 (Marking)**.** *Given a Petri net N, a* marking *(also called a* state*) for N is a multiset on P_N, $X^{\mathbb{N}} : P_N \to \mathbb{N}$.*

The interpretation is that the marking assigns a finite, positive or zero number of tokens to each place of N. We denote that a net N comes endowed with a marking $X^{\mathbb{N}}$ using the notation $N^{\mathbb{N}}_X$. Equivalently, we can also say that N is in the *state* $X^{\mathbb{N}}$ to refer to $N^{\mathbb{N}}_X$.

Having formalized the concept of a marking, we can now take care of defining the dynamics of a Petri net.

Definition 2.3.5 (Enabled transition)**.** *Given a Petri net N in the state $X^{\mathbb{N}}$, we say that a transition $t \in T_N$ is* enabled *if*

$$^\circ(t)_N \leq X^{\mathbb{N}}$$

Note that since we are working with multisets, this is equivalent to

$$\forall p \in P_N,\ ^\circ(t)_N(p) \leq X^{\mathbb{N}}(p)$$

meaning, as we would expect, that a transition is enabled if and only if in any input place for t there are at least as many tokens available as t will have to consume.

Remark* 2.3.6 (Enabled check)**.** $\mathscr{M}^{\mathbb{N}}_{P_N}$ denotes the set of all possible multisets over P_N. For a net N we can define a function

$$\overline{(-)}_{(-)} : T_N \times \mathscr{M}^{\mathbb{N}}_{P_N} \to \{\top, \bot\}$$

that takes a transition t and a marking $X^{\mathbb{N}}$ as input and returns $\top$ if t is enabled in $X^{\mathbb{N}}$, and $\bot$ otherwise. This function can be generalized to sets of transitions $U \subseteq T_N$ by setting $\overline{U}^{\mathbb{N}}_X := \bigwedge_{t \in U} \overline{t}_M$, where $\bigwedge$ denotes the usual logical conjunction of predicates. This function is important from an implementation point of view as it allows for an efficient way to determine if a given transition can fire in a given state.

2.3.2 Firing semantics for Petri nets

Now, we have to define a *firing policy*, also called *firing semantics* by mathematically formalizing what happens when a transition fires. Given a Petri net P in the state $X^{\mathbb{N}}$, the firing of a transition t should have two properties:

- t should be able to fire only when enabled
- Firing t should consume some tokens and produce others, thus changing the state of P from $X^{\mathbb{N}}$ to some other marking $Y^{\mathbb{N}}$. We will indicate this using the notation $N^{\mathbb{N}}_X \xrightarrow{t} N^{\mathbb{N}}_Y$.

These two requirements can be captured by the following definition.

Definition 2.3.7 (Firing rule)**.** *Let N be a Petri net in a state $X^{\mathbb{N}}$, and let $t \in T_N$. We define:*

$$N^{\mathbb{N}}_X \xrightarrow{t} N^{\mathbb{N}}_Y := {}^\circ(t)_N \subseteq X^{\mathbb{N}} \wedge (t)^\circ_N \subseteq Y^{\mathbb{N}} \wedge X^{\mathbb{N}} - {}^\circ(t)_N = Y^{\mathbb{N}} - (t)^\circ_N$$

and say that t fires*, carrying N from $X^{\mathbb{N}}$ to $Y^{\mathbb{N}}$, if it is $N^{\mathbb{N}}_X \xrightarrow{t} N^{\mathbb{N}}_Y$.*

Note that in Definition 2.3.7 the requirements ${}^\circ(t)_N \subseteq X^{\mathbb{N}}$ and $(t)^\circ_N \subseteq Y^{\mathbb{N}}$ are redundant, since $X^{\mathbb{N}} - {}^\circ(t)_N$ and $Y^{\mathbb{N}} - (t)^\circ_N$ are defined only under such assumption. We decided to list them explicitly to elucidate the fact that for $N^{\mathbb{N}}_X \xrightarrow{t} N^{\mathbb{N}}_Y$ to be true, t has to be enabled in $N^{\mathbb{N}}_X$.

Our firing policy says that, given a place $p \in P_N$, when a transition fires, *exactly* ${}^\circ(t)_N(p)$ tokens are consumed from p, and exactly $(t)^\circ_N(p)$ tokens are produced in p. This causes the net to go from the state $X^{\mathbb{N}}$ to the state $Y^{\mathbb{N}}$, where $Y^{\mathbb{N}}$ is obtained from $X^{\mathbb{N}}$ by adding/subtracting the relevant number of tokens as prescribed by the input and output functions evaluated on t.

Note that the same transition can produce and consume tokens from the same places, that is, a place can act both as an input and an output – they are not mutually exclusive. The net in Figure 2.5 is an example of this, where ${}^\circ(t)_N \cap (t)^\circ_N$ is non-empty.

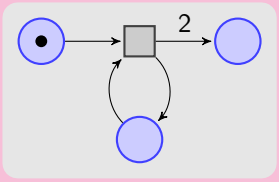


Figure 2.5: An example of a transition with intersecting input and output places.

Remark* 2.3.8 (Generalized firing policy)**.** Our firing policy can of course be generalized to arbitrary multisets of transitions $U \subseteq T_N$, defining things in the obvious way:

$$N^{\mathbb{N}}_X \xrightarrow{U} N^{\mathbb{N}}_Y := {}^\circ(U)_N \subseteq X^{\mathbb{N}} \wedge (U)^\circ_N \subseteq Y^{\mathbb{N}} \wedge X^{\mathbb{N}} - {}^\circ(U)_N = Y^{\mathbb{N}} - (U)^\circ_N$$

2.4 Examples

Petri nets are good for representing the development stage of a product, and concurrent behavior. We will show this using examples.

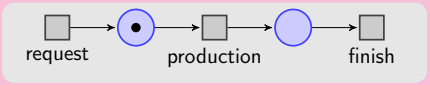


Figure 2.6: Product development example.

Example 2.4.1 (Product development)**.** We can describe the life stages of a product, from order to production, using Petri nets. The simplest case we can think of is the one in Figure 2.6. In this case, transitions correspond to different processing stages for a product. Clearly, we can design processes that are much more complicated than this, for instance introducing *exclusive choices* as in the Petri net in Figure 2.7, where the two transitions must compete to fire. Here we can imagine that a user can decide which transition fires, maybe by pressing a button or by filling a form. And with this model we can represent the fact that once one decision is taken the other one is automatically disabled.

Example 2.4.2 (Traffic Light)**.** Concurrent behavior model situations where two or more systems have to compete to get the needed resources to run. One typical example is given by a couple of traffic lights (denoted 1 and 2, respectively) at a crossing: For simplicity, each traffic light can be green or red, but they cannot be both green at the same time, otherwise cars would crash. We can model this using Petri nets (see Figure 2.8a), where two systems – representing the traffic lights – have to compete for the token in the middle to turn green.

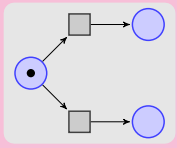
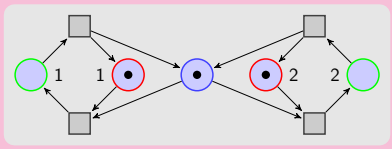


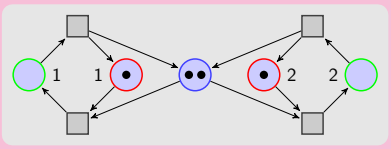
Figure 2.7: Petri net modeling exclusive choice.

In Figure 2.8a, the places have been colored in red and green, representing "the colour a traffic light is in". Transitions represent the switches that change a given traffic light's color. The numbers labeling places represent the traffic light that each place refers to.

Note that with the marking provided as in the figure above, it can never happen that both lights are green at the same time, thanks to the token in the center place: One light, say 1, could always be "better" at becoming green, thus preventing the second one to ever fire, but situations causing crashes would never happen. Note moreover that since there could be more than one token in each place, there are other markings that do not prevent this situation from happening, such as the one in Figure 2.8b.



(a) An example of a traffic light model.



(b) An example of a faulty state of the traffic light model.

Figure 2.8: Traffic light models.

2.5 Further properties of Petri nets

Now that we have defined Petri nets formally and clarified why we deem them useful, it is time to explore the properties that a Petri net can have, and to state them formally.

2.5.1 Reachability, safeness and deadlocks

Let's go back to Example 2.4.2. We already saw two different markings, generating two completely different behaviors, in Figure 2.8. We recognize that, in Figure 2.8b, firing both transitions at the bottom leads us to the marking in Figure 2.9. ...But this is exactly the situation we wanted to avoid, since now cars start crashing! This prompts a question: Given some marking $X^{\mathbb{N}}$, is it possible to *reach* a marking $Y^{\mathbb{N}}$ with a sequence of transition firings? The traffic light example should clarify how

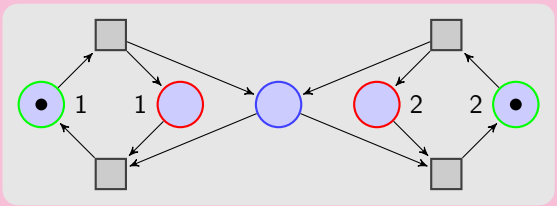


Figure 2.9: The evolution of a faulty state of the traffic light model.

important answering this question is. As usual, something important deserves a definition.

Definition 2.5.1 (Reachability)**.** *Given a Petri net $N_X^{\mathbb{N}}$ we say that a marking $Y^{\mathbb{N}}$ is* reachable *from $X^{\mathbb{N}}$ if there is a* finite *sequence of transitions $t_0, ..., t_n$ such that*

$$X^{\mathbb{N}} \xrightarrow{t_0} X_1^{\mathbb{N}} \xrightarrow{t_1} \dots \xrightarrow{t_{n-1}} X_n^{\mathbb{N}} \xrightarrow{t_n} Y^{\mathbb{N}}$$

If s is a finite sequence of transitions $(t_0, ..., t_n)$, we express the statement above by simply writing $X^{\mathbb{N}} \xrightarrow{s} Y^{\mathbb{N}}$.

Remark 2.5.2 (Studying reachability)**.** In the traffic light example our Petri net is simple enough to allow us to manually deduce if some marking can be reached from its initial state by writing out all the possible states of the net. Clearly, when we start designing complex systems, we want to develop formal tools to automatically provide answers to this question. This will be covered later on.

Another important concept is the one of *deadlock*. The idea behind this is that a net is deadlocked if "it is going to jam", meaning that at some point nothing will be able to fire anymore. This can be formalized as follows:

Definition 2.5.3 (Deadlock)**.** *Given a Petri net N in the state $X^{\mathbb{N}}$, we say that $N_X^{\mathbb{N}}$ is* deadlocked *if there is some marking $Y^{\mathbb{N}}$, reachable from $X^{\mathbb{N}}$, in which no transition can fire.*

Example 2.5.4 (Deadlocked net)**.** In Figure 2.2c it is very easy to see that the net is deadlocked, but things are not always so clear. Consider, for instance, Figure 2.10a on the left: Here everything sems fine, but firing t_2 and then t_1 two times gets us to the state in Figure 2.10b on the right, which is deadlocked. Deadlock is bad for us because it means that our process cannot progress in any way. As in the case of reachability, we want to develop higher order tools to study if a given Petri net is deadlocked or not.

(a) (b)

Figure 2.10: An example of a deadlocked Petri net.

On the other end of the spectrum, opposed to the concept of deadlock, we have the concept of *liveness*. Liveness means, in short, absence of deadlocks, as it can be easily seen from the following definition:

Definition 2.5.5 (Liveness)**.** *Given a Petri net $N_X^{\mathbb{N}}$, we say that a transition $t \in T_N$ is*

- Dead *if it can never fire.*
- Alive *if, for any marking $Y^{\mathbb{N}}$ reachable from $X^{\mathbb{N}}$, there is a firing sequence that, from $Y^{\mathbb{N}}$, leads to a marking $Z^{\mathbb{N}}$ in which t can be fired.*

We say that $N_X^{\mathbb{N}}$ is alive *if all of its transitions are alive.*

This in particular means that, starting from $N_X^{\mathbb{N}}$, we can apply any firing sequence and be sure that, if we keep going, we will always end up in a situation in which t can fire.

Remark 2.5.6 (Dead and alive are not incompatible)**.** Note that a transition can be both not dead and not alive at the same time: It may be fireable in the state $X^{\mathbb{N}}$ but become dead later on. The definition above can be generalized, introducing intermediate degrees of liveness between the ones we gave above, but we are not interested in this right now. What is interesting for us is that it is trivial to prove that an alive Petri net is not deadlocked, and that every transition will always be enabled in the future, no matter what we do.

Example 2.5.7 (Traffic light nets are alive)**.** The traffic light nets provided in Figure 2.8 are both alive, while the net in Figure 2.2c is not, consisting only of a dead transition.

If being deadlocked is considered a bad quality for a Petri net to have another property, called *boundedness*, is quite the opposite.

Definition 2.5.8 (Boundedness and safeness)**.** *Given a Petri net $N_X^{\mathbb{N}}$, we say that a place $p \in P_N$ is* k-bounded *if it never contains more than k tokens in any reachable marking. We also say that a place is* bounded *if it is k-bounded for some k.*

We can extend these definitions to the whole net saying that $N_X^{\mathbb{N}}$ is k-bounded (bounded) if all its places are k-bounded (bounded) in any state reachable from $X^{\mathbb{N}}$. Finally, we say that a Petri net is safe *if it is 1-bounded.*

Example 2.5.9 (Traffic light nets are bounded)**.** The nets in Figure 2.8 are both bounded. The one in Figure 2.8a is also safe.

If we think about Petri nets as modeling process behavior, boundedness is a desirable quality because it means that at any stage tokens do not accumulate. For example, consider the net in Figure 2.6: This net is not bounded, because the leftmost transition could keep firing accumulating tokens in the leftmost place. This would happen if, for instance, the firing rate of the leftmost transition exceeds the firing rate of the middle one, meaning that the demand exceeds the production capability.

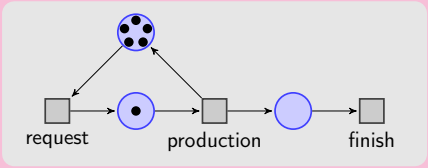


Figure 2.11: An example of a bounded Petri net modeling production.

Remark 2.5.10 (Making a net bounded)**.** Note that in a situation like the one in Figure 2.6 we can make the net k-bounded artificially by adding a place with k tokens in it, as in Figure 2.11, where we just made the net above 6-bounded. The interpretation of the added place is that it represents the *maximum production capabilities of the process*. In this case the "request" transition is *automatically disabled* (i.e. it can't fire) when there are six or more tokens waiting for production.

As usual, we would like a theoretical framework to establish when a net is bounded, or when strategies like the one above can work to make it such. Many of the questions asked here will be answered formally with the categorification of Petri nets, that will unveil their compositional nature.

2.6 Why is this useful?

This is a perfectly legitimate question, that deserves a prompt answer. We will proceed by analytically listing the ways in which Petri nets can be useful for software-design purposes. We hope to give the reader reason to believe that Petri nets are, in fact, a good formalism to base a programming language

on. In introducing Petri nets, we pointed out the following characteristics that make them very appetible for software-design purposes:

- Petri nets are inherently graphical. Since the very beginning, we were able to introduce and manipulate Petri nets diagrammatically. The pictorial representation of Petri nets is very intuitive, and allows us to quickly draft how a complex system is supposed to work. This makes designing infrastructure with Petri nets much easier than by, say, using traditional code;

- Petri nets represent concurrency well. The idea of transitions having to "fight" for resources is very useful in representing processes that could be run independently on the same resources. Again, this can be easily represented graphically, giving us a neat, intuitive explanation of what's going on. Such a feature is of great value in modeling complex systems, often consisting of multiple, independent parties performing concurrent operations on different machines;

- Petri nets can be studied formally. The graphical formalism we rely on to model systems is backed up by a sound mathematical model. This guarantees that our drawings are not just drawings on one hand, and that computers can "understand" our drawings by means of the corresponding mathematics on the other;

- Interesting properties of nets can be expressed in terms of reachability. Since reachability is formally defined, we can develop technical tools to infer if a given condition holds or not for a net. This, in particular, means that we can ask a computer to answer such questions for us. If the possibility of algorithmically deciding if some property holds or not for a given net may seem not very important, it is because all the examples provided up to now consisted of very small nets. The reader should be aware that in production applications, Petri nets can easily have many hundreds of places/transitions, and answering reachability questions without the aid of a computer is basically impossible. Clearly, up to now we don't know anything about how efficient algorithms can be in solving such problems, and indeed verifying some properties can take an exponential time (or even worse) in the size of the net. This means that as our net grows in size the time needed to know if some property holds or not for it will increase exponentially. This prompts for the development of efficient methods, such that when an efficient solution to answer a question exists, it is attained.

For the reasons above, we decided to use Petri nets as the language that Statebox uses to design code at the highest level of abstraction. In our vision, the programmer should be able to use Petri nets to draft how the software should behave by modeling it as a process, and then dive into details by filling in all the remaining information by means of a well defined procedure, backed up by sound mathematics to ensure consistency of such method.

As this is the direction we want to take, we need a way to recast the Petri nets formalism in a way that makes it compatible with other mathematical gadget we want to use for the "filling the blanks" stage we mentioned above. This will be done with the aid of *category theory*, that we will introduce in the next Chapter.

Chapter 3

Introduction to category theory

In Chapter 2 we introduced Petri nets, and definined some of their properties. Now we proceed by introducing the other main actor in the Statebox project, category theory. Category theory is a relatively young branch of mathematics that originated during the second half of the last century [9], and since then it has had an increasingly pervasive influence in the way modern mathematicians and computer scientists think. Category theory can be seen as "the glue of mathematics" and has the marvelous ability of making very different theories interact consistently with each other. Set theory is also a universal language for mathematics, with the difference that while sets focus on defining a structure "imperatively" – e.g. by specifying which properties the elements of a structure need to satisfy – category theory defines mathematical structures behaviorally, that is, by specifying patterns and interactions of a structure with structures of similar kind. In this sense, it is clear how working from a categorical perspective makes studying the interaction of different theories easier.

Since one of the main characteristics of the Statebox project is unification of advancements in very different fields of computer science, the reader can already appreciate why category theory will end up being very relevant for us. Indeed, the standard modus operandi of this document will most often reduce to the following pattern:

- Introduce a new idea;
- Find a mathematical theory that captures the idea well;
- Categorify it, that is, translate it to the language of category theory;
- Study how what we obtained interacts with what we already had. One of the main advantages of category theory is that its extensive toolbox makes this step much easier.

Albeit having just sketched out why category theory will have a central role in the development of our theoretical framework, we already have what we need to introduce it in all of its glory. We will need increasingly more categorical tools as this document progresses, and some of the concepts covered here will only be used far away in the future. The reader should then employ this Chapter as a reference every time a new categorical concept will be needed, and not worry too much if something explained here initially doesn't seem very "useful". Eventually, every detail will find its place in the environment we are building up.

3.1 What is category theory?

That's a great question – in many ways the answer deepens every day. Category theory is primarily a way of thinking, more than just a theory in the usual sense of the term. Probably the simplest idea

of category theory is that everything is interrelated. This applies not only to mathematics, but also computation, physics, and other sciences which are just beginning to be elucidated and unified via the use of categories – and is precisely the reason why category theory has such natural real-world applications. Of course the pertinent application here is in the context of computer science, and the mission of Statebox is to make programming concrete, principled, and universal. First, we begin with a simple mathematical overview of category theory.

According to [19], a nice way to describe category theory is as *the language for describing and observing patterns in mathematics*. Every *object* is of a certain kind, which is interrelated by a *morphism* intrinsic to the kind. For example, a morphism of *sets* is simply a *function*, while a morphism of structured objects cooperates with the structure, e.g. an algebraic operation. Taken together, the objects and morphisms form a *category*, which encapsulates the particular notion, and more so, connects it to all of mathematics – the category is *itself* a kind of object, and we can consider the category of categories! The morphisms between categories, called *functors*, respect the *composition* of morphisms in the related categories, providing a fundamental connection between distinct concepts. A functor witnesses how the reasoning patterns found in a certain theory are "compatible" with the patterns found in another. This entails a well-behaved notion of compatibility between different theories, an essential aspect of principled theoretical modeling called *compositionality*. In a way, this perspective already empowers us when thinking about mathematics as a whole! But let's slow down and see the basic definitions.

Definition 3.1.1 (Category)**.** *A* category $\mathscr{C}$ *consists of*

- *A collection of* objects*, denoted as* $\mathrm{obj}\ \mathscr{C}$*;*
- *A collection of* morphisms*, denoted as* $\mathrm{Hom}_{\mathscr{C}}$*;*
- *Two functions* $\mathrm{s}(-), \mathrm{t}(-) : \mathrm{Hom}_{\mathscr{C}} \to \mathrm{obj}\ \mathscr{C}$ *called* source *(or* domain*) and* target *(or* codomain*), respectively;*
- *A partial function* $(-);(-) : \mathrm{Hom}_{\mathscr{C}} \times \mathrm{Hom}_{\mathscr{C}} \to \mathrm{Hom}_{\mathscr{C}}$*, called* composition*, that assigns to every pair* $f, g \in \mathrm{Hom}_{\mathscr{C}}$*, such that* $s(g) = t(f)$*, the arrow* $f; g$*;*
- *An* identity *function* $\mathrm{obj}\ \mathscr{C} \to \mathrm{Hom}_{\mathscr{C}}$*, that assigns to every object* x *an arrow* id_x*.*

Moreover, we require that the following axioms have to be satisfied:

- $\mathrm{s}(f) = \mathrm{s}(f; g)$ *and* $\mathrm{t}(f; g) = \mathrm{t}(g)$*;*
- $\mathrm{s}(id_x) = x = \mathrm{t}(id_x)$*;*
- $f; (g; h) = (f; g); h$ *for each* f, g, h *arrows such that composition is defined;*
- $f; id_{\mathrm{t}(f)} = f = id_{\mathrm{s}(f)}; f$ *for each arrow* f*.*

An arrow f *such that* $\mathrm{s}(f) = A, \mathrm{t}(f) = B$ *is often denoted with* $f : A \to B$ *or* $A \xrightarrow{f} B$*.*

The concept of category is a very powerful one, and we redirect the reader who wants to know more to [18]: Category theory can indeed become very difficult to grasp only with the introduction of its simplest concepts, and this document is not the right place for an in-dept exposition. Nevertheless, it is worth to give an intuitive explanation of the definition given above: *Objects* can be thought of as representing systems, resources, or states of a machine. *Arrows* represent transformations between them, that is, processes that turn a given system (or resource, or state) into another according to some rules. Moreover, *composition* tells us that transformations can be serialized: Transforming A into B using f and then B into C using g is the same as transforming A into C using $f; g$. The axioms tell us that composing transformations is *associative*, and moreover that for each system A "doing nothing" can be regarded as an *identity transformation* id_A.

Remark 3.1.2. In interpreting objects as states of a system and morphisms as transformations between them, we already see some similarity with the interpretation we gave of Petri nets in Chapter 2. This similarity will be described in depth in Chapter 4.

Example 3.1.3 (Sets and functions)**.** There is a category, denoted with **Set**, whose objects are sets and whose morphisms are functions between them. It is easy to see that composition of functions is a function, composition is associative, and that every set has an identity function carrying every element into itself. Hence **Set** is indeed a well-defined category.

Remark 3.1.4 (Notation)**.** From now on, we will stick to the convention of indicating generic categories with curly letters, like $\mathscr{C}, \mathscr{D}, \mathscr{E}$. Objects will be denoted with capital Latin letters, preferably from the beginning of the alphabet, A, B, C etc. Morphisms will be denoted with small Latin letters, preferably from the middle of the alphabet, f, g, h etc. Categories that deserve a name of their own, like the one in Example 3.1.3, will have the name denoted in bold letters, as in **Set**.

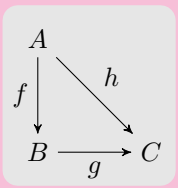
Example* 3.1.5 (Functional programming)**.** We can build a category **Hask** where objects are data types and morphisms are Haskell [13] functions from one type to another. Associativity is composition of functions, and identity morphisms are the algorithms sending terms to themselves. Defining the category **Hask** is actually not as easy as it seems and we will discuss more about this issue in Remark 5.2.5.

Example* 3.1.6 (Groups, topological spaces)**.** Groups and homomorphisms between them form a category, called **Group**. So do topological spaces and continuous functions, forming the category **Top**.

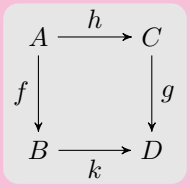
Remark* 3.1.7 (Free categories from graphs)**.** There is an evident connection between the definition of a category and the definition of a graph. A category just looks like "the transitive closure of a graph, with loops added at every vertex". This connection between categories and graphs is indeed real, and one can always generate a free category from a directed graph [18, Ch. 2, Sec. 7].

Remark* 3.1.8 (Size issues)**.** The reader with experience in mathematics will have noticed how we have been vague in saying what we mean by "a collection of objects" in the definition of a category. Indeed, note how the objects of the category **Set**, **Group** and **Top** do not form a set, but a *proper class*. All these *size issues* are deeply covered in any comprehensive book about category theory, and we refer the reader to [18, Ch. 1, Sec. 6] for details.

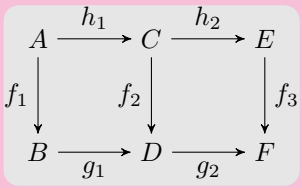
Remark 3.1.9 (Commutative diagram)**.** A neat way to express equations between morphisms in a category is via *commutative diagrams*. A commutative diagram is just a picture that shows us how morphisms compose with each other. Commutative diagrams are interpreted as follows: Vertexes are objects in a category. Paths between objects are compositions of morphisms. If there are multiple paths from one object to another, this means that the corresponding morphisms are equal. For example, the diagram in Figure 3.1a states that $f; g = h$, while the diagram in Figure 3.1b states that $h; g = f; k$.



(a) $f; g = h$.



(b) $h; g = f; k$.



(c) Commutativity of squares ensures commutativity overall.

Figure 3.1: Examples of commutative diagrams.

Commutative diagrams are a fundamental tool in category theory, and are routinely used to prove things. The standard way to prove something in category theory is to draw a diagram representing our thesis,

and then try to prove that the diagram commutes. A way to do this is by dividing the diagram into multiple sub-diagrams and proving the commutativity of each of them separately. The commutativity of the overall diagram can then be inferred by the commutativity of its components. To see how this works, consider Figure 3.1c: If we know that the left and right squares commute, then so does the rectangle obtained from their composition, in fact:

$$f_1; g_1; g_2 = (f_1; g_1); g_2 = (h_1; f_2); g_2 = h_1; (f_2; g_2) = h_1; (h_2; f_3) = h_1; h_2; f_3$$

where the second equality follows from the commutativity of the left square, while the fourth follows from the commutativity of the right one.

To conclude this section, we introduce the concept of *isomorphism*. Intuitively, an isomorphism in a category is a morphism that allows us "to go back and forth between two objects". This is easily defined as follows:

Definition 3.1.10 (Isomorphism)**.** *Given a category $\mathscr{C}$ we say that a morphism of $\mathscr{C}$ $f : A \to B$ is an* isomorphism *(or just an* iso*) if there is a morphism $f^{-1} : B \to A$ such that*

$$f; f^{-1} = id_A \qquad f^{-1}; f = id_B$$

The definition of isomorphism is nothing new, and captures the idea of a "reversible process". We already know examples of this:

Example 3.1.11 (Isos in **Set**)**.** In **Set**, the isomorphisms are exactly the bijective functions.

Example* 3.1.12 (Isos in **Group** and **Top**)**.** In **Group**, the isomorphisms are exactly the bijective homomorphisms. In **Top**, the isomorphisms are exactly the homeomorphisms.

3.2 Functors, natural transformations, natural isomorphisms

We mentioned functors en passant in the introduction of this Chapter, when we said that *a functor is a morphism between categories*. We moreover added that a morphism in a category can be thought of as a transformation that preserves *all the relevant structure* from its domain to its codomain. So, if a functor is a morphism of categories, which is the relevant structure it has to preserve?

Well, in a general category the only things that we have are identities for each object and composition of morphisms, so it seems reasonable to require these to be preserved by a functor. This is, indeed, enough:

Definition 3.2.1 (Functor)**.** *A* functor *F from a category $\mathscr{C}$ to a category $\mathscr{D}$, often denoted with $F : \mathscr{C} \to \mathscr{D}$ or $\mathscr{C} \xrightarrow{F} \mathscr{D}$, consists of the following:*

- *A map from $\mathrm{obj}\ \mathscr{C}$ to $\mathrm{obj}\ \mathscr{D}$, that associates to the object A of $\mathscr{C}$ the object FA of $\mathscr{D}$;*
- *A map from $\mathrm{Hom}_{\mathscr{C}}$ to $\mathrm{Hom}_{\mathscr{D}}$, that associates to the morphism $f : A \to B$ of $\mathscr{C}$ the morphism $Ff : FA \to FB$ of $\mathscr{D}$.*
- *We moreover require that the following equalities hold:*

$$F_{id_A} = id_{FA} \qquad F(f; g) = Ff; Fg \tag{3.2.1}$$

In particular, Equations 3.2.1 mean that identities get carried to identities and compositions to compositions, as we would have expected. Note how *this is enough to guarantee that F sends any commutative diagram in $\mathscr{C}$ to a commutative diagram in $\mathscr{D}$*. This is the whole point about functors: If the main way to prove facts in category theory is by using commutative diagrams, a functor is basically sending facts about $\mathscr{C}$ to facts about $\mathscr{D}$. This allows us to "export" theorems from one category to another, and is a tremendously powerful feature to carry results across mathematical theories.

Example 3.2.2 (Identity functor)**.** For each category $\mathscr{C}$ there is a functor $id_{\mathscr{C}}$ that sends each object and each morphism of $\mathscr{C}$ to itself, respectively.

Example* 3.2.3 (Homotopy groups)**.** There is a functor from the category of topological spaces and homotopy classes of continuous functions, **hTop**, to the category **Group**. This is exactly what makes it possible to deduce if a given topological space is connected or not – studying its homotopy group.

Remark 3.2.4 (Functor composition)**.** Given two functors $\mathscr{C} \xrightarrow{F} \mathscr{D} \xrightarrow{G} \mathscr{E}$ we can compose them by composing their maps on objects and morphisms. The composition sends an object A of $\mathscr{C}$ to an object FGA of $\mathscr{E}$, and a morphism $f : A \to B$ in $\mathscr{C}$ to $FGf : FGA \to FGB$ in $\mathscr{E}$.

Remark 3.2.5 (Notation)**.** It is commonplace to denote functors using capital Latin letters from the middle of the alphabet, F, G, H etc. Also, the application of a functor to an object or a morphism is usually written without using parentheses, as in FA, Ff.

We can now start playing with the definition of functor a bit more. First, something simple:

Definition 3.2.6 (Isomorphism of categories)**.** *Using Remarks 3.2.2 and 3.2.4 it is not difficult to convince ourselves that categories and functors form the objects and morphisms, respectively, of a category, called* ***Cat****. Then we can apply Defintion 3.1.10 in this context and obtain that the two categories $\mathscr{C}$ and $\mathscr{D}$ are* isomorphic *when there are functors $F : \mathscr{C} \to \mathscr{D}$ and $F^{-1} : \mathscr{D} \to \mathscr{C}$ such that $F; F^{-1} = id_{\mathscr{C}}$ and $F^{-1}; F = id_{\mathscr{D}}$.*

The definition of isomorphism between categories is not really the interesting one for us. This is because it is too restrictive. We can, though, relax it a bit to make it more manageable. To do this, we first need to introduce some properties.

Definition 3.2.7 (Full and faithful functors)**.** *A functor $F : \mathscr{C} \to \mathscr{D}$ is called* full *if, for any objects A, B in $\mathscr{C}$ and any morphism $f : FA \to FB$ in $\mathscr{D}$, there is always a morphism g in $\mathscr{C}$ such that $Fg = f$.*

On the other hand, F is called faithful *if given morphisms $f, g : A \to B$ in $\mathscr{C}$, $f \neq g$ implies $Ff \neq Fg$ in $\mathscr{D}$.*

When a functor is full and faithful, we sometimes say that it is fully faithful.

In essence, a functor $F : \mathscr{C} \to \mathscr{D}$ is full when every morphism between objects of the form FA, FB – that is, objects that are hit by F – comes from $\mathscr{C}$. This means that the morphisms $A \to B$ are *at least as many* as the morphisms $FA \to FB$. Similarly, fullness implies that the morphisms $FA \to FB$ are *at least as many* as the ones $A \to B$, since different morphisms from A to B go to different morphisms from FA to FB. When a functor is fully faithful, then, all the morphisms between objects of $\mathscr{C}$ are carried to $\mathscr{D}$ exactly as they are, and all the objects of the form FA for some A in $\mathscr{C}$, together with their morphisms, form "a copy" of $\mathscr{C}$ in $\mathscr{D}$.

This is pretty close to an equivalence of categories, but in $\mathscr{D}$ there could be other morphisms that are not hit by F, viz. objects that cannot be written as FA for some A in $\mathscr{C}$. Since these objects are not hit by F they could behave as they want to, while the structure of objects of type FA and their morphisms is completely determined by $\mathscr{C}$ and the full faithfulness of F. To rule out this eventuality, we give the following definition:

Definition 3.2.8 (Equivalence of categories)**.** *Two categories $\mathscr{C}$ and $\mathscr{D}$ are said to be* equivalent *when there is a functor $F : \mathscr{C} \to \mathscr{D}$ that is fully faithful and* essentially surjective*, meaning that each object of $\mathscr{D}$ is isomorphic to an object of the form FA for some A in $\mathscr{C}$.*

Now we see that Definition 3.2.8 is the right one to describe categories that are, structurally speaking, the same: All the objects and morphisms in $\mathscr{D}$ are forced to behave like objects and morphisms in $\mathscr{C}$, since either they are hit by the functor F, and then are taken care of by the full faithfulness of F, or they are not, in which case they are isomorphic to some object which is. Equivalence of categories will have a big role in Chapter 4, and we postpone any meaningful example until then.

Now that functors have been introduced, it is legitimate to ask if there is a notion of "morphism between functors": Suppose we have categories $\mathscr{C}$ and $\mathscr{D}$, and functors $F, G : \mathscr{C} \to \mathscr{D}$. We know that if $\mathfrak{D}$ stands for a commutative diagram in $\mathscr{C}$, the functors F, G carry it to a couple of commutative diagrams in $\mathscr{D}$. The question, then, is: Is it possible to establish a relationship between the diagrams in $\mathscr{D}$ to which $\mathfrak{D}$ is carried to by F and G, respectively? The answer to this question is *yes*, and the notion we are looking for is the one of *natural transformation*.

Definition 3.2.9 (Natural transformation)**.** *Given functors $F, G : \mathscr{C} \to \mathscr{D}$, a* natural transformation from F to G*, denoted with $\eta : F \to G$, consists of a collection of morpshisms of $\mathscr{D}$*

$$\{\eta_A : FA \to GA\}_{A \in \mathrm{obj}\, \mathscr{C}}$$

such that, for every morphism of $\mathscr{C}$ $f : A \to B$, the diagram in Figure 3.2 commutes.

$$\begin{array}{ccc} FA & \xrightarrow{\eta_A} & GA \\ {\scriptstyle Ff}\downarrow & & \downarrow{\scriptstyle Gf} \\ FB & \xrightarrow[\eta_B]{} & GB \end{array}$$

Figure 3.2: Commutativity for a natural transformation.

Definition 3.2.9 is a bit tricky. The morphisms defining η, also called its *components*, live in $\mathscr{D}$, but are indexed by objects of $\mathscr{C}$. This is for the following reason: We want to find a procedure to turn every diagram where each vertex and edge is an application of F to an object or morphism of $\mathscr{C}$, respectively, into a diagram where each vertex and edge is an application of G to the same object or morphism of $\mathscr{C}$. In practice, this means looking for a rewriting procedure that strips all the occurences of F from the diagram and replaces them with G. To do this, what we need to do is establish a correspondence between vertexes, that is, a correspondence $FA \to GA$ for each object A. Since FA, GA are objects of $\mathscr{D}$, this correspondence will have to be a morphism of $\mathscr{D}$. Clearly, we need as many of these correspondences as there are FA and GA, so one for each object A of $\mathscr{C}$. Moreover, it is easy to prove that the commutativity of the square in Definition 3.2.9 is everything we need so that the correspondence between the diagrams doesn't break their commutativity.

Finally, we can combine Definitions 3.2.9 and 3.1.10 to capture the concept of "going back and forth between diagrams only made of applications of F and diagrams made only by applications of G", as follows:

Definition 3.2.10 (Natural isomorphism)**.** *Given functors $F, G : \mathscr{C} \to \mathscr{D}$, a natural transformation $\eta : F \to G$ is called a* natural isomorphism *if each component of η is an isomorphism in $\mathscr{D}$.*

We see that the concept of a natural isomorphism is a very strong one. It consists of a number of isomorphisms $\eta_A : FA \to GA$ which are *consistently connected with each other*. Moreover, it is easy to say that if $\eta : F \to G$ is a natural isomorphism, then there is a natural transformation $G \to F$ defined by taking the inverse of each η_A, and that these two natural transformations are each other's inverses. The concept of natural isomorphism is very useful to express all sorts of "coherence conditions" which are the categorical tools capturing the idea of "it doesn't matter in which way you stack up these commutative diagrams, the result will be the same". We will see an example of this in the next section.

3.3 Monoidal categories

Up to now, we have only had one operation between morphisms in a category, composition. Composition has a very clear time-like interpretation, especially if we interpret objects as states of a system, and morphisms between them as processes. In fact, we can clearly read $f;g$ as "apply f and *then* apply g". The question, then, is if there is a categorical notion that captures the idea of "things happening in parallel". The answer to this question is positive, and is provided by the following definition.

Definition 3.3.1 (Monoidal category)**.** *A* monoidal structure *for a category $\mathscr{C}$ consists of:*

- *A functor $\otimes : \mathscr{C} \times \mathscr{C} \to \mathscr{C}$, called the* monoidal product *or, sometimes, the* tensor product *(because traditionally the symbol used to denote it, $\otimes$, denotes tensor products in linear algebra). Note that in this case $\mathscr{C} \times \mathscr{C}$ is a product of categories, which will be formally introduced in Definition 3.6.2 and Example 3.6.3. Intuitively, the functor $\otimes$ can be interpreted as having two arguments: It associates an object (a morphism, respectively) of $\mathscr{C}$ to each couple of objects (morphisms, respectively) of $\mathscr{C}$ such that the functor laws hold for both components:*

$$id_A \otimes id_B = id_{A\otimes B} \qquad (f;g) \otimes (h;k) = (f \otimes h);(g \otimes k)$$

- *A selected object I of $\mathscr{C}$, called the* monoidal unit*;*
- *A natural isomorphism*

$$\alpha : ((-) \otimes (-)) \otimes (-) \xrightarrow{\simeq} (-) \otimes ((-) \otimes (-))$$

  called associator*, with components in the form*

$$\alpha_{A,B,C} : (A \otimes B) \otimes C \xrightarrow{\simeq} A \otimes (B \otimes C)$$

  that expresses the fact that the tensor operation is associative;

- *Natural isomorphisms*

$$\lambda : I \otimes (-) \xrightarrow{\simeq} (-) \qquad \rho(-) \otimes I \xrightarrow{\simeq} (-)$$

  called left and right unitors*, respectively, with components in the form:*

$$\lambda_A : I \otimes A \xrightarrow{\simeq} A \qquad \rho_A : A \otimes I \xrightarrow{\simeq} A$$

  that express the fact that I behaves as a unit;

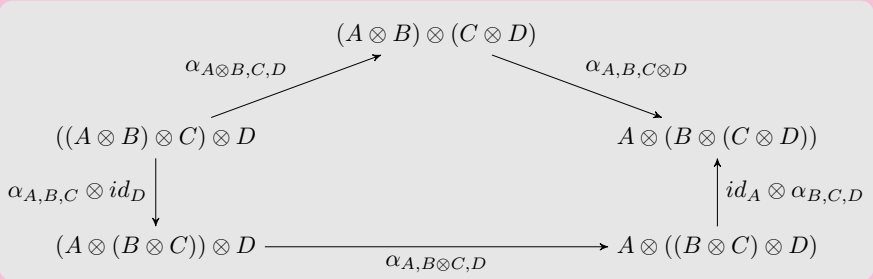
- *These natural isomorphisms have to respect the so called* coherence conditions*, that imply that associator and unitors are well behaved, and can thus be used in full generality. Coherence conditions are expressed in the form of* commutative diagrams*, as in Figure 3.3.*

A category $\mathscr{C}$, together with a monoidal structure, is called a monoidal category*.*

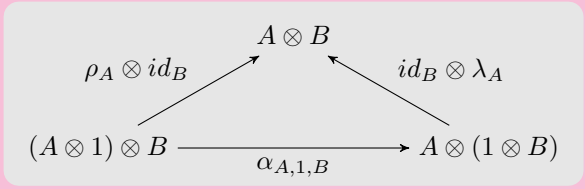
Let us try to make this definition more explicit: The monoidal product $\otimes$ captures the idea of parallel composition. $A \otimes B$ represents two systems existing at the same time. $f \otimes g$ represents two processes f, g being applied at the same time on different systems.

The associator captures the idea that the monoidal product is associative: We can always go from $(A \otimes B) \otimes C$ to $A \otimes (B \otimes C)$ and vice-versa without destroying any fact proven by commutative diagrams (that is why we need associators to be *natural* isomorphisms!)

The monoidal unit represents the *trivial system*. This behavior is enforced by left and right unitors, which tell us that we can go from $A \otimes I$ to A to $I \otimes A$ in any way we want, without losing information. We can deduce that the system I doesn't add or remove any information when composed with A.



(a) Coherence condition for the associator.



(b) Coherence condition for unitors.

Figure 3.3: Coherence conditions for monoidal categories.

Coherence conditions require a few more words. They are what make associators behave like associators and unitors behave like unitors, and are expressed by two commutative diagrams. These two diagrams are very important, because it can be proved that when they commute *any other diagram* made uniquely of associators, monoidal products and unitors commutes, effectively meaning that adding I to a monoidal product or changing the bracketing in any possible way really doesn't change anything, as we would expect.

Remark 3.3.2 (Notation)**.** When we want to make explicit that $\mathscr{C}$ is a monoidal category, we use the notation $(\mathscr{C}, \otimes, I)$, where I represents the monoidal unit and $\otimes$, the monoidal product. For instance, if we say that $(\mathscr{C}, \otimes, I)$ and $(\mathscr{D}, \square, I')$ are monoidal categories, we are denoting the tensor product as $\otimes$ in $\mathscr{C}$ and as $\square$ in $\mathscr{D}$, and their monoidal units as I, I', respectively.

Example 3.3.3 (Products of sets)**.** The category **Set** can be made into a monoidal category $(\mathbf{Set}, \times, \{\star\})$, where $\times$ is the cartesian product of sets and $\{\star\}$ is the one element set. The associator is the usual rebracketing for tuples, while unitors are the isomorphisms sending both $(a, \star)$ and $(\star, a)$ to a.

Example 3.3.4 (Coproducts of sets)**.** The category **Set** admits another monoidal structure, and can thus also be turned into a monoidal category $(\mathbf{Set}, \sqcup, \emptyset)$, where $\sqcup$ denotes the disjoint union of sets and $\emptyset$ denotes the usual empty set. The associator is the usual rebracketing of disjoint unions, while unitors are the identities expressing the fact that taking the disjoint union of a set A with the empty set gives back A.

Remark 3.3.5 (Monoidal structures are not unique)**.** Examples 3.3.3 and 3.3.4 show that the category **Set** admits two different monoidal structures. It is very easy to see that $(\mathbf{Set}, \times, \{\star\})$ and $(\mathbf{Set}, \sqcup, \emptyset)$ are different monoidal categories, since in general $A \times B \neq A \sqcup B$. This proves that it is often incorrect to refer to a category as monoidal without explicitly stating what the monoidal structure is, unless it is clear from the context. If we say that **Set** is a monoidal category, to which monoidal structure are we referring to?

Example* 3.3.6 (Monoidal structures for **Group** and **Top**)**.** The cartesian product of groups, with operations defined component-wise, defines a monoidal structure on **Group**. Similarly, the product of topological spaces defines a monoidal structure on **Top**.

Note that, in a monoidal category, $A \otimes B$ is not the same object as $B \otimes A$, and there is no general way to go from one to the other. This can be a useful feature if we want to model a notion of parallel composition which is "position-sensitive", but in other situations it can be a blocker. For instance, it conflicts with the idea of systems that can be *swapped*, meaning that it doesn't matter which system is on the left and which system is on the right, since we can always exchange their places.
If we want to describe entities that can be composed in parallel where swapping is permitted, we have to require this explicitly, imposing more properties that our monoidal category has to satisfy.

Definition 3.3.7 (Symmetric monoidal category)**.** *A* symmetric monoidal category *is a monoidal category* $(\mathscr{C}, \otimes, I)$ *together with a natural isomorphism*

$$\sigma : (-) \otimes (-) \xrightarrow{\simeq} (-) \otimes (-)$$

called symmetry *(or* swap*), with components in the form*

$$\sigma_{A,B} : A \otimes B \xrightarrow{\simeq} B \otimes A$$

such that the diagram in Figure 3.4 commutes and, moreover,

$$\sigma_{A,B}; \sigma_{B,A} = id_{A \otimes B} \tag{3.3.1}$$

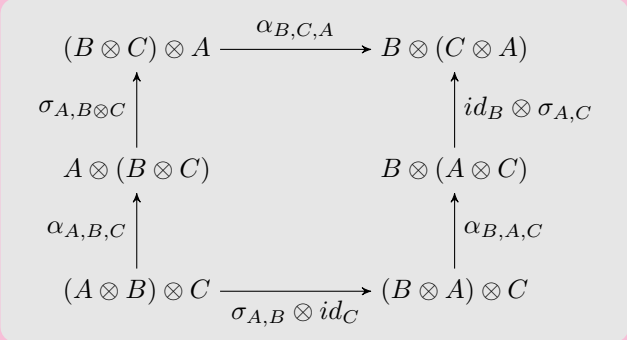


Figure 3.4: Additional coherence condition for symmetric monoidal categories.

Notice how Equation 3.3.1 suffices to state that σ is its own inverse (consistent with the idea that swapping A for B and then B for A amounts to do nothing), while the diagram in Figure 3.4 guarantees that the order in which we swap more than two objects doesn't matter.

Example 3.3.8. (Symmetric monoidal categories in **Set**) Both $(\mathbf{Set}, \times, \{\star\})$ and $(\mathbf{Set}, \sqcup, \emptyset)$ are symmetric monoidal categories. In the first case, σ is just the natural isomorphism that swaps terms in a couple:

$$(a, b) \mapsto (b, a)$$

In the second, remembering that $A \sqcup B$ can be represented as couples (x, y) where $y = 0$ if $x \in A$ and $y = 1$ if $x \in B$, then σ is the natural isomorphism

$$(x, y) \mapsto (x, y + 1 \mod 2)$$

Example* 3.3.9 (Non-symmetric monoidal category)**.** Left modules over a ring R and their module homomorphisms form a category. The usual tensor product of modules defines a monoidal category, with the trivial left module R serving as unit. If R is not commutative, this monoidal category is not symmetric.

We conclude this section with a last definition, that is just a strengthening of Definition 3.3.1.

Definition 3.3.10 (Strict monoidal category)**.** *We say that a category is* strict monoidal *when* associators and unitors are identities. *This means that, in a strict monoidal category,*

$$(A \otimes B) \otimes C = A \otimes (B \otimes C) \qquad I \otimes A = A = A \otimes I$$

Note how in a strict monoidal category the coherence conditions for associators and unitors become trivial, since all the morphisms are equalities.

Example* 3.3.11 (The category of endofunctors is strict monoidal)**.** Given a category $\mathscr{C}$ we can consider the category $[\mathscr{C}, \mathscr{C}]$ that has functors of the form $F : \mathscr{C} \to \mathscr{C}$ as objects and natural transformations between them as arrows. Maybe counterintuitively, functor composition defines a monoidal structure on $[\mathscr{C}, \mathscr{C}]$, with monoidal unit being the identity functor $id_{\mathscr{C}}$. Strictness of $[\mathscr{C}, \mathscr{C}]$ follows immediately from associativity of composition and identity laws between functors, that hold with equality.

Example 3.3.12 (Products of sets are not strict)**.** $(\mathbf{Set}, \times, \{\star\})$ is not a strict monoidal category. This is because a generic couple $((a, b), c)$ is not equal to the couple $(a, (b, c))$, albeit one can be mapped into the other and vice-versa. While mathematicians often ignore this phenomenon, functional programmers are particularly sensitive to this sort of nuance, which often prevents a program from correctly type-checking.

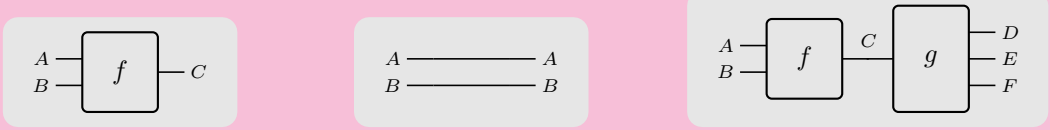
Remark 3.3.13 (Monoidal categories are equivalent to strict ones.)**.** A quite useful result, that can be found in [18, Ch. 11, Sec. 3, Thm. 1], proves that every monoidal category is *equivalent* to a strict monoidal one. We didn't formally define, in this document, what an equivalence of monoidal categories is, but you can guess it by massaging the Definition 3.2.8: It is just a normal equivalence where our functor is monoidal (moidal functors will be defined in Section 3.5)! This is useful since it means that every time we are working with a monoidal category we can also work with a strict version of it, where many of the important properties stay the same but life is easier. This scales to symmetric monoidal categories in the obvious way.

3.4 String diagrams

One of the most striking features of strict monoidal categories is that they admit a convenient *graphical calculus* that allows us to forget the mathematical notation altogether and work just using pictures - string diagrams. The best thing about this approach is that these pictures are formally defined, ensuring that if we manipulate our drawings following some basic rules we are correctly manipulating morphisms in the underlying category.
In the graphical formalism, to be read left to right, objects are represented as *typed wires*, and morphisms as *boxes*, as in Figure 3.5a. Identity morphisms are just represented as wires (see Figure 3.5b), which is clearly consistent with the idea of identity morphisms "doing nothing". As we already noticed, composition of morphisms can express the idea of sequential composition, and is thus represented by *connecting the output wire of a box with the input wire of another when the wire types match*, as in Figure 3.5c.

The monoidal product, representing the idea of parallel composition, is depicted by *placing boxes and wires next to each other*, as shown in Figure 3.5d. This is consistent with the idea that $(f \otimes g) \otimes h \simeq f \otimes (g \otimes h)$ via the associator, hence we don't need to represent any bracketing. The unit wire I represents the trivial system, and is thus *not drawn*, see Figure 3.5e. This again backs up the intuition that $I \otimes A$, A and $A \otimes I$ are morally the same. Finally, symmetry is represented by just *swapping wires*, as in Figure 3.5f.

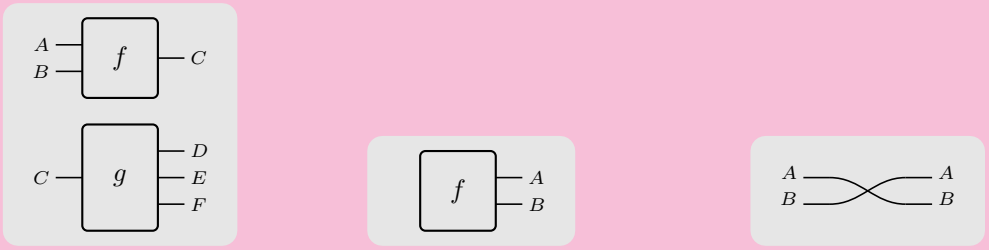
Remark 3.4.1 (Equivalence to a strict category is necessary for diagrammatics)**.** Note that in depicting monoidal products without brackets, and in choosing not to draw the monoidal unit, we are implicitly making use of the result mentioned in Remark 3.3.13. Working in the graphical formalism means exactly working in the strict symmetric monoidal category equivalent to the monoidal category we want to study.



(a) A morphism $f : A \otimes B \to C$.

(b) Identity morphism $id_A \otimes id_B = id_{A \otimes B}$.

(c) Composition $f; g : A \otimes B \to D \otimes E \otimes F$.



(d) $f \otimes g : A \otimes B \otimes C \to C \otimes D \otimes E \otimes F$.

(e) $f : I \to A \otimes B$. Trivial system wire I is not depcited.

(f) The symmetry $\sigma_{A,B} : A \otimes B \to B \otimes A$.

Figure 3.5: Graphical calculus for symmetric monoidal categories.

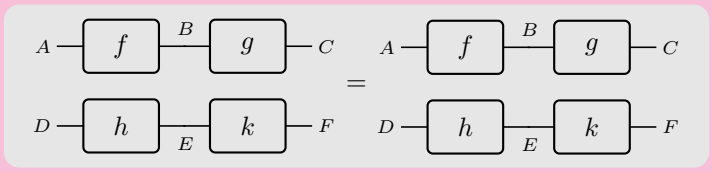


Figure 3.6: Graphical proof of the Eckmann-Hilton argument.

Example 3.4.2 (Eckmann-Hilton argument)**.** To point out how powerful the diagrammatic formalism is, note that results such as the Eckmann-Hilton argument for monoidal categories, that is, one of the equations expressing the functoriality of the monoidal product:

$$(f; g) \otimes (h; k) = (f \otimes h); (g \otimes k)$$

reduce to tautologies, making proofs much easier (see Figure 3.6). This is very interesting considering that the equation above doesn't look trivial, while the corresponding diagrams surely do: The graphical formalism helps by stripping away many of the irrelevant details when we work with monoidal categories.

The study of string diagrams goes often under the name of *process theory*, of which [6] is one of the most complete references.

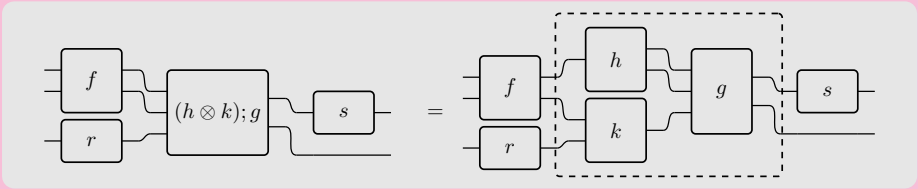


Figure 3.7: The graphical calculus allows us to explode boxes.

The analogy between process theories and programming is more than evident: A box can be thought of as a piece of software that performs some operations on data having certain types, and functional

programming can be entirely formalized using these diagrams. Moreover, note that we can *explode* a box, that is, boxing more components into a unique one. For instance, in Figure 3.7 (where the wire types have been omitted to avoid clutter), we are considering the morphism $(h \otimes k); g$ as a unique box (dashed). This allows us to zoom in/out our processes and hide the features that are irrelevant at a given level of generality. We can then form new boxes by just stacking up some other processes and considering them as one.

Remark 3.4.3 (Completeness of graphical calculi)**.** The kind of string diagrams covered here is one of the most simple graphical formalisms studied in process theories, but it is good to unveil how category theory can provide nice tools to reason about compositionality without having to learn difficult maths. The reason why it works, viz. why categorical proofs can be carried out graphically, relies on a *completeness theorem* which results from linking things that are graphically provable to things that are provable in monoidal categories. Details about this can be found in [25].

Remark 3.4.4 (String diagrams and commutative diagrams are different things)**.** Often pictures in the graphical calculus are referred to as *diagrams*. Do not confuse these diagrams with the *commutative diagrams* introduced in Remark 3.1.9!

As we hinted in the beginning of this Chapter, category theory together with its links to graphical calculi will act as "deus ex machina" in the formalization of Statebox: All the theories presented in the remainder of this document will admit a strong categorical formalization, from which it will be possible to create an equivalent graphical formalization in a safe way. "Pure" category theory will be used to "sew together" all these different theories, and obtain a formally organic and satisfying foundation on which Statebox will be implemented.

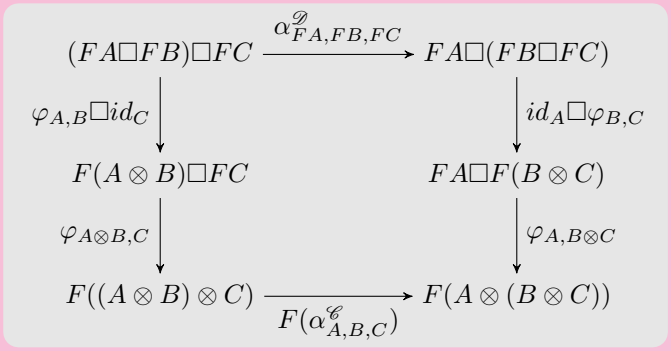
This is exactly what backs up our claim that, in Statebox, it is possible to do software engineering in a way that is at the same time *purely graphical and purely correct*.

3.5 Monoidal functors

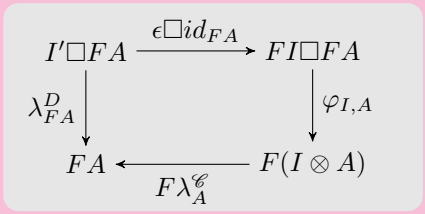
What happens to our functors if our categories are monoidal? If $(\mathscr{C}, \otimes, I)$ and $(\mathscr{D}, \square, I')$ are monoidal categories and there is a functor $F : \mathscr{C} \to \mathscr{D}$, there is nothing in principle that says that monoidal products will be preserved. For instance, if we consider $A \otimes B$, then $F(A \otimes B)$ and $FA \square FB$ may be totally unrelated. Embracing the idea that "a functor is a morphism between categories", we see that in restricting to monoidal categories there is some additional, relevant structure that functors are not preserving. We deduce, then, that the notion of a functor is not the correct one to model morphisms between monoidal categories. Here, we want to find conditions for F to *preserve the monoidal structure*. This idea prompts various different definitions, that are nevertheless related to each other.

Definition 3.5.1 (Lax monoidal functor)**.** *A* lax monoidal functor *between two monoidal categories* $F : (\mathscr{C}, \otimes, I) \to (\mathscr{D}, \square, I')$ *is specified by the following infomation:*

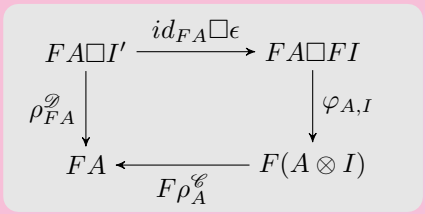
- *A functor* $F : \mathscr{C} \to \mathscr{D}$*;*
- *A morphism in* $\mathscr{D}$
$$\epsilon : I' \longrightarrow FI$$
- *A natural transformation*
$$\varphi : F(-) \square F(-) \longrightarrow F((-) \otimes (-))$$
with components in the form
$$\varphi_{A,B} : FA \square FB \longrightarrow F(A \otimes B)$$



(a) Associator condition for lax monoidal functors.



(b) Left unitor condition for lax monoidal functors.



(c) Right unitor condition for lax monoidal functors.

Figure 3.8: Coherence conditions for a lax monoidal functor.

Such that the diagrams in Figure 3.8 commute, where superscripts $\mathscr{C}, \mathscr{D}$ denote if the associator/left unitor/right unitor are the ones in $\mathscr{C}$ or the ones in $\mathscr{D}$, respectively.

The diagram in Figure 3.8a expresses the idea that the monoidal functor respects associators: It says that there is no real difference in applying the associator in $\mathscr{C}$ and then applying F to the result or applying the associator in $\mathscr{D}$ to the images through F of the objects in $\mathscr{C}$. Same reasoning applies for left and right unitors, as depicted in Figures 3.8b and 3.8c.

The concept of a lax monoidal functor is one of the weakest ways to relate monoidal categories. In the following definition, we will refine this concept requiring more properties to be satisfied, making the way monoidal categories are related to each other increasingly stronger.

Definition 3.5.2 (Symmetric, strong, strict monoidal functors)**.** *A lax monoidal functor $F : (\mathscr{C}, \otimes, I) \to (\mathscr{D}, \Box, I')$ is said to be:*

- Symmetric *if F preserves symmetries, meaning that the diagram in Figure 3.9 also commutes;*
- Strong *if both φ and ϵ are natural isomorphisms;*
- Strict *if both φ and ϵ are equalities. In this case we have*

$$FI = I' \qquad F(A \otimes B) = FA \Box FB$$

Remark 3.5.3 (strictness of symmetries follows from strictness.)**.** Note that, if F is symmetric and strict, strictness and the diagram in Figure 3.9 automatically imply

$$F\sigma_{A,B} = \sigma_{FA,FB}$$

We will see important examples of monoidal functors, both lax and strong, in the remainder of this document.

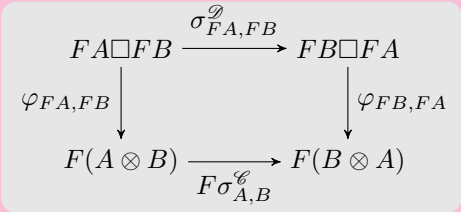


Figure 3.9: Additional coherence condition for lax symmetric monoidal functor.

3.6 Products, coproducs, pushouts

Now we introduce another very well known concept in category theory. The arguments covered here are just a small fragment of a much more developed theory, and are particular instances of *limits* and *colimits*. Due to the risk of losing the reader's attention, we refer one to [2, Ch. 2] and [18, Ch. 3] for a fully-detailed coverage of the story.

Let's think about the category of sets and functions, **Set**. In Examples 3.3.3 and 3.3.4 we already mentioned the concepts of a *cartesian product* and *disjoint union* of sets, and we highlighted how these constructions can be used to define different symmetric monoidal structures on **Set**. But what is a cartesian product? And a disjoint union? Do we have a way to capture these notions purely categorically, that is, without making any explicit reference to elements?

In principle, we would be tempted to say "no". The main idea when dealing with cartesian products is that if we have two sets A, B then we are able to consider *the set of couples*:

$$A \times B := \{(a, b) \mid a \in A, b \in B\}$$

This definition makes explicit use of elements, so how can we restate it just in terms of sets and functions? Surprisingly, it turns out that there is a way, as we are about to show.

First things first, we note that if we have a cartesian product $A \times B$ then we have a couple of functions, usually called *projections*, that "forget" about one side of the product:

$$\begin{aligned} \pi_1 : A \times B &\to A \\ (a, b) &\mapsto a \end{aligned} \qquad\qquad \begin{aligned} \pi_2 : A \times B &\to B \\ (a, b) &\mapsto b \end{aligned}$$

Moreover, we also note that every time we have two functions $f : C \to A$ and $g : C \to B$, we can construct a function $f \times g$ pairwise, setting

$$\begin{aligned} \langle f, g\rangle : C &\to A \times B \\ c &\mapsto (f(c), g(c)) \end{aligned}$$

We also see quite easily that, by definition,

$$\langle f, g\rangle ; \pi_1 = f \qquad \langle f, g\rangle ; \pi_2 = g$$

All this information is indeed enough to capture the idea of cartesian product of sets, and can be presented as follows:

Example 3.6.1 (Products in **Set**)**.** For any two sets A, B, there exists a set $A \times B$, together with functions $\pi_1 : A \times B \to A$, $\pi_2 : A \times B \to B$, such that every time we have another set C and a couple of functions $f : C \to A$, $g : C \to B$, there is *one and only one function*, denoted with $f \times g$, that makes the diagram in Figure 3.10 commute.

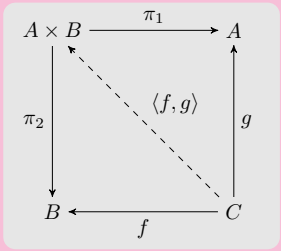


Figure 3.10: Universal property for products.

But now this definition of product doesn't make use of elements at all, and we can use it for any category!

Definition 3.6.2 (Products)**.** *Let $\mathscr{C}$ be a category. We say that $\mathscr{C}$ has* products *when the condition stated in Example 3.6.1 holds for any couple of objects A, B and for any couple of morphisms $C \to A$, $C \to B$.*

Example 3.6.3 (Product of categories)**.** It is not hard to see that **Cat**, the category of all small[1] categories and functors between them, admits a product structure. Given categories $\mathscr{C}, \mathscr{D}$, their product $\mathscr{C} \times \mathscr{D}$ can be defined as just

$$\mathrm{obj}\ \mathscr{C} \times \mathscr{D} := \mathrm{obj}\ \mathscr{C} \times \mathrm{obj}\ \mathscr{D} \qquad \mathrm{Hom}_{\mathscr{C}\times\mathscr{D}} := \mathrm{Hom}_{\mathscr{C}} \times \mathrm{Hom}_{\mathscr{D}}$$

With source and target defined component-wise as

$$\mathrm{s}((f,g)) := (\mathrm{s}(f), \mathrm{s}(g)) \qquad \mathrm{t}((f,g)) := (\mathrm{t}(f), \mathrm{t}(g))$$

Note how we used this product in Definition 3.3.1 to define the functor $\otimes$.

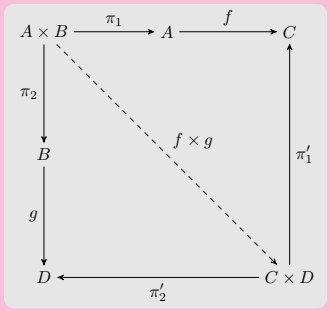


Figure 3.11: Product of morphisms f, g.

Example 3.6.4 (Product of morphisms)**.** We can make immediate use of the property of products, as follows: Suppose that we have morphisms $f : A \to C$, $g : B \to D$. *Thanks to the property of $C \times D$,* we can obtain a unique morphism $f \times g : A \times B \to C \times D$ setting

$$f \times g := \langle \pi_1; f, \pi_2; g \rangle$$

[1] There are issues in considering the category of all categories that make the theory inconsistent, exactly as it happens in set theory. To solve this, we have to restrict ourselves to particular types of categories, called *small* categories. All the categories usually considered in ordinary mathematics are small, so this is not a big deal for us!

where π_1, π_2 are the projections from $A \times B$ to A, B, respectively, as in Figure 3.11. This is exactly what allowed us to use the cartesian product to define a monoidal structure in Example 3.3.3, but holds in general: In any category with products, the product defines a monoidal structure.

Similarly, we can characterize the idea of "disjoint union" categorically, as follows:

Definition 3.6.5 (Coproducts)**.** *A category $\mathscr{C}$ has* coproducts *if, for every couple of objects A, B, there is an object $A \sqcup B$ together with morphisms (called* injections*) $i_1 : A \to A \sqcup B$ and $i_2 : B \to A \sqcup B$ such that, for each couple of morphisms $f : A \to C$ and $g : B \to C$, there is* one and only one *morphism $[f, g] : A \sqcup B \to C$ that makes the diagram in Figure 3.12 commute.*

Given two morphisms $f : A \to C$ and $g : B \to D$ we can, as for products, obtain a morphism $f \sqcup g : A \sqcup B \to C \sqcup D$ by setting:

$$f \sqcup g := [f; i_1, g; i_2] \tag{3.6.1}$$

Where i_1, i_2 are the injections from C, D to $C \sqcup D$, respectively.

Note that the usual disjoint union of sets respects the condition given above. Moreover, we are now able to see how disjoint union (categorically known as *coproduct*) and the cartesian product (categorically just known as *product*) are somehow connected: The definition of coproduct is just the same as the one of product, but *with all the arrows reversed!*

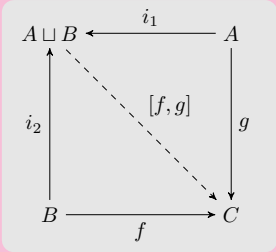


Figure 3.12: Universal property for coproducts.

Remark 3.6.6 (Coproducts induce monoidal structures)**.** It is again true that, in any category with coproducts, the coproduct can be used to define a monoidal structure. As in the case of products, this is implied by Equation 3.6.1. We saw this explicitly with the category **Set**, in Example 3.3.4.

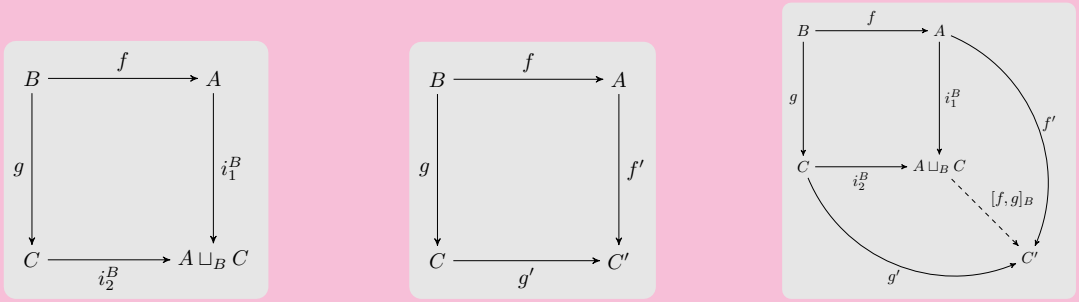
The product and coproduct construction, respectively, are said to be built by means of *universal properties*. Intuitively, the idea of universal property is that for each set of "preconditions" – whatever this means depends on context – there is *exactly one morphism* that makes some diagram commute.

There is another universal construction (that is, a categorical construction made by means of universal properties) that is worth mentioning which we will use quite a bit later on. This construction is called *pushout* and works as follows:

Definition 3.6.7 (Pushout)**.** *A category $\mathscr{C}$ has* pushouts *if, for each couple of morphisms $f : C \to A$ and $g : C \to B$, there is an object $A \sqcup_B C$ and morphisms $i_1^B : A \to A \sqcup_B C$, $i_2^B : C \to A \sqcup_B C$ that make the diagram in Figure 3.13a commute.*

Moreover, if we have morphisms $f' : A \to C'$ and $g' : C \to C'$ such that the diagram in Figure 3.13b commutes, then there is a unique morphism (here is where the universal property kicks in) $[f, g]_B : A \sqcup_B C \to C'$ such that the diagram in Figure 3.13c commutes too.

Note how in the pushout case we are conceptually going backwards: Before, we took set-theoretic concepts and we generalized them to arbitrary categories. Now, instead, we are giving a categorical definition, and in principle we don't even know if there are categories that satisfy it or, more specifically, if the category **Set**, on which we based many examples, does.



(a) (b) (c)

Figure 3.13: Universal property for pushouts.

This is a very important point when working abstractly, viz. while giving categorical definitions that are not based on specific examples we already know. Every time we give a categorical property we have to check:

- If the category we are interested in satisfies that property;
- How can that property be explicitly described in the category.

In the case of **Set**, we are indeed lucky.

Example* 3.6.8 (Pushouts in **Set**)**.** The category **Set** indeed has pushouts. The pushout of morphisms $f : B \to A$,and $g : B \to C$ can be characterized as follows:

$$A \sqcup_B C := A \sqcup C/\sim$$

where $\sim$ is the smallest equivalence relation that identifies $a \in A$ with $c \in C$ if there is some $b \in B$ such that $a = f(b)$ and $c = g(b)$. i_1^B (i_2^B, respectively) sends every element of A (of C, respectively) to its equivalence class. It is easy to see that, for each $b \in B$, it is true by definition that $f; i_1(b) = g; i_2(b)$.

To conclude, we note that in making the definition of a pushout explicit in **Set** we used the disjoint union, which we know is the coproduct in **Set**. We may hypothesize that these two things are somehow connected and this is indeed true in any category that has both pushouts and coproducts.

Remark 3.6.9 (Coproducts and pushouts are connected)**.** The pushout of morphisms $f : B \to A$ and $g : B \to C$ gives us morphisms $i_1^B : A \to A \sqcup_B C$, $i_2^B : C \to A \sqcup_B C$. In a category that has both pushouts and coproducts, we can then consider the situation in Figure 3.14, where the existence and uniqueness of the diagonal arrow is guaranteed by the universal property defining coproducts.

So, every time we have pushouts of some morphisms f, g and coproducts in a category, we always have a unique morphism connecting the two.

Example* 3.6.10. In the category **Set**, as we already saw, the unique morphism in Figure 3.14 is the one sending each element of the disjoint union to its equivalence class with respect to the relation $\sim$ defined in Example 3.6.8.

3.7 Why is this useful?

We admit that it is difficult to answer to this question at this stage. This chapter has been very dense in terms of mathematical definitions and results, and quite poor in terms of applicative purposes and

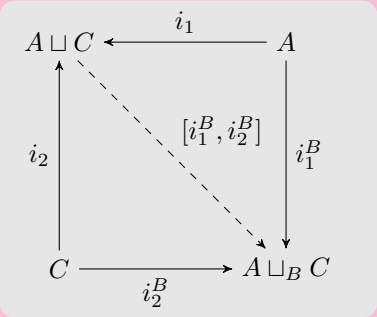


Figure 3.14: Interaction between coproducts and pushouts.

examples. We could not do much better than this, since the learning curve for category theory is steep and one needs to build quite a bit of machinery to successfully employ in modeling real-world problems.

The best we can do for now is reassure the audience that the fruits of such an involved reading will be reaped very soon, and limit ourselves to the following considerations:

- Categories are ubiquitous, and we can do all the known mathematics with them. The concepts of functors and natural transformations are very powerful, and allow us to establish formally consistent links between mathematical theories. We understand that if we have a categorical definition of Petri nets – that we will work out in Chapter 4 – and a categorical definition of some other meaningful tool we want to use, then we can employ our categorical techniques to combine these two together, as we will do in Chapter 5;

- Categories have a clear operative interpretation, allowing us to talk about processes happening in series or in parallel. This makes category theory readily applicable in the context of software design, as we will see in Chapter 5. What we lack is the idea of processes competing for resources, as we had for Petri nets, and this is exactly why we want to categorize them, bringing together the best of both worlds;

- As in the case of Petri nets, monoidal categories admit a neat graphical formalism. This means once more that in implementing a programming language based on monoidal categories we get a visual way to code/debug for free. The debugging functionality in particular is very good, since code that may be pretty hard to read is often translated to straightforward images, as we saw for the Eckmann-Hilton argument in Example 3.4.2.

How proficuous category theory will be for us will already become clear in the next Chapter, where we will use category theory to turn Petri nets into fully deterministic structures.

Chapter 4

Executions of Petri nets

Up to now, we introduced two main concepts: Petri nets – in Chapter 2 – and category theory – in Chapter 3. We moreover promised that the two things would be related, and that we would have used the second one to help us reasoning with the first. In this Chapter we start honoring this promise, modeling the executions of a Petri net categorically.

4.1 Problem overview

Consider the images in Figure 4.1, describing the evolution of a Petri net.

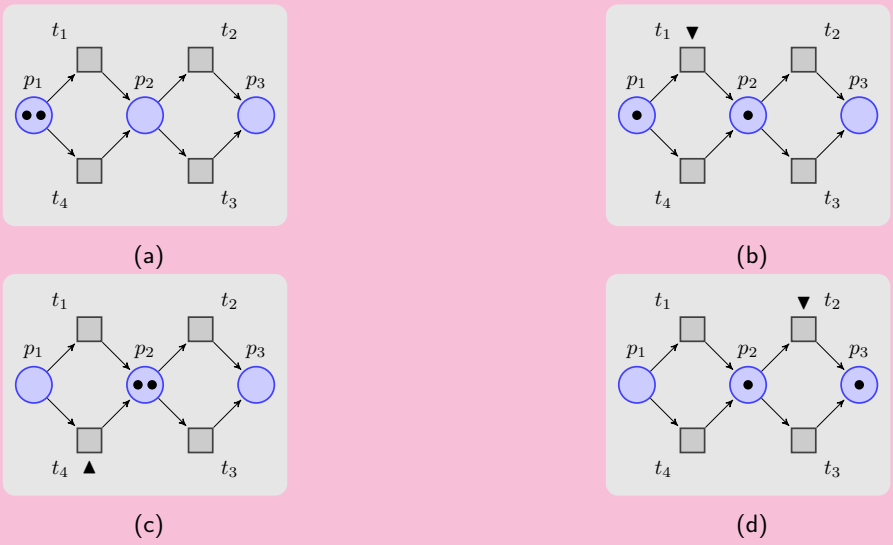


Figure 4.1: Evolution of a net.

In Figure 4.1b transition t_1 has fired. If we name the markings in Figures 4.1a, 4.1b, 4.1c and 4.1d respectively as $X^{\mathbb{N}}, Y^{\mathbb{N}}, Z^{\mathbb{N}}, K^{\mathbb{N}}$, the notation

$$X^{\mathbb{N}} \xrightarrow{t_1} Y^{\mathbb{N}} \xrightarrow{t_4} K^{\mathbb{N}} \xrightarrow{t_2} Z^{\mathbb{N}}$$

doesn't help us understand which one of the two tokens in p_2 the transition t_2 is consuming: It is impossible to say if the token t_2 is consuming has been previously produced by transition t_1 or by

transition t_4. This is problematic if we think of transitions as processes that consume and produce resources: A token represents a resource of the type of the place it is in, but these resources are not necessarily all the same. If a place in a net holds resources of type **Bool**, for instance, then tokens in that place will be boolean entities, meaning that they can either represent the value **True** or the value **False**. Similarly, if a place holds resources of type **Int** then each token represents an integer number, and two tokens in the same place may be different from each other. It is evident that it is not enough to say that a transition in a net consumes a given token to infer what is really happening: *Distinguishing between tokens becomes important*, since different tokens will be processed differently by transitions.

Example 4.1.1 (Different histories give different results)**.** Suppose that all the places of the net in Figure 4.1 hold resources of type **Int**, viz. integer numbers. Interpret the transitions of the net as functions from integers to integers:

$$t_1(x) = x + 1 \qquad t_2(x) = x \qquad t_3(x) = x^2 \qquad t_4(x) = 2x$$

Consider the tokens in Figure 4.1a to both represent the number 2. It is evident that the token processed by t_1 has value 3, while the token processed by t_4 has value 4. Since t_2 is the identity on integers, applying t_2 to one token or the other will produce different results, namely:

$$t_2(t_1(2)) = 3 \neq 4 = t_2(t_4(2))$$

Remark 4.1.2. Petri nets have been explicitly designed not to distinguish between tokens. This non-deterministic behavior – i.e. not knowing which token a transition is processing – is intended, since the formalism is concerned only with studying structural properties of distributed, concurrent systems, and abstracting from such details comes in handy. But this is unsuitable if one wants to use Petri nets to actively design complex infrastructure.

Now that we convinced ourselves that distinguishing between tokens is important, we still have to figure out how to do it.

Remark 4.1.3 (Distinguishing between tokens)**.** To distinguish between tokens, the most appropriate criterion that we can think of is that *tokens are considered to represent the same resource if they have the same history*, meaning that they have been processed by the same transitions. This indeed makes sense, since it is the only way we have to distinguish between tokens *within* the net: Consider again Figure 4.1a. Here we have two tokens in p_1, but we don't know anything else about them: Surely, they may represent different resources but we have no way to infer this from the behavior of the net, since the tokens "were already there" before we started executing it. From within the net, these tokens may then be considered equal, since any additional information is not accessible.

Now that we understood what it means to consider two tokens equal or not, we want to formalize this mathematically. We will do this following [24]:

- We will organize Petri nets in a category, called $\mathbf{Petri}^{\mathbb{N}}$;
- To each Petri net N we will associate a category, denoted with $\mathfrak{F}(N)$, representing *all the possible histories of all the possible tokens in the net*;
- We sketch how this correspondence is functorial and can be extended to an equivalence of categories, linking nets with executions in a reversible fashion.

Notice how our modus operandi represents very well the philosophy behind category theory that we highlighted in the previous Chapter: Category theory is the study of patterns, and we want to create a correspondence between Petri nets and their possible evolutions in a way that is compatible with the way nets interact: Patterns have to be preserved, hence we want a functor.

Remark 4.1.4. Albeit conceptually simple, the procedure in this chapter requires quite a lot of technicalities, that we have omitted for exposition purposes. We redirect the reader to [24] for details. We hope the reader will forgive us as we tried to simplify the mathematics here as best as we could, but only succeeded partially.

4.2 The category $\mathbf{Petri}^{\mathbb{N}}$

Our first task is to organize Petri nets into a category. There are many different ways to do this, all useful, and a big part of the upcoming versions of this monograph will be dedicated to investigate how these different ways to categorize Petri nets relate to each other. For now, we proceed by requiring Petri nets to be the objects of the category we want to define. If nets are objects, then we need a suitable notion of morphism between nets. To do this, we recall the definition of Petri net:

Definition 2.3.1 (Petri net)**.** *A* Petri net *is a quadruple*

$$N := (P_N, T_N, {}^\circ(-)_N, (-)^\circ_N)$$

Where:

- P_N *is a* finite *set, representing places;*
- T_N *is a* finite *set, representing transitions;*
- P_N *and* T_N *are disjoint: Nothing can be a transition and a place at the same time;*
- ${}^\circ(-)_N : T \to \mathscr{M}^{\mathbb{N}}_{P_N}$ *is a function assigning to each transition the multiset of* P *representing its* input *places;*
- $(-)^\circ_N : T \to \mathscr{M}^{\mathbb{N}}_{P_N}$ *is a function assigning to each transition the multiset of* P *representing its* output *places.*

We will often denote with $T_N, P_N, {}^\circ(-)_N, (-)^\circ_N$ *the set of places, transitions and input/output functions of the net* N*, respectively.*

So we see that what we have are places, transitions and input/output functions. A suitable notion of morphism between nets will have to involve at least some of these objects. Consider nets $(P_N, T_N, {}^\circ(-)_N, (-)^\circ_N)$ and $(P_M, T_M, {}^\circ(-)_M, (-)^\circ_M)$. One of the most naive things to do is to send transitions to transitions, defining a function $f : T_N \to T_M$ representing a correspondence between processes of N and processes of M. Similarly, it makes sense to send places of N to places of M, that is, to define a function $g : P_N \to P_M$. This means that the resource types in N will correspond to resource types in M. Notice, though, that since processes consume and produce resources in places, f and g have to be somehow connected: If $t \in T_N$ is sent to $f(t) \in T_M$, then it must be that if $p \in {}^\circ(t)_N$ (respectively, $p \in (t)^\circ_N$), then $g(p) \in {}^\circ(g(t))_M$ (respectively, $f(p) \in (g(t))^\circ_M$), otherwise our correspondence will make no sense. This suggests that if we define g to be a function $\mathscr{M}^{\mathbb{N}}_{P_N} \to \mathscr{M}^{\mathbb{N}}_{P_M}$ then we are able to use g to express the compatibility conditions for input/output functions stated above. Moreover, since g carries multisets on P_N to multisets on P_M, it automatically specifies a function $P_N \to P_M$, since P_N, P_M are just multisets that send every element of P_N, P_M, respectively, to 1. This shows how this new definition of g encompasses the naive one.

But is a function $\mathscr{M}^{\mathbb{N}}_{P_N} \to \mathscr{M}^{\mathbb{N}}_{P_M}$ enough to get a suitable notion of morphism between nets? Not quite. Multisets are not just sets, and the firing rule for multisets (recall Definition 2.3.7) is defined in terms of multiset sum and difference. This forces us to require additional properties if we don't want our correspondence to misbehave with respect to the firing rules of $(P_N, T_N, {}^\circ(-)_N, (-)^\circ_N)$ and $(P_M, T_M, {}^\circ(-)_M, (-)^\circ_M)$. The definition we need is the one of *multiset homomorphism*, and it is stated below:

Definition 4.2.1 (Multiset homomorphism)**.** *Consider $\mathscr{M}_P^{\mathbb{N}}$ and $\mathscr{M}_{P'}^{\mathbb{N}}$, the sets of multisets on P and P', respectively. A* multiset homomorphism *is a function $f : \mathscr{M}_P^{\mathbb{N}} \to \mathscr{M}_{P'}^{\mathbb{N}}$ such that*

$$f(\emptyset_P) = \emptyset_{P'} \qquad f(P_1^{\mathbb{N}} + P_2^{\mathbb{N}}) = f(P_1^{\mathbb{N}}) + f(P_2^{\mathbb{N}})$$

for each $P_1^{\mathbb{N}}, P_2^{\mathbb{N}} \in \mathscr{M}_P^{\mathbb{N}}$, that is, a multiset homomorphism is a function between $f : \mathscr{M}_P^{\mathbb{N}} \to \mathscr{M}_{P'}^{\mathbb{N}}$ that carries the zero multiset to the zero multiset and respects multiset sums.

Remark* 4.2.2 (Multisets homomorphisms are free monoid homomorphisms)**.** The reader fluent in algebra, recalling Remark 2.2.7, will have noticed that a multiset homomorphism is just a homomorphism of free commutative monoids.

The definition of multiset homomorphism is perfect, since it will allow us to carry the input (output, respectively) function of a net N to the input (output, respectively) function of a net M in a way that respects the firing rules of both nets. We are ready to give the definition we were seeking:

Definition 4.2.3 (Petri nets morphisms)**.** *Given nets $(P_N, T_N, {}^\circ(-)_N, (-)^\circ_N)$, $(P_M, T_M, {}^\circ(-)_M, (-)^\circ_M)$, a* morphism of Petri nets *$M \to N$ is specified by a pair $\langle f, g \rangle$ where:*

- *f is a function $T_N \to T_M$;*
- *g is a multiset homomorphism $\mathscr{M}_{P_N}^{\mathbb{N}} \to \mathscr{M}_{P_M}^{\mathbb{N}}$;*
- *Diagrams in Figure 4.2 commute.*

Figure 4.2: Properties of net moprhisms.

This definition neatly packs all the discussion above, and the diagrams in Figure 4.2 represent the idea that transitions in $(P_N, T_N, {}^\circ(-)_N, (-)^\circ_N)$ and transitions in $(P_M, T_M, {}^\circ(-)_M, (-)^\circ_M)$ are organized in a compatible way.

Remark 4.2.4 (Net morphisms are simulations)**.** One way to interpret a morphism of nets is in terms of *simulations*. since a morphism of nets $N \to M$ is made of a couple of *functions*, multiple transitions (multisets, respectively) of N can correspond to the same transition (multiset, respectively) of M. This means that transitions and places of M hit by the morphism act as placeholders for transitions and places of N, and we can interpret this as if the process represented by M contains a subprocess simulating the one represented by N.

It is easy to see that, for each net N, there is a pair $\langle id_{T_N}, id_{\mathscr{M}_{P_N}^{\mathbb{N}}} \rangle$ that sends everything to itself. Similarly, if $\langle f, g \rangle : N \to M$ and $\langle f', g' \rangle : M \to K$ are net homomorphisms, then $\langle f; f', g; g' \rangle$ is a net homomorphism $N \to K$, and morphism composition is associative. Then we can define:

Definition 4.2.5 (The category $\mathbf{Petri}^{\mathbb{N}}$)**.** *We define the category $\boldsymbol{Petri}^{\mathbb{N}}$ as having Petri nets as objects and morphisms between them as morphisms.*

Now that we managed to organize our nets into a category, it is time to get to the next step.

4.3 The Execution of a net

Let us focus on the transitions of a net. There are fundamentally two ways in which transitions can interact:

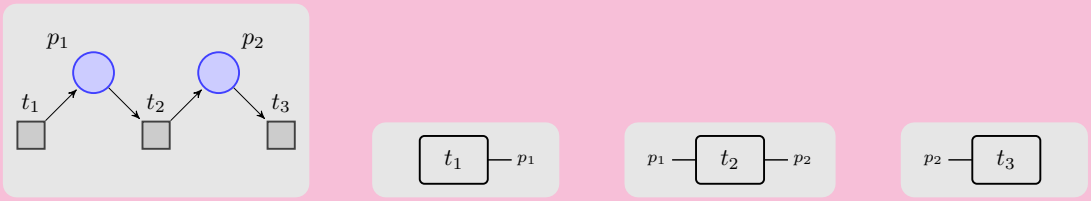
- The firing of one transition is independent from the firing of the other (e.g. transitions t_1, t_4 in Figure 4.1);
- The firing of a transition depends on the firing of the other (e.g. transitions t_1, t_2 in Figure 4.1).

This should clearly suggest that a *monoidal category* (recall Definition 3.3.1) is the structure we want to represent transition firings, since it comes with a notion of sequential and parallel composition, representing presence or absence of interaction between transitions. Let us see how we can use symmetric monoidal categories to represent an execution.

Consider a net N, and a monoidal category $\mathscr{C}$ such that each place of N corresponds to an object in $\mathscr{C}$. To avoid clutter, we will denote the places and the objects they correspond to in the same way. Consider then a place $p \in P_N$. The object p can be thought of as representing a token in p. Using the tensor product, we can iterate this: $p \otimes p$ represents two tokens in p; $p \otimes p \otimes p$ represents three tokens in p, and so on. Similarly, if we have two places $p, q \in P_N$, then $p \otimes q$ stands for one token in p and one in q.

Notice that, according to this idea, $p \otimes q \otimes p$ and $p \otimes p \otimes q$ both represent having two tokens in p and one in q: Tokens are just being considered in a different order and then we should have a way to go from one object to the other. This means that we want our category to be *symmetric* (recall Definition 3.3.7).

Now, transitions. We can fully embrace our idea of transitions as processes carrying resources into other resources, saying that *a transition $t \in T_N$ corresponds to a morphism $t : {}^\circ(t)_N \to (t)^\circ_N$ in $\mathscr{C}$*. But this means that sequences of transitions are now just string diagrams! This has huge benefits, since using a string diagram we can represent which transition is consuming which tokens, and observing the wiring we can reconstruct how a single token is processed. Since we do not care about how we bracket parallel composition, it is clear that we want our category to be also strict (recall Definition 3.3.10).



(a) A net.

(b) Morphism $t_1 : I \to p_1$

(c) Morphism $t_2 : p_1 \to p_2$

(d) Morphism $t_3 : p_2 \to I$

Figure 4.3: A net and its morphisms.

Example 4.3.1. Consider the net in Figure 4.3a. Its transitions can be represented as the morphisms of strict symmetric monoidal category as in Figures 4.3b,4.3c and 4.3d and all the possible string diagrams, as, for instance, the ones in Figure 4.4, represent sequences of transition firings. We also see how the monoidal unit, that is not drawn in the pictures according to our convention (see Section 3.4), is useful to represent transitions with no inputs and/or no outputs. Each set of vertically aligned wires in the diagram represents a state of the net, and transitions carry states into states. Note how this allows us to completely disambiguate the problem of distinguishing tokens. For instance, in Figure 4.4b, t_1 produces a token in p_1, and t_2 consumes a token from p_1 as well. But the diagram states clearly how these tokens are not the same: t_2 is consuming a token that was already present in p_1 before t_1 fired.

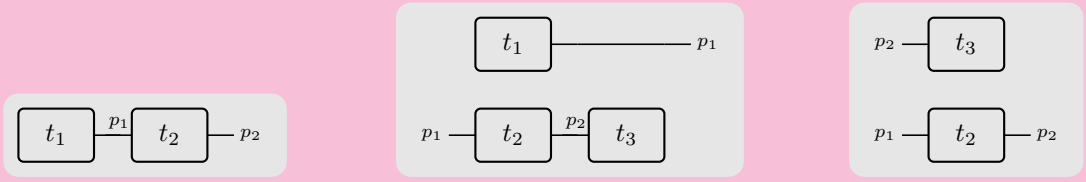


Figure 4.4: Some of the possible sequences of firings for the net in Figure 4.3

Remark 4.3.2. We realize quite quickly why we need our category to be strict (Def. 3.3.10) symmetric (Def. 3.3.7) monoidal (Def. 3.3.1): Observe Figure 4.5. It is not important, at this stage, to know to which net this diagram may be referring to. We start from the state $A \otimes B \otimes C$, meaning that we have one token in A, one token in B and one token in C. Then transitions t_1 and t_2 fire, and we can represent this event as simultaneous since the two transitions have nothing to do with each other, hence firing priority doesn't matter in this situation. What matters is that the state produced is $D \otimes E$, so one token in D and one in E. Now, transition t_3 has to fire, but it is expecting a state $E \otimes D$: This is morally the same thing, one token in E and one token in D, but since category theory distinguishes between these two objects, we have to introduce a swapping morphism to make this composition possible.

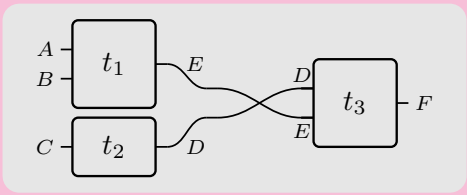


Figure 4.5: Swaps are necessary to do the required bookkeeping.

Remark 4.3.3. Looking at our string diagrams closely, we get quickly aware of the fact that *firing sequences cannot correspond to string diagrams uniquely*: If the category describes all the possible ways to execute the net, it is clear that the same firing sequence can correspond to different diagrams. This is consistent with the idea that the category provides additional information that the net cannot capture, namely token histories. To see this, consider Figures 4.1. The string diagrams in Figure 4.6 are all legitimate executions describing the same sequence of firings.

Figure 4.6: Some of the possible sequences of firings for the net evolution in Figures 4.3

Now that we grasped how to represent computations of a net categorically, at least at an intuitive level, we are ready to formalize this. Up to now, in fact, we just stated how the category of executions should

look like, but we didn't build it. There are many categories which may be good candidates to represent a net execution, so how do we pick one? We will find out in the next section, while we conclude the present one with a summary of what we learned so far.

Places	correspond to	Objects
States	correspond to	Monoidal products of objects (places)
Transitions	correspond to	Morphisms
Firing sequences	correspond to	String diagrams

4.4 The category $\mathfrak{F}(N)$

Given a net N, we want to generate a category representing all the possible ways to execute N and we know, thanks to the previous section, that this category has to be strict symmetric monoidal to allow us to do sequential, parallel composition and all the necessary bookkeeping given by swapping tokens around. For now, let us focus on a simpler task that will come in handy later on, that is, let us try to characterize *the most general* strict symmetric monoidal category we can think of. We start with a definition.

Definition 4.4.1 (Strings generated by a set)**.** *Let S be a set. We denote the set of strings of finite length of elements in S as $S^{\oplus}$:*

$$S^{\oplus} := \{s_1 s_2 \dots s_n \mid n \in \mathbb{N} \wedge \forall i, s_i \in S\}$$

We see that we can think about $S^{\oplus}$ also as *the set of all possible monoidal products of elements in S*: If S denotes a set of objects in a monoidal category, then we can interpret a string $s_1 \dots s_n$ as $s_1 \otimes \dots \otimes s_n$: String concatenation stands for monoidal product, and the empty string stands for the monoidal unit, which is consistent with the idea that $s \otimes I = s$ for each object s. From this we can infer a very important fact, that will be useful later on:

Remark 4.4.2 ($S \in \mathrm{obj}\ \mathscr{C}$ implies $S^{\oplus} \in \mathrm{obj}\ \mathscr{C}$)**.** If S is a set, and we map each element of S to an object in a strict symmetric monoidal category, then each element of $S^{\oplus}$ can be mapped in that category too. This is obvious, since a monoidal category is closed by monoidal products.

If we think of S as the set of places of a net, and we want to define a category $\mathscr{C}$ representing all the possible ways to execute it, then it makes sense to require that S, and all the possible monoidal products of elements in S, are objects of $\mathscr{C}$. This is because we know, from the previous section, that we are going to model states of a net as monoidal products of tokens, represented by the place they live in. From this we realize that $S^{\oplus}$ seems to be a good candidate to define $\mathrm{obj}\ \mathscr{C}$: In this case, the objects of $\mathscr{C}$ are *just* the states or, to be precise, all the possible ways to enumerate states.

What else do we need to get a strict symmetric monoidal category? Not much: Just *identites and symmetries*. We need identity morphisms if we want to define a category, because the axioms require it. Similarly, the presence of symmetries is required by axioms for symmetric monoidal categories. But we do not need anything else: These and all their possible compositions are the only morphisms that are "obligatory" according to the axioms. We can give, then, the following definition:

Definition 4.4.3 (Free strict symmetric monoidal category generated by S)**.** *Let S be a set. The category $\mathscr{S}_S$ is called* the free strict symmetric monoidal category generated by S*. It has elements of $S^{\oplus}$ as objects, and its arrows are generated by the rules:*

$$\frac{u \in S^{\oplus}}{id_u : u \to u} \qquad \frac{u, v \in S^{\oplus}}{\sigma_{u,v} : u \otimes v \to v \otimes u} \tag{4.4.1}$$

$$\frac{\alpha : u \to v,\ \beta : u' \to v'}{\alpha \otimes \beta : u \otimes u' \to v \otimes v'} \qquad \frac{\alpha : u \to v,\ \beta : v \to w}{\alpha ; \beta : u \to w} \tag{4.4.2}$$

modulo the axioms that make it into a strict symmetric monoidal category:

$$\alpha; id_v = \alpha = id_u; \alpha \qquad (\alpha;\beta);\gamma = \alpha;(\beta;\gamma) \tag{4.4.3}$$

$$id_u \otimes id_v = id_{u\otimes v} \qquad (\alpha\otimes\alpha');(\beta\otimes\beta') = (\alpha;\beta)\otimes(\alpha';\beta') \tag{4.4.4}$$

$$I\otimes\alpha = \alpha = \alpha\otimes I \qquad (\alpha\otimes\beta)\otimes\gamma = \alpha\otimes(\beta\otimes\gamma) \tag{4.4.5}$$

$$\sigma_{u,v\otimes w} = (\sigma_{u,v}\otimes id_w);(id_v\otimes\sigma_{u,w}) \qquad \sigma_{u,v};\sigma_{v,u} = id_{u\otimes v}$$

$$\sigma_{u,u'};(\beta\otimes\alpha) = (\alpha\otimes\beta);\sigma_{v,v'} \qquad (\alpha : u\to v \wedge \beta : u'\to v')$$

This is a big, meaty definition, so let's try to unpack it. This category represents all the possible bookkeeping we can do between tokens, and nothing more: Objects, as we said, correspond to all the possible ways to represent a state, following the intuition from Section 4.3. Again, we define $\otimes$ as string concatenation on the objects. Now, the morphisms: Rules are read top to bottom: Every time a condition expressed above the line is realized, the condition expressed below it is inferred. Rules 4.4.1 tell us that for each object u we get a morphism $id_u : u \to u$ that will – unsurprisingly – be our identity morphism. Similarly, for each couple of objects u, v of $\mathscr{S}_S$, we get a morphism $\sigma_{u,v} : u\otimes v \to v\otimes u$, that will be our symmetry. Rules 4.4.2, on the other hand, populate our $\mathscr{S}_S$ with all the possible parallel and sequential compositions of morphisms. These four rules define all the possible morphisms in $\mathscr{S}_S$, and by iterating them one quickly becomes aware of how any morphism in $\mathscr{S}_S$ is just a big composition – parallel and sequential – of identities and symmetries.

To understand the second part of the definition, notice this: Up to now, we get a morphism id_u for each object u, but nothing assures us that this morphism behaves as an identity: We have id_u because we defined a rule that formally introduces it, but the rule doesn't say anything about its behavior. The behavior has to be formally imposed by identifying morphisms with each other. Axioms in 4.4.3 are the necessary identification to obtain a category, since they entail identity and associativity axioms to hold. Axioms in 4.4.4 entail that $\otimes$ is a functor, while axioms 4.4.5 imply that this functor defines a strict monoidal structure. Finally, the remaining axioms are needed to have $\mathscr{S}_S$ symmetric.

Remark 4.4.4 (Axioms and rules are all necessary)**.** Note that all the axioms and rules in Definition 4.4.3 are necessary: If we strip out only one of these ingredients then it is not possible anymore to prove that $\mathscr{S}_S$ is strict symmetric monoidal. In this sense, $\mathscr{S}_S$ is the most general free strict monoidal category containing S among its objects, since it has no superfluous components of any kind. This generality is also proved by the following property, called *freeness*.

Remark 4.4.5 (Freeness)**.** If $\mathscr{C}$ is a strict symmetric monoidal category, then for each function $f : S \to \mathrm{obj}\ \mathscr{C}$ *there is a unique strict symmetric monoidal functor* $F : \mathscr{S}_S \to \mathscr{C}$ *extending* f, meaning that, for each $s \in S$, $Fs = f(s)$. Observe how this amounts to a generalization of Remark 4.4.2 from objects to the entire category.

This is great, because it says that every time we map elements of S in the objects of a strict symmetric monoidal category, all the structure of $\mathscr{S}_S$ "follows along" by means of a uniquely determined functor. In this sense, the strict symmetric monoidal structure of $\mathscr{S}_S$ is completely determined by S.

For now, we have all the necessary ingredients to represent all the possible bookkeeping on the places of a net. We only have to introduce morphisms representing transitions. Before doing that, we need a last definition.

Remark 4.4.6 (Multiplicity)**.** There is an obvious mapping $\mathfrak{M} : S^{\oplus} \to \mathscr{M}_S^{\mathbb{N}}$ that associates to each string $str \in S^{\oplus}$ a multiset $S \to \mathbb{N}$:

$$\mathfrak{M}(str)(s) := \text{ Occurrences of } s \text{ in } str$$

What $\mathfrak{M}$ does is very simple. You give it a string, you give it an element in S, and it counts how many times the element occurs in the string. Let us put this definition to good use, finally formalizing what we are interested in.

Definition 4.4.7 (The category $\mathfrak{F}(N)$)**.** *Let $N := (P_N, T_N, {}^\circ(-)_N, (-)^\circ_N) \in \mathrm{obj}$ **Petri**$^{\mathbb{Z}}$. We define $\mathfrak{F}(N)$ (called* the category of executions of N*) to be the strict symmetric monoidal category having elements of $S^\oplus$ as objects, and arrows generated by the rules:*

$$\frac{t \in T_N}{t_{u,v} : u \to v} \qquad \forall u, v.(\mathfrak{M}(u) = {}^\circ(t)_N \wedge \mathfrak{M}(v) = (t)^\circ_N) \tag{4.4.6}$$

$$\frac{u \in S^\oplus}{id_u : u \to u} \qquad \frac{u, v \in S^\oplus}{\sigma_{u,v} : u \otimes v \to v \otimes u}$$

$$\frac{\alpha : u \to v,\ \beta : u' \to v'}{\alpha \otimes \beta : u \otimes u' \to v \otimes v'} \qquad \frac{\alpha : u \to v,\ \beta : v \to w}{\alpha ; \beta : u \to w}$$

modulo the axioms that make it into a strict symmetric monoidal category:

$$\begin{aligned} \alpha ; id_v = \alpha = id_u ; \alpha \qquad & (\alpha ; \beta) ; \gamma = \alpha ; (\beta ; \gamma) \\ id_u \otimes id_v = id_{u \otimes v} \qquad & (\alpha \otimes \alpha') ; (\beta \otimes \beta') = (\alpha ; \beta) \otimes (\alpha' ; \beta') \\ I \otimes \alpha = \alpha = \alpha \otimes I \qquad & (\alpha \otimes \beta) \otimes \gamma = \alpha \otimes (\beta \otimes \gamma) \\ \sigma_{u, v \otimes w} = (\sigma_{u,v} \otimes id_w) ; (id_v \otimes \sigma_{u,w}) \qquad & \sigma_{u,v} ; \sigma_{v,u} = id_{u \otimes v} \\ \sigma_{u,u'} ; (\beta \otimes \alpha) = (\alpha \otimes \beta) ; \sigma_{v,v'} \qquad & (\alpha : u \to v \wedge \beta : u' \to v') \end{aligned}$$

and the additional family of axioms:

$$p ; t_{u',v'} ; q = t_{u,v} \qquad \forall p, q.(p \in \mathrm{Hom}_{\mathscr{S}_{P_N}}[u, u'] \wedge q \in \mathrm{Hom}_{\mathscr{S}_{P_N}}[v', v]) \tag{4.4.7}$$

Again, this is a big definition, but many parts of it are now familiar. The rule 4.4.6 is the only new one, and tells us that for each transition of the net, we introduce *a family of morphisms* in the category. Why a family and not just one? Well, the net identifies inputs and outputs of transitions with multisets, while the category uses strings. This means that for each multiset there are many strings corresponding to it (all permutations of each other), so which one do we pick to define domain and codomain of the morphisms in $\mathfrak{F}(N)$? Instead of introducing arbitrary choices, we pick *all of them*, and this is precisely what rule 4.4.6 is telling us.

Figure 4.7: Axioms 4.4.7, graphically. We wrote $u_1u_3u_2u_4$ instead of $u_1 \otimes u_3 \otimes u_2 \otimes u_4$ to avoid clutter, and similarly for the other subscripts.

Axioms 4.4.7, instead, tell us how these morphisms are related. $\mathrm{Hom}_{\mathscr{S}_{P_N}}[u, u']$ stands for "all the morphisms in $\mathscr{S}_{P_N}$ from u to u'". Since $\mathfrak{F}(N)$ contains all the identities and symmetries, each morphism $u \to u'$ in $\mathscr{S}_{P_N}$ is also in $\mathfrak{F}(N)$ (if you want a more formal way to understand this, note that it is sufficient to apply Remark 4.4.5 to $\mathfrak{F}(N)$). So, essentially, this axiom reads as: "If you reshuffle inputs and outputs of a morphism coming from a transition of the net N through rule 4.4.6, you again get

a morphism coming from the same transition through rule 4.4.6". This is perhaps better explained graphically, as in Figure 4.7.

These axioms say that all the $t_{u,v}$ are just variations of the same thing, one for each different bookkeeping context. All the other axioms are as in Definition 4.4.3, and are needed to get a genuine strict symmetric monoidal category.

Remark 4.4.8 (Why not impose $u \otimes v = v \otimes u$?)**.** The reader may have noticed that the biggest source of cumbersome technicalities comes from the fact that objects of $\mathfrak{F}(N)$ do not commute with respect to the monoidal product (i.e. $u \otimes v \neq v \otimes u$). This means that for each multiset $S^{\mathbb{N}}$ there are a lot of different strings $s \in S^{\oplus}$ such that $\mathfrak{M}(s) = S^{\mathbb{N}}$. This forces us to send transitions to families of morphisms and so on, so one could ask: Why don't we just impose that, for each couple of objects u, v,

$$u \otimes v = v \otimes u$$

as we did for the other axioms, and solve the problem? We surely can do this, but unfortunately it has been proved in [24, Thm. 2.2] that, in this case, there is no way to make our correspondence $N \to \mathfrak{F}(N)$ functional. This means that in doing so we would give a definition that doesn't take into consideration the way Petri nets relate with each other, which is something we don't want. This is the reason why we chose the hard path and had to deal with such a large amount of technical difficulties.

4.5 Functors between executions

Now we have a correspondence that assigns, to each N, a category $\mathfrak{F}(N)$. We also know that Petri nets are the objects of a category $\mathbf{Petri}^{\mathbb{N}}$, so we ask: Is it possible to extend $\mathfrak{F}(-)$ to a functor? This amounts to asking if there is some appropriate notion of morphism between categories of the form $\mathfrak{F}(N) \to \mathfrak{F}(M)$ such that, to each morphism of nets $N \to M$, corresponds a morphism $\mathfrak{F}(N) \to \mathfrak{F}(M)$.

First of all, to define a functor, we need a couple of categories. We already have one, that is $\mathbf{Petri}^{\mathbb{N}}$. But where do the $\mathfrak{F}(N)$ live? They are strict symmetric monoidal categories, so they surely live in **SSMC**, the category that has strict symmetric monoidal categories as objects and strict symmetric monoidal functors between them as morphisms. So we can reformulate our question asking if we can generalize $\mathfrak{F}(-)$ to a functor $\mathbf{Petri}^{\mathbb{N}} \to \mathbf{SSMC}$.

We won't go into much detail here, for which we redirect the reader to [24], but we will try to sketch how the functor $\mathfrak{F}(-)$ should behave on morphisms.

Example 4.5.1 (Transition-preserving functors)**.** Imagine we have a transition in a net N, call it t. According to Axiom 4.4.6, it will correspond to a family of morphisms $t_{u,v}$ in $\mathfrak{F}(N)$. Now suppose we have another net M and a morphism $\langle f, g \rangle : N \to M$. We want to use the information in $\langle f, g \rangle$ to define a strict symmetric monoidal functor $\mathfrak{F}(\langle f, g \rangle) : \mathfrak{F}(N) \to \mathfrak{F}(M)$. This is intuitively simple: The functor has to map identities to identities, symmetries to symmetries, and any morphism $t_{u,v}$ to a morphism $f(t)_{g(u),g(v)}$. This last requirement means that our functor should send morphisms coming from the transition $t \in T_N$ to morphisms coming from the transition $f(t) \in T_M$, and backs up our idea that the functor should mimic what f does. We call functors that map families of morphisms coming from a transition to families of morphisms coming from a transition *$\overline{\text{transition}}$-preserving*, where we are using an overline notation to avoid ambiguities since this is a property of functors between categories and not of morphisms between nets.

Remark* 4.5.2 ($\overline{\text{Transitions}}$ are natural transformations)**.** Here we elaborate more on Example 4.5.1. All the $t_{u,v}$ coming from the same transition in N can be identified with some particular natural transformations between a couple of functors in $\mathfrak{F}(N)$. This correspondence is unique, and allows us to speak of a $\overline{\text{transition}}$ in $\mathfrak{F}(N)$ referring to the correspondent natural transformation. We can then say that a strict symmetric monoidal functor $\mathfrak{F}(N) \to \mathfrak{F}(M)$ *preserves $\overline{\text{transitions}}$* if it carries components of

$\overline{\text{transitions}}$ in $\mathfrak{F}(N)$ to components of $\overline{\text{transitions}}$ in $\mathfrak{F}(M)$, meaning that if $\{t_{u,v}\}$ defines a $\overline{\text{transition}}$, so does $\{Ft_{Fu,Fv}\}$. Details about this can be found in [24].

Example 4.5.3 (Extending $\mathfrak{F}(-)$ to a functor implies choice)**.** The problem with Example 4.5.1, though, is that there are many ways to define $\mathfrak{F}(\langle f, g\rangle)$. Suppose, for instance, that a transition t has two inbound edges, from places u and v, respectively, having weight 1, and no outbound edges. According to our rules, then, we will have two morphisms

$$t_{u\otimes v,I} : u \otimes v \to I \qquad t_{v\otimes u,I} : v \otimes u \to I$$

in $\mathfrak{F}(N)$. Now suppose that we have the identity morphism $\langle id_{T_N}, id_{\mathscr{M}_{P_M}^{\mathbb{N}}}\rangle : N \to N$. According to the recipe in Example 4.5.1, $\mathfrak{F}(\langle id_{T_N}, id_{\mathscr{M}_{P_M}^{\mathbb{N}}}\rangle)$ should map the family of morphisms $\{t_{u\otimes v,I}, t_{v\otimes u,I}\}$ to itself. But there are clearly two ways of doing this:

$$\begin{array}{llll} t_{u\otimes v,I} \mapsto t_{u\otimes v,I} & t_{v\otimes u,I} \mapsto t_{v\otimes u,I} & \text{Mapping 1} \\ t_{u\otimes v,I} \mapsto t_{v\otimes u,I} & t_{v\otimes u,I} \mapsto t_{u\otimes v,I} & \text{Mapping 2} \end{array}$$

These two markings define different functors $\mathfrak{F}(N) \to \mathfrak{F}(N)$ that can both be taken to be $\mathfrak{F}(\langle id_{T_N}, id_{\mathscr{M}_{P_M}^{\mathbb{N}}}\rangle)$, so how do we choose?

The problem highlighted in Example 4.5.3 is that, as usual, $\mathfrak{F}(N)$ has many more objects than N, and hence a morphism $N \to M$ corresponds to many possible morphisms in $\mathfrak{F}(N) \to \mathfrak{F}(M)$, each one dealing with the "object surplus" in a different way. But we note that our two functors are morally equivalent: They can be turned one into the other by applying some symmetries, that is, just by means of ordinary bookkeeping.

Remark 4.5.4 (Identical functors)**.** We can say that two $\overline{\text{transition}}$-preserving functors $F, G : \mathfrak{F}(N) \to \mathfrak{F}(M)$ are *identical* if there is a natural transformation (Definition 3.2.9) between them having only compositions of symmetries and identities as components. This means that the two functors behave in the same way on morphisms representing transitions, and differ only on the irrelevant bookkeeping.

Remark* 4.5.5 (Being identical is a congruence)**.** Being identical defines an equivalence relation between $\overline{\text{transition}}$-preserving functors that is compatible with functor composition, hence a congruence.

Another problem we notice is that the $\mathfrak{F}(N)$ are very special instances of strict symmetric monoidal categories. There could be categories in **SSMC** that are not in the form $\mathfrak{F}(N)$ for some N, and this should tell us that the category **SSMC** is probably too big: It also contains categories that have nothing to do with net executions, and functors that are not $\overline{\text{transition}}$-preserving. The way to solve these problems is as follows:

- We restrict ourselves to a *subcategory* of **SSMC**. This means, albeit we are not being 100% precise, that we consider only some of the objects and arrows of **SSMC**. We call this category **ExPetri**$^{\mathbb{N}}$;
- We sketch how **ExPetri**$^{\mathbb{N}}$ looks like:
  - It has objects of the form $\mathfrak{F}(N)$ for some net N, or objects isomorphic to them. Intuitively, this means that all the objects in **ExPetri**$^{\mathbb{N}}$ "look like a $\mathfrak{F}(N)$"
  - Morphisms are strict $\overline{\text{transition}}$-preserving functors, where moreover we consider identical functors to be the same morphism. This means that two strict symmetric monoidal functors F, G are considered the same morphism in **ExPetri**$^{\mathbb{N}}$ if, for each family of morphisms coming from a transition $t_{u,v}$, $F(t_{u,v})$ and $G(t_{u,v})$ define the same family.

- Since we selected only some objects and some arrows of **SSMC**, things may break: We have to prove then that **ExPetri**$^{\mathbb{N}}$ is again a category.
- Having done this, we try to define $\mathfrak{F}(-)$: **Petri**$^{\mathbb{N}} \to$ **ExPetri**$^{\mathbb{N}}$. On objects, we associate to each net N the category $\mathfrak{F}(N)$ as we defined it in Section 4.4. It remains to define it on morphisms. If we have a morphism of nets $\langle f, g\rangle : N \to M$, proceeding as in Example 4.5.1 we get a family of identical, transition-preserving functors $\mathfrak{F}(\langle f, g\rangle) : \mathfrak{F}(N) \to \mathfrak{F}(M)$ sending the family $t_{u,v}$ to the family $f(t)_{g(u),g(v)}$. But from the definition of **ExPetri**$^{\mathbb{N}}$ these functors turn out to be all the same morphism in **ExPetri**$^{\mathbb{N}}$, so we are really sending $\langle f, g\rangle : N \to M$ to one morphism in **ExPetri**$^{\mathbb{N}}$, making things well-defined.
- At this point we just have to prove that this assignment respects composition and identities, and we obtain that $\mathfrak{F}(-)$ is a functor.

Albeit handwavy, the procedure above gives an idea of how to turn $\mathfrak{F}(-)$ into a functor. But there is more:

Remark 4.5.6 (The functor $\mathfrak{U}(-)$)**.** There is also a functor $\mathfrak{U}(-)$: **ExPetri**$^{\mathbb{N}} \to$ **Petri**$^{\mathbb{N}}$ that turns a category of executions into its correspondent net. These functors are somehow one the inverse of the other, meaning that

- For each net N in **Petri**$^{\mathbb{N}}$, $\mathfrak{U}(\mathfrak{F}(N))$ is isomorphic to N (following Definition 3.1.10, there is an invertible morphism of Petri nets $\mathfrak{U}(\mathfrak{F}(N)) \to N$);
- For each category $\mathscr{C}$ in **ExPetri**$^{\mathbb{N}}$, $\mathfrak{F}(\mathfrak{U}(\mathscr{C}))$ is isomorphic to $\mathscr{C}$ (again following Definition 3.1.10, in **ExPetri**$^{\mathbb{N}}$ there is an invertible morphism $\mathfrak{F}(\mathfrak{U}(\mathscr{C})) \to \mathscr{C}$).

What is even nicer, is that we can use $\mathfrak{F}(-)$ and $\mathfrak{U}(-)$ to prove that **Petri**$^{\mathbb{N}}$ and **ExPetri**$^{\mathbb{N}}$ are equivalent (recall Definition 3.2.8), showing once and for all how Petri nets and their executions are just two different ways to look at the same thing.

Remark* 4.5.7 (**ExPetri**$^{\mathbb{N}}$ and **Petri**$^{\mathbb{N}}$ are equivalent)**.** We have an adjunction $\mathfrak{F}(-) \vdash \mathfrak{U}(-)$. Remark 4.5.6 proves that the unit and counit of this adjunction are isos, and thus define an equivalence of categories.

The right way to interpret Remark 4.5.6 is that if we apply the couple of functors $\mathfrak{F}(-), \mathfrak{U}(-)$ in any order, we get back to something that behaves like the object we started with. The fact that we are not getting back to *exactly* the same object should not be considered as negative: As usual in category theory, we are interested in studying the behavior of objects in terms of patterns. Getting back to an object that is *isomorphic* to the object we started with means that, from the point of view of the patterns that we deem relevant, the two objects have exactly the same behavior.

So, in the end, we succeeded in our task: We linked each Petri net, in a unique way, to a category representing all the possible ways to move tokens around it. This result is invaluable and, as we will explain more in detail in Chapter 5, will allow us to turn Petri nets into a way to produce software.

4.6 Beyond standard Petri nets

Up to now, we considered normal Petri nets and categorically described their executions. But what happens if we change our notion of Petri net? A nice change to make would be, for instance, to allow the net to have *negative tokens*. If we represent a token as a black dot in a place, we can represent a negative token as *red*; we can moreover consider transitions that consume/produce negative tokens (Figure 4.8a). We call a net that allows for negative tokens an *integer Petri net*.

(a) An integer Petri net.

(b) Integer tokens.

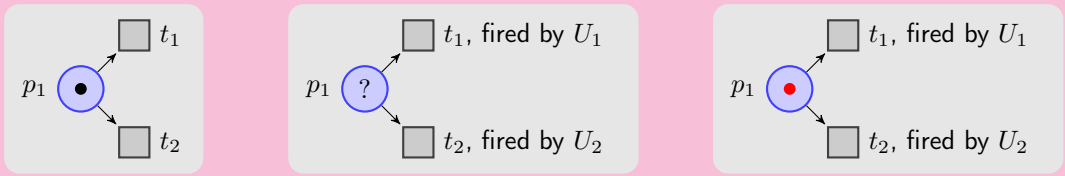
(c) Before firing.

(d) After firing.

Figure 4.8

If we start to explore this definition further, we see that very strange things can happen now. Since clearly a negative token and a positive one "annihilate", exactly as $-1+1=0$, we can produce couples of tokens in any place, as in Figure 4.8b. The consequence of this is that, as in Figures 4.8c and 4.8d, now transition can fire borrowing tokens from a place, and so are always enabled!

Is there a use for this generalization? Most likely, yes. In fact, the study of executions of integer Petri nets is a genuine contribution of the Statebox team to academic research, that resulted in a paper [11]. What motivated us to investigate in this direction is that integer nets can be useful to model conflict resolution in concurrent behaviour. Consider, for instance, the net in Figure 4.9a: We know how transitions t_1 and t_2 have to compete for the token in p_1 and, at least in the case of standard nets, they cannot both fire. Now suppose that there are two users, say U_1 and U_2, that can operate on the net, deciding which transition to fire. When a user takes a decision, it is broadcast to the other one, and the overall state of the net is updated. In a realistic scenario, though, broadcasting takes time (imagine, for instance, that our users have bad internet connections): User U_1 could decide to fire t_1 and user U_2 could decide to fire t_2 while the broadcast choice of U_1 has still to be received, putting the overall net into an illegal state (Figure 4.9b).



(a)

(b)

(c)

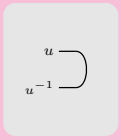
Figure 4.9

In such a situation we need a way to re-establish *consensus*, that is, decide unambiguously in which legal state the net is. There are multiple ways to do this, but our main concern here is that the usual Petri net formalism does not have a way to represent illegal states, which is fundamental to attacking the problem mathematically. With integer Petri nets we are able to easily represent such a situation using negative tokens, as in Figure 4.9c. Intuitively, we can say that a net is in an illegal state if the state contains a negative number of tokens in some place, and re-establishing consensus from an illegal state then amounts to getting back to a state where the number of tokens in each place is non-negative. Clearly, the fact that any net can now fire just by borrowing positive tokens by its input places is consistent with the idea that, for whatever reason, every transition can put the net into an illegal state. There are, even, transitions like the one in Figure 4.8a that de facto "produce illegal states" out of thin air. This could be used, for instance, to represent a faulty component in our net architecture.

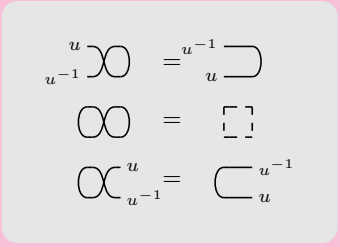
Since the category of executions of a net carries much more information than the net itself (we can track precisely the history of each token in the net), it makes sense to study this category to see if there are naive ways to resolve an illegal situation, at least in some cases. The category of executions of an integer net looks very similar to what we already saw in Section 4.4, and we do not have to change much of what we already have. First, we need to add a new couple of bookkeeping morphisms in our

formalism, along with identities and symmetries. These are depicted as a cup (Figure 4.10a) and a cap (Figure 4.10b) and represent the creation or annihilation of couples of negative and positive tokens in a place. These new morphisms have to satisfy the axioms in Figures 4.10c and 4.10d (these last couple of axioms are called *yanking* or *snake equations* [6], for obvious visual reasons). Strict symmetric monoidal categories that have cups and caps and respect such axioms are called *strict compact closed categories*[16].

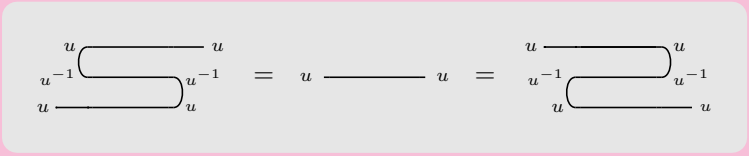
(a) Cup.



(b) Cap.



(c) Axioms for cup and cap.



(d) Snake equations.

Figure 4.10: Additional structural morphisms and axioms for executions of integer nets.

Among other differences, we have that Axioms 4.7 now assume the form in Figure 4.11. This is to be expected: We are just adding more bookkeeping to our category, and as morphisms coming from transitions weren't bothered by bookkeeping before, they aren't now!

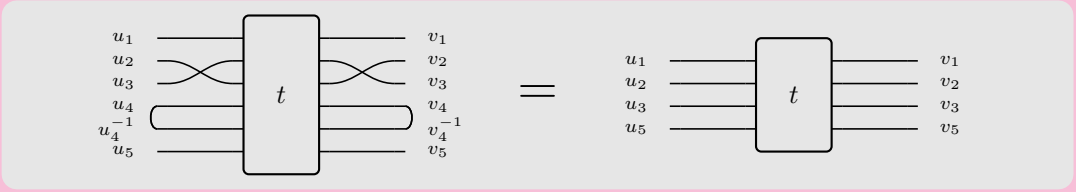


Figure 4.11: The revised version of Axioms 4.7. Subscripts have been omitted to avoid clutter.

.

What is really interesting, though, is not to dive into the categorical nuances of integral Petri nets executions, but to see how they solve some of the problems regarding nets in illegal states by default. Consider the situation in Figure 4.12: As before, imagine that a user U_1 fires transition τ, while another user fires transition ν putting the net into an illegal state. We can represent this graphically in our category of executions introducing a cup that produces a pair of positive and negative tokens. At this point we apply a morphism corresponding to ν and carry the "positive part" produced by the cup to Y.

Now comes the interesting part: Suppose that some other user fires transition μ. This transition produces a token in X that *effectively cancels* the debt left in X by the firing of ν, reporting the net into an legal state. But this sequence of firings is still not acceptable, since the firing of ν couldn't have happened in the first place!

Nevertheless, the categorical model offers us a solution straight out of the box: When the token

produced by μ lands in X, it annihilates the negative token left there. In the category of executions, this amounts to add a cap to our string diagram. But now the magic happens: We can straighten the string diagram using the snake equations obtaining the sequence of firings τ, then μ, then ν.

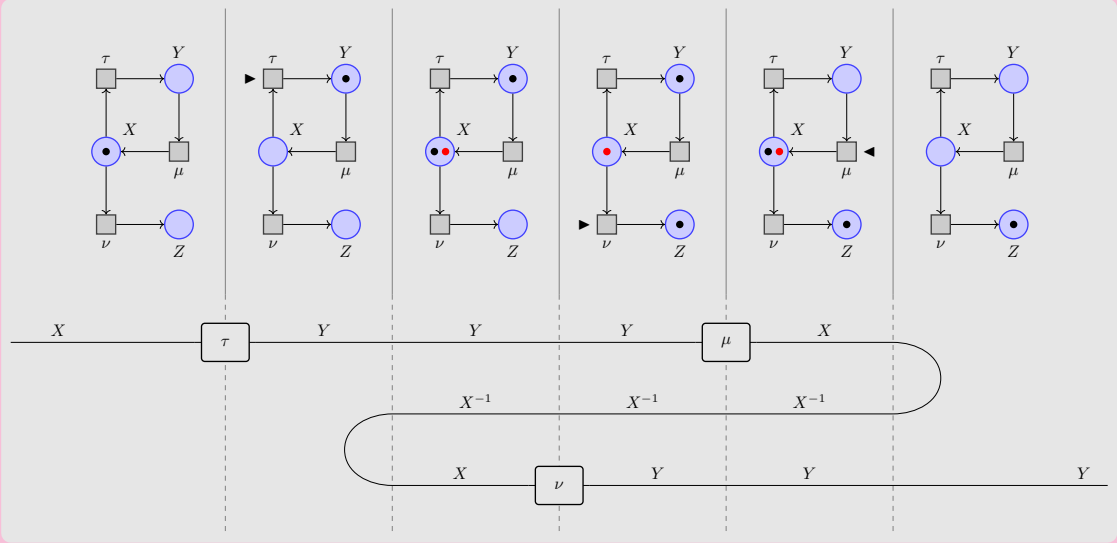


Figure 4.12: Conflict resolution using integer Petri nets.

The right way to read the diagram in Figure 4.12 is as follows: The vertical lines divide different instants in time *in the real world*. If we were to attach a timestamp to each transition firing, we would actually observe that τ has been fired, then ν has been fired, and finally μ has been fired. On the contrary, the wire represents *the causal flow of the network itself*: It doesn't matter which transitions have been fired first in the real world, the flow represented by following the wire – namely τ, then μ, then ν – is the flow that doesn't break causality, allowing for a sequence of *completely legal firings*. The category of executions then offers naive solutions to re-establish consensus by reshuffling the order of transition firings, cancelling out any illegal state.

If we observe the net carefully, we notice that, actually, we still need to establish consensus on something: We presumed that user U_1 fired τ *before* user U_2, but obviously user U_2 is not of this opinion, otherwise he wouldn't have fired ν in the first place!

This means that we need a way to establish which of the two users fires first, which is still a consensus problem. We argue, though, that this kind of consensus is much simpler to reach (for instance implementing a global clock) than having to reach consensus on an entire merging problem such as the one of re-establishing the causal order of transitions is. The category of executions for integer Petri nets takes care of this part of the problem for us, and needs only a very small amount of consensus to work properly.

All the details about this construction can be found in [11], where we proved results that are akin to the ones in Section 4.5. Namely, we arranged integer Petri nets in a category, called **Petri**$^{\mathbb{Z}}$. We did the same for categories of executions, obtaining a category of categories **ExPetri**$^{\mathbb{Z}}$. Finally, we proved that the mapping of integer nets to their executions gives us a couple of functors $\mathfrak{F}(-)$, $\mathfrak{U}(-)$ that behave exactly like their counterparts in Section 4.5, giving us an equivalence of categories.

Remark 4.6.1. Now the reader can finally understand why we denoted the categories in Section 4.5 as **Petri**$^{\mathbb{N}}$ and **ExPetri**$^{\mathbb{N}}$, respectively: The $\mathbb{N}$ used as superscript denotes that in those categories we are considering only positive tokens, distinguishing them by their counterparts **Petri**$^{\mathbb{Z}}$ and **ExPetri**$^{\mathbb{Z}}$!

The study of integer Petri nets is in its infancy, and many questions have yet to be answered. The "conflict resolution procedure" sketched in this section, for instance, is only of theoretical interest at the moment and far away from an industry-strength implementation but we will keep pushing in this direction in the hope of obtaining something that can eventually become a useful feature to be used in our language.

It has to be noted that integer Petri nets also present very nice characteristics when one tries to model transaction flows and money flows in general. This is relevant for a number of applications, among which are Blockchain-based techniques [20], for which it is commonplace to represent any kind of asset – even computations – by monetizing it [5], and the general characterization of economic phenomena in terms of process theories, which is the object of a broader research that the Statebox team is carrying out along with multiple partners, and which also involves open games [12], macroeconomics [27] and open systems [26].

4.7 Why is this useful?

The answer to this question should be pretty clear: Executions allow us to track which transitions process which tokens, and to formalize the idea of "history of a net". Being able to represent the causal relationships between firings precisely and reliably is fundamental to concatenate processes in a meaningful way, and categories of executions, serving exactly this purpose, will function as a bridge to consistently link nets, seen as abstract design tools for complex systems, to the actual implementation – we will start developing this point of view in Chapter 5. Note how leveraging this formal bridge is exactly what makes Statebox different from any other project based on Petri nets. Petri nets have, in fact, been used as design tools for software many times in the past, but the general modus operandi was as follows:

- The programmer would draft how the software about to be written was supposed to work in the abstract, using a Petri net. The properties of the net would be formally studied to ensure some pre-set performance standards;
- Afterwards, code would be produced, using the net implementation as a guide. This passage would be totally handmade and there would be no formal link between the net and the actual codebase. All things considered, the formal relationship between nets and code would amount to zero, and using nets to design it was not much different than sketching flow diagrams on a piece of paper: Helpful, but needing a lot of common sense to be implemented properly;
- As a consequence, it would happen that the software implementation could not properly reflect the net topology due to human error, and performing even small modifications in the net layout would result in huge code refactoring.

Category theory, on the other hand, completely automates all these steps giving us a neat way to represent net executions. In the first release of the Statebox language, we will implement a simplified version of this, since designing a data structure to store net executions in an efficient way is not a trivial task. This problem will obviously be fixed in the upcoming releases, and we will be able to achieve, few months from now, total representability of net executions on the code side.

Chapter 5

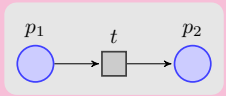
Folds

In this chapter we will start reaping what we sow up to now. Many of the concepts presented in the previous chapters, purely theoretical on their own, will start displaying evident applicative potential when put together. Here we will start sketching the actual plan to turn Petri nets and category theory into a useful software development toolkit.

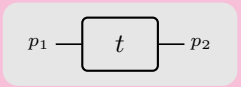
We devoted Chapter 4 to the endeavor of building categories associated with Petri nets. We pointed out how the main reason to do this was to be able to describe the net behavior in a completely deterministic way, keeping track of the history of any token. In truth, we can do much more than this: If for each net N we have a category $\mathfrak{F}(N)$, we can use our categorical intuition and do the most sensible thing to do when you have a category, namely mapping it to somewhere by means of a functor.

5.1 Problem Overview

A perfectly legitimate question, at this stage, is: Why should we map executions to other categories? Note how, at the moment, both Petri nets and their executions can't do much. We are able to draw a net and, as we said, we interpret its transitions as processes that, when firing, consume and produce resources, but where is this information stored? Clearly nowhere, at least for the moment. This interpretation exists only in our mind and is not backed up by any meaningful mathematics. Similarly, in defining executions, we said that we associate, to each transition, a family of morphisms representing the processes actually performed by the transition during firing. But again, this is an interpretation, since the actual definition of such processes is lacking in our category of executions. Recalling definition 4.4.7, we obtain the morphisms $t_{u,v}$, for each transition t, by means of an inference rule and some equations that just tell us how the $t_{u,v}$ behave with respect to the bookkeeping morphisms, but that's pretty much it! Our processes lack any sort of actual specification.



(a) A net representing quicksort.



(b) Morphism associated to Figure 5.1a.

Figure 5.1: Quicksort and its execution.

Example 5.1.1 (Quciksort)**.** Consider the net in Figure 5.1a. We interpret the places as holding resources of type **List[Int]**, that is, a token in a place represents a list of integers. Transition t represents an application of the *quicksort algorithm* [15], that sorts the list. This information is clearly not captured

by our net, which just describes how the transition turns one resource into another. Also our idea of tokens being of type **List[Int]** is overimposed, since the behavior of this data structure is not represented by the net (for instance, we cannot concatenate tokens or perform any sort of list operation on them). Similarly, in Figure 5.1b we represent the morphism associated to t in its category of executions. Again, in this setting, t is just "a box", and the information describing its behavior (namely, quicksort) is nowhere to be found.

Example 5.1.2 (Is quicksort being done right?)**.** An obvious counterargument to the reasoning in Example 5.1.1 could be that the net in Figure 5.1a doesn't capture the meaning of the quicksort algorithm because it is not the right model for it: It's not that nets are bad at representing such thing, it's that we have been bad at using them. Up to some extent, this is actually the case, since we can definitely try to model sorting algorithms in a much more convincing way using nets. What is worth to stress, though, is that this is often *not the right way of thinking about Petri nets*. The application of a mathematical gadget in computer science should, nearly always, serve the purpose of stripping away complexity and making things easier, possibly without giving up formal correctness and consistency of our methods. Does it make sense, then, to spend time to define something that already has fully debugged and efficient implementations, spanning just a few lines of code? The answer is clearly no, since what we'd get, at best, is something that needs a considerable amount of time and thought to get the level of performance found in existing solutions. Petri nets should make our life easier, and we would like to leverage already existing implementations of algorithms if we have them, as in the case of quicksort.

5.2 Mapping executions

What Example 5.1.1 entails could look like a huge downside: Our math is good for nothing, and our nets cannot do anything interesting without becoming really complicated. Luckily enough, this is not the case. What we obtained is, instead, far more valuable, and akin to what a logician would call *the separation between syntax and semantics*. We obtained a model of how our Petri nets behave without having to refer to any particular detail which complements the declarative functional approach we take in the implementation of our language. We can talk about quicksort, as in Example 5.1.1, without giving any specification of what quicksort does, aside of how it fits in the infrastructure we are designing, represented by the Petri net. The actual specification of quicksort (that is, its *semantics*) can be modeled separately in another category, and then be targeted appropriately by mapping the category of executions of the net into it. The fact that this mapping is functorial guarantees that syntax and semantics are being glued consistently together.

Example 5.2.1 (Design advantages)**.** The clear utility of this separation is that we can undertake our design efforts in stages. For instance, imagine that we are automating the infrastructure of an entire company. We first talk with people from various departments, asking them about their daily routines and needs. We then draft a Petri net describing how these processes interact with each other in real time, and since the Petri net formalism is completely graphical, people giving us this information even help us visually debug it, pointing out where the diagram representing a process in their workflow is incorrect. After having done this, we apply tools to study reachability problems on the net, verifying that it has the properties that we desire (for instance absence of deadlocks or illegal states, recall Definition 2.5.3). If these requirements are not met, then we can reshape the net until we get what we want. At this point – and only at this point – we can start writing down the code (typically in a lower level language) for the programs associated to each transition, and the functorial mapping from the net execution to the actual code takes care of putting everything together.

Having intuitively described the essence of folds, let us try to fix the concept with a definition.

Definition 5.2.2 (Folds)**.** *Given a Petri net N and a symmetric monoidal category $\mathscr{S}$, a* fold for N *is a symmetric lax monoidal functor (recall Definition 3.5.1) $\mathfrak{F}(N) \to \mathscr{S}$.*

The reason we are choosing a Lax monoidal functor is because it is the weakest requirement we can think of. It is always a good practice to state something in the greatest level of generality possible, and strengthen the requirements only if needed.

Before we start digging into the real stuff, notice that there is an obvious fold that we can take:

Remark 5.2.3 (Trivial Fold)**.** The identity functor $\mathfrak{F}(N) \to \mathfrak{F}(N)$ is trivially symmetric lax monoidal, hence it generates a *trivial fold* for N. Note that this remark is what prompted for the notation $\mathfrak{F}(-)$ to define the category of executions of a net. Executions are, to some extent, the simplest fold possible, where the meaning we attach to any execution of the net is the execution itself.

To create more complicated instances of folds, we need to create semantic categories to which it makes sense to map executions. A good starting point is to recall Example 3.1.5, and use algorithms written in a functional programming language (Haskell, in our case) as semantics.

Definition 5.2.4 (Haskell, again)**.** *The category **Hask** is defined as follows:*

- *Objects are data types. A data type is a way a computer uses to represent a certain kind of information. Common data types are **Bool**, consisting of the booleans **True** and **False**; **Int**, consisting of integer numbers, **List[Int]**, consisting of finite lists of integer numbers, and many other;*
- *Morphisms are* terminating haskell algorithms*, that is, algorithms that take terms of some data type as inputs , apply a sequence of operations that at some point terminates, and output a term of some data type as a result.*

*The category **Hask** can be made into a symmetric monoidal category using the natural cartesian product structure that data types and morphisms admit (a product of types A, B is just the type of couples (a, b) where a has type A and b has type B).*

Remark* 5.2.5 (Is Haskell a category?)**.** The reader with experience in the abstract theory of programming languages will have risen an eyebrow reading Example 5.2.4. In fact, the matter of defining the category **Hask** is quite a can of worms. In our case we are considering the strict symmetric monoidal category equivalent – via Remark 3.3.13 – to what is known in the functional programming folklore as the *platonic Haskell category*[14], where types do not have bottom values, valid morphisms are just terminating algorithms, algorithms are considered equal if they agree on all inputs and the use of `seq` is very limited. In general, the question of casting a category out of the Haskell programming language is still very debated, but we want to stress how this is not fundamental with respect to what we are going to do here. The point is that the fold from $\mathfrak{F}(N)$ to **Hask** is implementable, and offers a consistent way to map transitions into pieces of software.

Example 5.2.6 (The fold to **Hask**, in practice)**.** It is interesting to see how the mapping $\mathfrak{F}(N) \to$ **Hask** works in practice. Let's build a strict symmetric monoidal functor $F : \mathfrak{F}(N) \to$ **Hask**: Each place in P_N is also an object of $\mathfrak{F}(N)$, and as a consequence will be mapped by F to a particular Haskell data type. We have complete freedom in defining this mapping as we please. The mapping on monoidal products of objects will then have to follow since we set, by definition, $F(u \otimes v) = (Fu, Fv)$.

Next, the bookkeeping morphisms. Identities on a object u will be mapped to the algorithm $Fu \to Fu$ that takes any term of type Fu in input and outputs the term itself without changing it. Symmetries $\sigma_{u,v} : u \otimes v \to v \otimes u$ will be mapped to the algorithm that takes tuples (x, y) with x of type Fu and y of type Fv and outputs (y, x).

For each transition $t \in T_N$ we get, from Definition 4.4.7, a family of morphisms $t_{u,v}$ such that $\mathfrak{M}(u) = {}^\circ(t)_N$ and $\mathfrak{M}(v) = (t)^\circ_N$. Each one of these morphisms will be mapped to an Haskell algorithm

$Ft_{u,v} : Fu \to Fv$. Note that, since we are imposing strictness to our functor F, it is enough to define this mapping only for *one* $t_{u,v}$: Suppose that $t_{u,v} : u \to v$, for some fixed objects u, v, is mapped to $Ft_{u,v} : Fu \to Fv$. Now take $t_{u',v'}$: Since by definition

$$\mathfrak{M}(u) = {}^{\circ}(t)_N = \mathfrak{M}(u') \qquad \mathfrak{M}(v) = (t)^{\circ}_N = \mathfrak{M}(v')$$

The objects u, u' and v, v' are just permutations of the same objects. There are, then, symmetries $p : u' \to u$ and $q : v \to v'$ and, by Axioms 4.4.7, we obtain

$$t_{u',v'} = p; t_{u,v}; q$$

From here, we can just use functoriality and infer that

$$Ft_{u',v'} = F(p; t_{u,v}; q) = Fp; Ft_{u,v}; Fq$$

Where the components of the composition on the right-hand side are all already determined.

Remark 5.2.7 (Strictness makes life easier)**.** Note that, in Example 5.2.6, requiring F to be strict monoidal is what saved the situation, allowing us to define it only on places and transitions and leveraging strictness to extend it to all objects and morphisms. If we require F to be just lax, then we have much more choice to define it, which is a good thing on one hand, giving implementational freedom, but bad on the other, since we have to specify more things "manually" to make it work. The appropriate choice clearly depends on context.

Example 5.2.8 (Quicksort, continued)**.** We now turn our naive interpretation of Example 5.1.1 to something formal. Call N the net in Figure 5.1a. We define the fold $\mathfrak{F}(N) \to$ **Hask** mapping p_1 and p_2 to **List[Int]**, and t to the code:

```
quicksort :: [Int] -> [Int]
quicksort [] = []
quicksort (p:xs) = (quicksort lesser) ++ [p] ++ (quicksort greater)
        where
                lesser  = filter (< p) xs
                greater = filter (>= p) xs
```

Finally, t is now formally identified with quicksort!

Remark 5.2.9 (Terminating algorithms work better)**.** In Definition 5.2.4 we explicitly required morphisms of **Hask** to be *terminating algorithms*. It is worth to spend a few words on this: The interpretation of a fold is that transitions of a net get mapped to algorithms. Folds tell us which algorithm to run on which data when a transition fires. Clearly, in case the algorithm is not terminating, things break down: The transition in the net fires, but no tokens can be ever produced since the algorithm will hang forever. We get rapidly aware of how mapping transitions to algorithms that are not guaranteed to terminate *is not, in general, good practice*, since our formalism is no longer able to ensure consistency. This means that such "unsafe" mappings should be used only in very restricted contexts, when it is absolutely necessary, and in a very localized way, so that we are always able to keep track of which transitions in the net can exhibit a pathological behavior. Keeping the problem circumscribed is the easiest way to fix problems should they arise.

With respect to this, some functional programming languages as Idris [3], [4] offer the useful functionality of a *totality checker* already embedded in their compiler. What this means is that, for a restricted class of algorithms, the compiler is able to tell us if our algorithm will terminate on every input. This feature is incredible in the context of Folds, since it will allow us to map transitions to code that we know to be well behaved.

5.3 Different folds, shared types

In reviewing the material covered in the last section, we become aware that there is nothing special about using **Hask** as our semantics, and that in fact every functional programming language – or in general every language in which types can be defined – does, more or less, the job. One of the first things that comes to mind is: What happens if given a net N we chose $\mathfrak{F}(M)$, for some other net M, to define the semantics of a fold? We are not requiring, here, for this functor to be transition-preserving, as we did in Chapter 4. The idea is that we could map a single transition of N to an entire sequence of firings in M. This sort of "net inception" concept is very powerful, and backs up the intuition of transitions in a net triggering the execution of other nets as subprocesses, but needs far more work to be used properly. This will indeed be one of the main issues to be tackled in the next releases of this document, and we will put it on hold for now.

What we can already do with the tools developed so far is sketching how different folds relate to each other:

Definition 5.3.1 (Morphisms of folds)**.** *Given folds $F_1 : \mathfrak{F}(N) \to \mathscr{S}_1$, $F_2 : \mathfrak{F}(N) \to \mathscr{S}_2$, a* morphism of folds *is a symmetric lax monoidal functor $G : \mathscr{S}_1 \to \mathscr{S}_2$ such that $F_1; G = F_2$. Folds and their morphisms form a category.*

Remark* 5.3.2. The category of folds and their morphisms can be seen as the *co-slice category of strict symmetric monoidal categories and symmetric lax monoidal functors over $\mathfrak{F}(N)$.*

Embracing the interpretation of semantics in terms of data types and algorithms, a morphism of folds can be seen as a "translation" from one programming language to another, that allows us to rewrite our mapping altogether. This is not very useful in practice, since such translation exists very rarely since different programming languages have different properties.

What would be very useful, on the contrary, would be to have *all the programming languages modeled in the same category*. In fact, up to now, we are mapping executions into one programming language at a time, but this is not always what we would like to have. If our Petri nets represent a complex system, then transitions can represent processes radically different in nature, that would be better implemented in different programming languages, or, most likely, for which efficient implementations already exist in different languages.

With respect to this we want to be greedy, and be able to use as much preexisting stuff as we can. Experienced programmers know, in fact, that one of the biggest barriers in the adoption of a new programming language is having to rewrite entire libraries from scratch: This is not only time-consuming, forcing developers to spend many hours of good work on just preparing the software instead of using it to solve the problems at hand, but also very inefficient, since rewriting complex code is a tedious process that needs a lot of further testing. In real-life applications, there is virtually no porting of industry-strength products that works out of the box, and in translating libraries from one language to another one is almost always guaranteed sub-optimal performance – both in terms of time/space efficiency and presence of bugs and errors – for a big portion of the development stage. The ideal semantic category we would aim at, then, looks like this:

Definition 5.3.3 (Generalized semantics for folds)**.** *Let $\mathscr{L}_1, \ldots, \mathscr{L}_n$ denote programming languages. We define a category having:*

- *As objects, data types of $\mathscr{L}_1, \ldots, \mathscr{L}_n$;*
- *As morphisms, terminating algorithms between data types in the languages $\mathscr{L}_1, \ldots, \mathscr{L}_n$.*

This definition is obviously pathological, and will never serve any real purpose. We can see it directly considering the net in Figure 5.2: Suppose that transition t_1 has to correspond to some very efficient algorithm we want to use, written in Haskell. On the other hand, the best choice for t_2 would be to map it to some code written in Elm [7], [8]. This implies that the place p has to correspond to a data

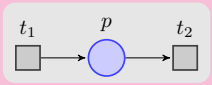


Figure 5.2: A net showing how Definition 5.3.3 is pathological.

type that is shared by Haskell and Elm, which is basically impossible since different languages implement data types differently.

For the same reason, defining a symmetric monoidal structure on the category in Definition 5.3.3 is next to impossible, since we don't even know what it means to take tuples of data types defined using different specifications. Luckily enough, there is a solution to this problem, that relies precisely on *defining a shared data type specification for many languages*. It is clear that such shared data types will have to be somehow limited, since types have different capabilities in different languages (for the experienced readers, notice how Idris has dependent types while Haskell does not, so they cannot be part of our shared type structure). This is something to which the Statebox team is devoting a lot of work, and its specification, called *Typedefs*, will be disclosed in the next versions of this monograph. Typedefs will make it easier to mix programming language and empower developers to use Petri nets to design their software without having to give up on the tools they already developed, which is a very desirable feature.

5.4 Why is this useful?

At this point, we are really tempted to just write "trivial." and conclude the Section. The usefulness of folds is self-evident: They allow for neat compartmentalization of development stages and, by giving the programmer freedom to chose the semantic category that suits their needs best, ensure full backwards compatibility. With Typedefs, finally, this compatibility can be seamlessly extended across different languages, finally allowing for a consistent linking between different layers of a complex system. From the point of view of industry-strength coding, especially in fail-sensitive applications, such features are simply invaluable, and put Statebox into a unique status among the myriad of programming languages out there. *As category theory is the glue of mathematics, Statebox is the glue of programming*.

Part II

Extending nets

Chapter 6

Open Petri nets

Up to now we have defined Petri nets (Chapter 2), categorified them, linked them with their executions (Chapter 4) and showed how this can be used to separate software topology from its meaning by means of a functorial mapping into a semantics (Chapter 5). This is great, but our software topology, at the moment modeled with Petri Nets, is still pretty limited and for this reason quite difficult to use in everyday programming. In more detail, all the nets defined up to now are *static*, in the sense that once the net is defined it cannot be changed, and the only form of dynamics we get is the one corresponding to its internal flow of tokens. To make nets usable in real-life programming scenarios, we need to introduce the following elements:

- Offline net composition, that allows us to obtain a bigger net from smaller ones at design stage. This kind of composition shouldn't happen at runtime;
- Net syncronization at runtime, that allows the binding of the firing of some transition(s) in a net with the firing of transition(s) in other net(s);
- Resource sharing at runtime, meaning that in a given net the tokens in some designated places should be shared with places in some other running nets.

Our main requirement is to obtain this extra expressiveness while sacrificing as little as possible in terms of the "nice properties" that our nets have. This ties in with a well-known problem, namely that for many useful generalizations of Petri nets some properties as boundedness, reachability and liveness become *undecidable* [17]. Such results apply in particular for generalizations towards hierarchic nets, that is, nets whose tokens represent nets themselves and so, quite counter intuitively, we cannot generalize our nets by introducing some form of abstraction – i.e. "holes" in a net to be filled with other nets. If we would, then we would no longer be able to prove if a given property – i.e. being live – holds. Clearly such an outcome has a huge drawback for us, and de facto means that, albeit more expressive, our representation of software topology in terms of nets would no longer help the coder understand the behavior of the software being designed.

To solve this, we will proceed in a very cautious way, generalizing one small step at a time and embracing a paradigm based on message layers by modeling net composition and synchronization, in a way similar to the one found in [1]. Once our notion of generalized net will be fixed, in forthcoming chapters we will focus on its categorification in order to characterize net executions and analysis.

6.1 Open nets, first definitions

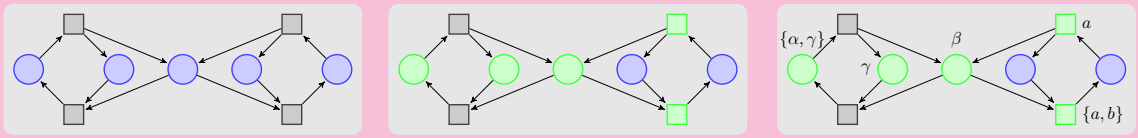
Intuitively we define an *open net* as a Petri net with two characteristics:

- Some of its places and transitions are *visible from the outside world* and can be accessed or used by "external actors". This is the feature that will allow us to intertwine the behavior of our net at runtime with the behavior of other nets running in the same environment (i.e. on a blockchain or a virtual machine);

- Nets can be composed together by means of gluing some places of the first net with some places of the second net. The places in a net that are "gluable" are identified by means of *ports*. This feature enables modularity while creating nets, and has the same role that libraries have in programming.

Before diving into mathematical details, let's try to make sense of this graphically. We represent places accessible from the outside world by highlighting them with another color (in this document we chose green, see Figure 6.1). What we mean by this is that in highlighted places tokens can be removed and added by some sort of "external agent" in a way that we will formally define later. Similarly, transitions highlighted in green are visible from the outside world, and can be fired by an external agent, again to be formally defined.

Let us think about this more in detail. If highlighted places and transitions are accessible from the outside world, how does an external agent specify with which transition or place to interact? We need a system to make this possible, that is, we need to give highlighted places and transitions some addresses that the outside world can use. To do this, we fix a couple of countable, *disjoint* sets, Σ and Ξ, that have to be considered as *sets of global labels*. In each net, highlighted places and transitions can be labeled with elements of Σ and Ξ, respectively, so a label has to be thought of as a *public address* for some highlighted place/transition in a given net. We demonstrate this in Figure 6.1c, with $\alpha, \beta, \gamma \in \Sigma$ and $a, b \in \Xi$. Notice that we are adopting the convention of denoting labels for highlighted places with greek letters and labels for highlighted transitions with latin ones. Moreover, we do allow different places/transitions in the same net to have the same label, and the same place/transition to have more than one label at the same time. This last feature can be thought of as equivalent to *aliasing*, where multiple addresses can refer to the same thing.



(a) A net.

(b) A net with some highlighted places and transitions (in green).

(c) An open net with labels.

Figure 6.1: A standard net (left) where some places and transitions are now open (right).

Now we focus on the second point, that is, we need to give a recipe that will allow us to compose nets. We want to do this by defining *ports* as we mentioned at the beginning of this section, and again before diving into mathematical definitions we reason graphically. The main idea is that net composition will work by means of gluing places together, and ports will denote which places have to be used in the gluing. The first distinction we have to make is between places that can be used to compose with a net on the left and places that can be glued to compose with a net on the right. Since ports are just a way to mark such places, this means that we have to further divide them into *left and right ports*. If we know that, say, a net has three left ports and two right ports, we can enumerate them: Left port 1,2,3 and right port 1,2, respectively. Graphically, we express this by marking places with a second layer of decorations, representing to which left/right ports a place is connected, and apply the following convention: Places decorated with a positive number are attached to a left port (i.e. a place decorated with "2" is connected to left port 2) while places highlighted with a negative number are attached to a

right port (i.e. a place decorated with "-2" is connected to right port 2). A net with ports is shown in Figure 6.2.

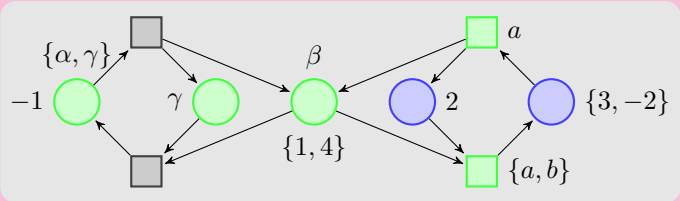


Figure 6.2: An open net with ports.

Looking at the figure we can already notice some things:

- Both highlighted and not highlighted places can be attached to a port;
- The same port cannot be connected to more than one place at a time, i.e. graphically there can't be two places both decorated with -1;
- On the other hand, the same place can be connected to more than one port;
- Ports have to be assigned in counting order, meaning that, for instance, there can't be a situation where the decorations $1, 2, 4$ appear but not 3.

Now that we made our intuition clear, let's try to enclose this information in a neat definition.

Definition 6.1.1 (Open Petri net)**.** *Fix two countable,disjoint sets* Σ *and* Ξ*, called* the global label sets for places and transitions, *respectively. An* Open Petri net *is a tuple*

$$N := (P_N, T_N, {}^\circ(-)_N, (-)^\circ_N, O_N, \lambda_{P_N}, V_N, \lambda_{T_N}, n_N, l_N, m_N, r_N)$$

Where:

- $(P_N, T_N, {}^\circ(-)_N, (-)^\circ_N)$ *is a Petri net;*
- O_N *is a subset of* P_N*, called* the set of open places of N*, representing highlighted places (the ones in green in Figure 6.1b);*
- $\lambda_{P_N} : O_N \to \mathscr{P}(\Sigma)\backslash\{\emptyset\}$ *is a function assigning finite, non-empty sets of labels to highlighted places (greek letters in Figure 6.1c);*
- V_N *is a subset of* T_N*, called* the set of visible transitions of N*, representing highlighted transitions (the ones in green in Figure 6.1b);*
- $\lambda_{T_N} : V_N \to \mathscr{P}(\Xi)\backslash\{\emptyset\}$ *is a function assigning finite, non-empty sets of labels to highlighted transitions (latin letters in Figure 6.1c);*
- n_N *and* m_N *are natural numbers, and represent the number of left and right ports of* N*, respectively;*
- $l_N : \{1, 2, \ldots, n_N\} \to P_N$ *is a function assigning every left port to a place (positive numbers in Figure 6.2). If* $n_N = 0$*, then* l_N *is set to be the empty function;*
- $r_N : \{1, 2, \ldots, m_N\} \to P_N$ *is a function assigning every right port to a place (negative numbers in Figure 6.2). If* $m_N = 0$*, then* r_N *is set to be the empty function.*

A marking $N^{\mathbb{N}}$ for the open net N is defined as usual, that is, as a multiset $P_N \to \mathbb{N}$. Sometimes we will adopt the concise notation:

$$N := (P_N, T_N, {}^\circ(-)_N, (-)^\circ_N, \lambda_{P_N}, \lambda_{T_N}, l_N, r_N)$$

To stand for the net $(P_N, T_N, {}^\circ(-)_N, (-)^\circ_N, O_N, \lambda_{P_N}, V_N, \lambda_{T_N}, n_N, l_N, m_N, r_N)$, where we implicitly define:

- $O_N := \operatorname{dom} \lambda_{P_N}$;
- $V_N := \operatorname{dom} \lambda_{T_N}$;
- $n_N := |\operatorname{dom} l_N|$;
- $m_N := |\operatorname{dom} r_N|$.

This is a rather cumbersome and heavy definition, but the intuition behind it should be easy to understand looking at the pictures above. We denote open places and visible transitions as subsets of places and transitions, and all the labeling is done via functions.

6.2 Static operations on open nets

Now we start playing with Definition 6.1.1 to see how we can compose nets together. The operations described in this section *will not* happen at runtime, that is, they are performed before the net is executed. This is because such operations are intrinsically *meta*, as the net cannot decide to merge with another net depending on its internal state. Enabling such a feature would make the analysis of reachability and other properties very difficult to perform (if not totally undecidable), resulting in a total loss of control over software behavior.

Such operations, then, are useful for the design process, ensuring modularity via the possibility of pre-defining nets that will have to be used later on as design components of bigger nets, in a very similar fashion to what happens with the usage of libraries in programming. The behavior of such compositions will not be just modular but *compositional*, which means that in the next section we will be able to establish a precise connection between composition operations and runtime behavior. This will ultimately be rendered possible by means of proofs linking the corresponding categorical semantics.

6.2.1 Sequential composition of open nets

The first way of composing nets that we take into account is by "gluing them together". The main idea is that open nets have ports, and if two nets have sets of ports that are somehow compatible, we should be able to link these ports together, obtaining a bigger net.

This composition, as we promised, works by means of place gluing. We call it *sequential composition*, and now proceed by making our ideas precise: Suppose that N_1, N_2 are nets. Then, n_{N_1}, n_{N_2} are the number of their left ports, respectively, while m_{N_1}, m_{N_2} are the number of their right ports, respectively. We can represent this fact using the notation:

$$\{1, 2, \ldots, n_{N_1}\} \xrightarrow{l_{N_1}} N_1 \xleftarrow{r_{N_1}} \{1, 2, \ldots, m_{N_1}\} \qquad \{1, 2, \ldots, n_{N_2}\} \xrightarrow{l_{N_2}} N_2 \xleftarrow{r_{N_2}} \{1, 2, \ldots, m_{N_2}\}$$

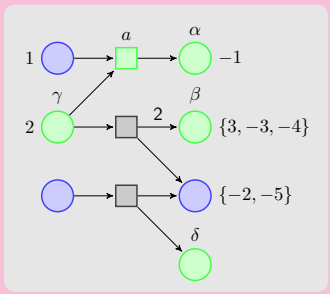
Let's suppose now that $m_{N_1} = n_{N_2}$, that is, N_1 has as many right ports as there are left ports of N_2. Then the notation above can be rewritten as:

$$\{1, 2, \ldots, n_{N_1}\} \xrightarrow{l_{N_1}} N_1 \xleftarrow{r_{N_1}} \{1, 2, \ldots, m_{N_1} = n_{N_2}\} \xrightarrow{l_{N_2}} N_2 \xleftarrow{r_{N_2}} \{1, 2, \ldots, m_{N_2}\}$$

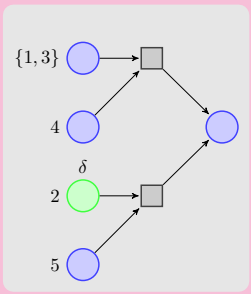
We see then that in such situation we can *connect each right port of N_1 with each left port of N_2 according to their number*, so right port 1 of N_1 is connected to left port 1 of N_2, and so on. From this we want to obtain a net, that we will denote as $N_1 \triangle N_2$, such that $n_{N_1 \triangle N_2} = n_{N_1}$ and $m_{N_1 \triangle N_2} = m_{N_2}$, that in our notation becomes

$$\{1, 2, \ldots, n_{N_1 \triangle N_2}\} \xrightarrow{l_{N_1 \triangle N_2}} N_1 \triangle N_2 \xleftarrow{r_{N_1 \triangle N_2}} \{1, 2, \ldots, m_{N_1 \triangle N_2}\}$$

What should $N_1 \triangle N_2$ look like? Well, the idea is to merge the places of N connected to a right port with the corresponding places of M connected to a left port of the same number. Moreover, if two open places are to be merged, their label sets should be unified, meaning that if for instance we are merging a place labeled with α with a place labeled by β, the resulting place should be labeled with $\{\alpha, \beta\}$. Transitions should be dealt with accordingly.



(a) First net to be composed, called N_1.



(b) Second net to be composed, called N_2.

Figure 6.3: Components of the sequential composition in Figure 6.4.

Example 6.2.1 (Sequential composition of nets, graphically)**.** As always, let's try to characterize sequential composition graphically first. Suppose you have the nets in Figures 6.3a and 6.3b. The right ports of N_1 match the left ones of N_2, since both range through the set $\{1, 2, 3, 4, 5\}$, hence sequential composition is possible. First, places decorated with a negative number in N_1 have to be identified with places decorated with a positive number in N_2: This means that the place in N_1 decorated with -1 has to be identified with the one in N_2 decorated with 1. This is in turn also decorated with 3, hence it has to be identified with the one decorated with $\{3, -3, -4\}$ in N_1, and hence, reasoning inductively, we end up identifying the following places: The ones decorated with -1 and $\{3, -3, -4\}$ in N_1, and the ones decorated with $\{1, 3\}$ and 4 in N_2.

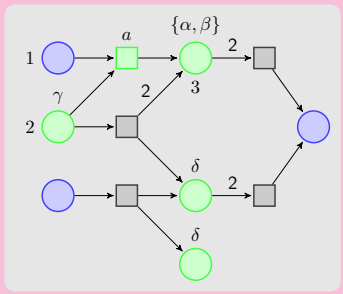


Figure 6.4: $N_1 \triangle N_2$, the sequential composition of nets in Figures 6.3a and 6.3b.

We moreover notice that the places decorated with -1 and $\{3, -3, -4\}$ in N_1 also have labels (α and β, respectively), and that the place decorated with $\{-3, -4\}$ is also connected to a left port (namely,

the port marked with 3). Since these places are glued together, the labels have to be unified, and the resulting place will have label $\{\alpha, \beta\}$, while still being connected to the left port 3.

In addition to this, the place in N_1 decorated with $\{-2, -5\}$ has to be identified with the places in N_2 decorated with 2 and 5, respectively. However, we notice that the place decorated with 2 in N_2 also has the label δ, which will also have to be preserved. Regarding transitions, if a transition has outgoing edges towards places p, q, and these are merged, the transition has an edge towards the merged place, with weight corresponding to the sum of the weights of the edges to p, q. The reasoning is similar for ingoing edges. The result of sequentially composing the nets in Figures 6.3a, 6.3b is shown in Figure 6.4.

Remark 6.2.2 (Sequential compositions are pushouts)**.** The reader familiar with categorical theories for networks will have noticed how place gluing works in a way that is very similar to the cospan formalism developed by Fong [10]. This is not a coincidence, and such techniques will be extensively used in modeling sequential net composition categorically.

Now that we developed a graphical intuition for composing open nets sequentially, let us make this precise mathematically.

Definition 6.2.3 (Sequential composition of open nets)**.** *Let*

$$N_1 := (P_{N_1}, T_{N_1}, {}^\circ(-)_{N_1}, (-)^\circ_{N_1}, O_{N_1}, \lambda_{P_{N_1}}, V_{N_1}, \lambda_{T_{N_1}}, n_{N_1}, l_{N_1}, m_{N_1}, r_{N_1})$$
$$N_2 := (P_{N_2}, T_{N_2}, {}^\circ(-)_{N_2}, (-)^\circ_{N_2}, O_{N_2}, \lambda_{P_{N_2}}, V_{N_2}, \lambda_{T_{N_2}}, n_{N_2}, l_{N_2}, m_{N_2}, r_{N_2})$$

be open Petri nets. The sequential composition of N_1, N_2*, denoted with $N_1 \triangle N_2$, is the open Petri net defined as follows:*

- $P_{N_1 \triangle N_2} := (P_{N_1} \sqcup P_{N_2})/\simeq$*, where $\simeq$ is the equivalence relation generated by the condition:*

  $$p \in P_{N_1} \simeq q \in P_{N_2} \text{ if } \exists x.(r_{N_1}(x) = p \wedge l_{N_2}(x) = q)$$

  Elements of $P_{N_1 \triangle N_2}$ are equivalence classes of $\simeq$, denoted as $[p]$. This means that the set of places is the disjoint union of the places of the component nets, modulo the identifications induced by setting as equals places that share ports;

- $T_{N_1 \triangle N_2} := T_{N_1} \sqcup T_{N_2}$*, hence the transitions are just the disjoint union of the transitions of the component nets;*

- *For each $t \in T_{N_1 \triangle N_2}$, ${}^\circ(t)_{N_1 \triangle N_2}$ is the multiset defined pointwise by:*

  $${}^\circ(t)_{N_1 \triangle N_2}([p]) := \begin{cases} \sum_{p' \in P_{N_1}, p' \in [p]} {}^\circ(t)_{N_1}(p') & \text{iff } t \in T_{N_1} \\ \sum_{p' \in P_{N_2}, p' \in [p]} {}^\circ(t)_{N_2}(p') & \text{iff } t \in T_{N_2} \end{cases}$$

  This means that the weight assigned to the edge from $[p]$ to t by ${}^\circ(t)_{N_1 \triangle N_2}$ equals the sum of the weights of the edges from each place in the equivalence class $[p]$ to t;

- *For each $t \in T_{N_1 \triangle N_2}$, $(t)^\circ_{N_1 \triangle N_2}$ is the multiset defined pointwise by:*

  $$(t)^\circ_{N_1 \triangle N_2}([p]) := \begin{cases} \sum_{p' \in P_{N_1}, p' \in [p]} (t)^\circ_{N_1}(p') & \text{iff } t \in T_{N_1} \\ \sum_{p' \in P_{N_2}, p' \in [p]} (t)^\circ_{N_2}(p') & \text{iff } t \in T_{N_2} \end{cases}$$

  The interpretation is analogous to the case above;

- $O_{N_1 \triangle N_2} := \{[p] \in P_{N_1 \triangle N_2} \mid \exists p' \in [p].((p' \in O_{N_1}) \vee (p' \in O_{N_2}))\}$*, meaning that a place is open if and only if in its equivalence class there is an open place of N_1 or an open place of N_2 (gluing with an open place results in an open place);*

- $\lambda_{P_{N_1 \triangle N_2}}$ *is defined pointwise as:*

$$\lambda_{P_{N_1 \triangle N_2}}([p]) := \left(\bigcup_{p' \in O_{N_1}, p' \in [p]} \lambda_{P_{N_1}}(p') \right) \cup \left(\bigcup_{p' \in O_{N_2}, p' \in [p]} \lambda_{P_{N_2}}(p') \right)$$

Hence labels are obtained by merging the label sets of places occurring in the gluing;

- $V_{N_1 \triangle N_2} := V_{N_1} \sqcup V_{N_2}$*, hence the set of visible transitions is just the disjoint union of the sets of visible transitions in the component nets;*

- $\lambda_{T_{N_1 \square N_2}}$ *is defined pointwise as:*

$$\lambda_{T_{N_1 \square N_2}}(t) := \begin{cases} \lambda_{T_{N_1}}(t) & \text{iff } t \in V_{N_1} \\ \lambda_{T_{N_2}}(t) & \text{iff } t \in V_{N_2} \end{cases}$$

So the transition labelings stay the same;

- $n_{N_1 \triangle N_2} := n_{N_1}$*, meaning that the number of left ports of the net is the same as in* N_1*;*

- $l_{N_1 \triangle N_2} := n_{N_1 \triangle N_2} \xrightarrow{id} n_{N_1} \xrightarrow{l_{N_1}} P_{N_1} \hookrightarrow P_{N_1} \sqcup P_{N_2} \twoheadrightarrow P_{N_1 \triangle N_2}$*. This means that the function assigning left interfaces to the net is morally equivalent to the one of* N_1*, composed with the obvious mapping of an element to its equivalence class in order to get the codomain right in the definition;*

- $m_{N_1 \triangle N_2} := m_{N_2}$*, meaning that the number of right ports of the net is the same as in* N_2*;*

- $r_{N_1 \triangle N_2} := m_{N_1 \triangle N_2} \xrightarrow{id} m_{N_2} \xrightarrow{r_{N_2}} P_{N_2} \hookrightarrow P_{N_1} \sqcup P_{N_2} \twoheadrightarrow P_{N_1 \triangle N_2}$*. The interpretation here is analogous to the one given for* $l_{N_1 \triangle N_2}$*.*

If N_1, N_2 *have markings* ${N_1}^{\mathbb{N}}, {N_2}^{\mathbb{N}}$*, respectively, the corresponding marking of* $N_1 \triangle N_2$ *is calculated taking their pointwise sum:*

$$(N_1 \triangle N_2)^{\mathbb{N}}([p]) := \sum_{p' \in P_{N_1}, p' \in [p]} {N_1}^{\mathbb{N}}(p') + \sum_{p' \in P_{N_2}, p' \in [p]} {N_2}^{\mathbb{N}}(p')$$

Again, this definition is very heavy, because there is a large amount of specification in an open net that has to be considered. Luckily enough, we are already able to prove some useful things, for instance,

Lemma 6.2.4 (Sequential composition of open nets is associative)**.** *Sequential composition of open Petri nets as in Definition 6.2.3 is associative, meaning that for open Petri nets* L, M, N *such that* $m_L = n_M$ *and* $m_M = n_N$*, it is*

$$(L \triangle M) \triangle N = L \triangle (M \triangle N)$$

6.2.2 Parallel composition of open nets

The second kind of composition we want to formalize should model minimal interaction. This amounts to merely "put nets next to each other", and is much easier to explain by resorting to pictures rather than to maths.

Example 6.2.5 (Parallel composition of nets, graphically)**.** Considering again the nets in Figures 6.3a and 6.3b, we define their *parallel composition*, denoted as $N_1 \square N_2$, as in Figure 6.5. So, what's going on here? Pretty much nothing, as we promised we are just putting the nets N_1 and N_2 next to each

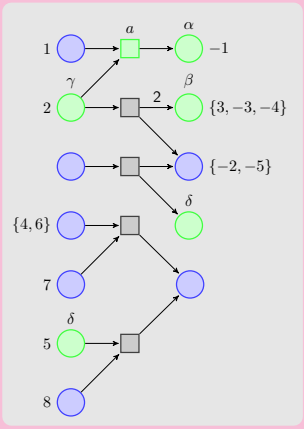


Figure 6.5: $N_1 \square N_2$, the parallel composition of nets in Figures 6.3a and 6.3b.

other. The only thing we need to do is to rename the left and right ports of N_2, to avoid overlappings with the ports of N_1: As we said, we are looking for minimal amount of interaction between the nets, but the ports $1, 2, 3$ and $1, 2, 3, 4, 5$ are already taken by N_1 on the left and on the right, respectively. This means that the left ports of N_2 will have to be renamed starting from 4, while its right ports should start from 6 (in our case this is not shown since N_2 has no right ports). As usual we now proceed with the mathematical formalization of this concept.

Definition 6.2.6 (Parallel composition of open nets)**.** *Let*

$$N_1 := (P_{N_1}, T_{N_1}, {}^\circ(-)_{N_1}, (-)^\circ_{N_1}, O_{N_1}, \lambda_{P_{N_1}}, V_{N_1}, \lambda_{T_{N_1}}, n_{N_1}, l_{N_1}, m_{N_1}, r_{N_1})$$
$$N_2 := (P_{N_2}, T_{N_2}, {}^\circ(-)_{N_2}, (-)^\circ_{N_2}, O_{N_2}, \lambda_{P_{N_2}}, V_{N_2}, \lambda_{T_{N_2}}, n_{N_2}, l_{N_2}, m_{N_2}, r_{N_2})$$

be open Petri nets. The parallel composition of N_1, N_2*, denoted with* $N_1 \square N_2$*, is the open Petri net defined as follows:*

- $P_{N_1 \square N_2} := P_{N_1} \sqcup P_{N_2}$*, hence the places are just the disjoint union of the places of the component nets;*
- $T_{N_1 \square N_2} := T_{N_1} \sqcup T_{N_2}$*, hence the transitions are just the disjoint union of the transitions of the component nets;*
- *For each* $t \in T_{N_1 \square N_2}$*,* ${}^\circ(t)_{N_1 \square N_2}$ *is the multiset defined by:*

$${}^\circ(t)_{N_1 \square N_2}(p) := \begin{cases} {}^\circ(t)_{N_1}(p) & \text{iff } t \in T_{N_1} \text{ and } p \in P_{N_1} \\ {}^\circ(t)_{N_2}(p) & \text{iff } t \in T_{N_2} \text{ and } p \in P_{N_2} \\ 0 & \text{Otherwise.} \end{cases}$$

- *For each* $t \in T_{N_1 \square N_2}$*,* $(t)^\circ_{N_1 \square N_2}$ *is the multiset defined pointwise by:*

$$(t)^\circ_{N_1 \square N_2}(p) := \begin{cases} (t)^\circ_{N_1}(p) & \text{iff } t \in T_{N_1} \text{ and } p \in P_{N_1} \\ (t)^\circ_{N_2}(p) & \text{iff } t \in T_{N_2} \text{ and } p \in P_{N_2} \\ 0 & \text{Otherwise.} \end{cases}$$

- $O_{N_1 \square N_2} := O_{N_1} \sqcup O_{N_2}$*, hence the set of open places is just the disjoint union of the sets of open places in the component nets;*

- $\lambda_{P_{N_1 \square N_2}}$ *is defined pointwise as:*

$$\lambda_{P_{N_1 \square N_2}}(p) := \begin{cases} \lambda_{P_{N_1}}(p) & \text{iff } p \in P_{N_1} \\ \lambda_{P_{N_2}}(p) & \text{iff } p \in P_{N_2} \end{cases}$$

  So the place labelings stay the same;

- $V_{N_1 \square N_2} := V_{N_1} \sqcup V_{N_2}$*, hence the set of visible transitions is just the disjoint union of the sets of visible transitions in the component nets;*

- $\lambda_{T_{N_1 \square N_2}}$ *is defined pointwise as:*

$$\lambda_{T_{N_1 \square N_2}}(t) := \begin{cases} \lambda_{T_{N_1}}(t) & \text{iff } t \in V_{N_1} \\ \lambda_{T_{N_2}}(t) & \text{iff } t \in V_{N_2} \end{cases}$$

  So the transition labelings stay the same;

- $n_{N_1 \square N_2} := n_{N_1} + n_{N_2}$*, meaning that the number of left ports of the net is the sum of the left ports of the components;*

- $l_{N_1 \square N_2}$ *is defined pointwise as:*

$$l_{N_1 \square N_2}(x) := \begin{cases} l_{N_1}(x) & \text{iff } 0 \leq x \leq n_{N_1} \\ l_{N_2}(x - n_{N_1}) & \text{iff } n_{N_1} < x \leq n_{N_1} + n_{N_2} \end{cases}$$

  This means that the function assigning left ports to the net behaves like the one of N_1 for the first n_{N_1} ports, and then as the one of N_2 until it reaches $n_{N_1} + n_{N_2}$;

- $m_{N_1 \square N_2} := m_{N_2} + m_{N_2}$*, meaning that the number of right ports of the net is the sum of the right ports of the components;*

- $r_{N_1 \square N_2}$ *is defined pointwise as:*

$$r_{N_1 \square N_2}(x) := \begin{cases} r_{N_1}(x) & \text{iff } 0 \leq x \leq m_{N_1} \\ r_{N_2}(x - m_{N_1}) & \text{iff } m_{N_1} < x \leq m_{N_1} + m_{N_2} \end{cases}$$

  The interpretation here is analogous to the one given for $l_{N_1 \square N_2}$.

To ${N_1}^{\mathbb{N}}, {N_2}^{\mathbb{N}}$ markings of N_1, N_2, respectively, we associate a corresponding marking of $N_1 \square N_2$ defined pointwise as:

$$(N_1 \square N_2)^{\mathbb{N}}(p) := \begin{cases} {N_1}^{\mathbb{N}}(p) & \text{iff } p \in P_{N_1} \\ {N_2}^{\mathbb{N}}(p) & \text{iff } p \in P_{N_2} \end{cases}$$

For parallel composition of open Petri nets, we can prove a result very similar to the one we proved in Lemma 6.2.4. These results will be useful for drafting categorical semantics for open nets in the subsequent chapters:

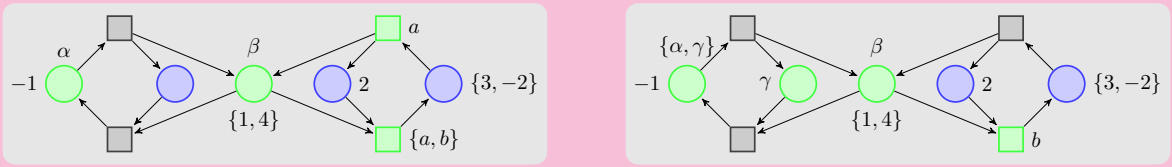
Lemma 6.2.7 (Parallel composition of open nets is associative)**.** *Parallel composition of open Petri nets as in Definition 6.2.6 is associative, meaning that for open Petri nets L, M, N it is*

$$(L \square M) \square N = L \square (M \square N)$$

6.2.3 Hiding of open Petri nets

We now introduce the last operation we can perform on open nets, *hiding*. The idea of hiding is very simple: We just make some open places (respectively, visible transitions) not open (respectively, visible) anymore. The idea of hiding is akin to *revoking an address* in a net. Suppose, for instance, that in a given net there are places that have the label α. Revoking α should mean that this label is stripped from all the places which have it, and if a place has no other label, it shouldn't be considered open anymore (this is consistent with the idea that highlighted places and transitions are considered reachable from the outside world, and hence need to have at least one address to be reached).

Similarly, revoking a label a on transitions should have the effect of stripping a from each transition label and regarding a transition that doesn't have any address but a as no longer visible. We note quite soon that revoking all the labels in Σ has the effect of making all the places not open anymore, and revoking all the labels in Ξ makes each transition not-visible. Moreover, given a net, we deem it sensible to be able to revoke multiple addresses at the same time. The behavior of hiding places and transitions is depicted in Figure 6.6.



(a) Revoking the label γ.

(b) Revoking the label a.

Figure 6.6: Hiding places and transitions of the net in Figure 6.2.

The matematical formalization of hiding is, luckily, way less involved compared to the ones defining sequential and parallel composition. It is as follows:

Definition 6.2.8. *(Hiding operators for open nets) For each $X \subseteq \Sigma$ we define the* place-hiding operator $\diamond_X^{Pl}(-)$*, acting on a net N as follows: $\diamond_X^{Pl}(N)$ is the net such that $P_{\diamond_X^{Pl}(N)}$, $T_{\diamond_X^{Pl}(N)}$, ${}^\circ(-)_{\diamond_X^{Pl}(N)}$, $(-)^\circ_{\diamond_X^{Pl}(N)}$, $V_{\diamond_X^{Pl}(N)}$, $\lambda_{T_{\diamond_X^{Pl}(N)}}$, $n_{\diamond_X^{Pl}(N)}$, $l_{\diamond_X^{Pl}(N)}$, $m_{\diamond_X^{Pl}(N)}$, $r_{\diamond_X^{Pl}(N)}$ and markings of N are all equal to their respective counterpart in N, while:*

- $O_{\diamond_X^{Pl}(N)} := O_N \setminus \{p \in O_N \mid \lambda_{P_N}(p) \subseteq X\}$*, meaning that we eliminate from O_N the places whose labels are all contained in the set X;*
- $\lambda_{P_{\diamond_X^{Pl}(N)}} := O_{\diamond_X^{Pl}(N)} \hookrightarrow O_N \xrightarrow{\lambda_{P_N}} \mathscr{P}(\Sigma)\setminus\{\emptyset\} \xrightarrow{\setminus X} \mathscr{P}(\Sigma)\setminus\{\emptyset\}$*, where $\setminus X$ is the function that substracts the set X from each element of $\mathscr{P}(\Sigma)\setminus\{\emptyset\}$. Explicitly, $\lambda_{P_{\diamond_X^{Pl}(N)}}$ is defined pointwise as $\lambda_{P_{\diamond_X^{Pl}(N)}}(p) := \lambda_{P_N}(p)\setminus X$.*

Similarly, for each $Y \subseteq \Xi$ we define the transition-hiding operator $\diamond_Y^{Tr}(-)$*, acting on a net N as follows: $\diamond_Y^{Tr}(N)$ is the net such that $P_{\diamond_Y^{Tr}(N)}$, $T_{\diamond_Y^{Tr}(N)}$, ${}^\circ(-)_{\diamond_Y^{Tr}(N)}$, $(-)^\circ_{\diamond_Y^{Tr}(N)}$, $O_{\diamond_Y^{Tr}(N)}$, $\lambda_{P_{\diamond_Y^{Tr}(N)}}$, $n_{\diamond_Y^{Tr}(N)}$, $l_{\diamond_Y^{Tr}(N)}$, $m_{\diamond_Y^{Tr}(N)}$, $r_{\diamond_Y^{Tr}(N)}$ and markings of N are all equal to their respective counterpart in N, while:*

- $V_{\diamond_Y^{Tr}(N)} := V_N \setminus \{t \in V_N \mid \lambda_{T_N}(t) \subseteq Y\}$*, meaning that we eliminate from V_N the transitions whose labels are all contained in the set Y;*
- $\lambda_{T_{\diamond_Y^{Tr}(N)}} := V_{\diamond_Y^{Tr}(N)} \hookrightarrow V_N \xrightarrow{\lambda_{T_N}} \mathscr{P}(\Xi)\setminus\{\emptyset\} \xrightarrow{\setminus Y} \mathscr{P}(\Xi)\setminus\{\emptyset\}$*, where $\setminus Y$ is the function that substracts the set Y from each element of $\mathscr{P}(\Xi)\setminus\{\emptyset\}$. Explicitly, $\lambda_{T_{\diamond_Y^{Tr}(N)}}$ is defined pointwise as $\lambda_{T_{\diamond_Y^{Tr}(N)}}(t) := \lambda_{T_N}(t)\setminus Y$.*

Noticing that both transition and place-hiding operators are defined in terms of set subtraction, it is very easy to prove the following lemma.

Lemma 6.2.9 (Properties of hiding operators)**.** *Given a net N and subsets $X, Y \subseteq \Sigma$, $X', Y' \subseteq \Xi$, we have:*

$$\diamond^{Pl}_X(\diamond^{Tr}_{X'}(N)) = \diamond^{Tr}_{X'}(\diamond^{Pl}_X(N))$$

$$\diamond^{Pl}_X(\diamond^{Pl}_Y(N)) = \diamond^{Pl}_Y(\diamond^{Pl}_X(N)) \qquad \diamond^{Tr}_{X'}(\diamond^{Tr}_{Y'}(N)) = \diamond^{Tr}_{Y'}(\diamond^{Tr}_{X'}(N))$$

$$\diamond^{Pl}_X(\diamond^{Pl}_X(N)) = \diamond^{Pl}_X(N) \qquad \diamond^{Tr}_{X'}(\diamond^{Tr}_{X'}(N)) = \diamond^{Tr}_{X'}(N)$$

$$\diamond^{Pl}_X(\diamond^{Pl}_Y(N)) = \diamond^{Pl}_{X\cup Y}(N) \qquad \diamond^{Tr}_{X'}(\diamond^{Tr}_{Y'}(N)) = \diamond^{Tr}_{X'\cup Y'}(N)$$

$$\alpha \notin \bigcup \lambda_{P_N}(O_N) \Rightarrow \diamond^{Pl}_X(N) = \diamond^{Pl}_{X\setminus\{\alpha\}}(N) \qquad a \notin \bigcup \lambda_{T_N}(V_N) \Rightarrow \diamond^{Tr}_{X'}(N) = \diamond^{Tr}_{X'\setminus\{a\}}(N)$$

$$\diamond^{Pl}_\emptyset(N) = N \qquad \diamond^{Tr}_\emptyset(N) = N$$

$$O_{\diamond^{Pl}_\Sigma(N)} = \emptyset \qquad V_{\diamond^{Tr}_\Xi(N)} = \emptyset$$

In particular, motivated by the fact that transition and place-hiding operators commute, we will write $\diamond_{X,X'}(N)$ to stand for $\diamond^{Pl}_X(\diamond^{Tr}_{X'}(N))$.

Remark 6.2.10 (Why only hiding?)**.** In Defintion 6.2.8 we formalized the idea of revoking some addresses from a net. A perfectly legitimate question would be: Why not make this procedure more precise, by e.g. targeting a specific place and transition of the net? And why not provide an *inverse* procedure, where we can actually assign new addresses to places and transitions?

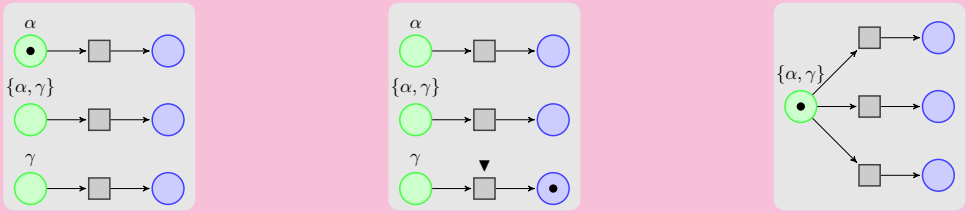
In both cases, the problem is that such operations would be *local*, meaning that "targeting a specific place or transition" is an operation which depends on the net we are operating on. The hiding operators as we defined them, instead, are *global*, meaning that they only depend on a subset of the global labels (Σ and Ξ, respectively). This allows us the define them for all the possible nets at once, exactly as we did for sequential and parallel composition.

6.3 Runtime operations on open nets

Now we finally get to the interesting part, namely how nets behave at runtime. To understand this, we have to understand what labels mean for places and transitions. They are addresses, yes, but their interpretations are quite different:

- Places sharing a label have to be thought of as memory pools: If a token is in a place labeled with α, all the other places in our environment having α among their labels can use that token. This reasoning quickly becomes inductive: Imagine three places: p_1 with label α, p_2 with label $\{\alpha, \gamma\}$, and p_3 with label γ. Putting a token in p_1 makes it accessible from p_2, which in turn makes it accessible in p_3, even if p_1 and p_3 do not share a label. This is because in p_2 the labels α, γ are set to be aliases of each other, making them effectively indistinguishable. Such behavior is illustrated in Figures 6.7a and 6.7b. The right way to interpret this is then "every time a place has labels, those labels have to be interpreted as aliases referring to the same memory pool", as in Figure 6.7c.

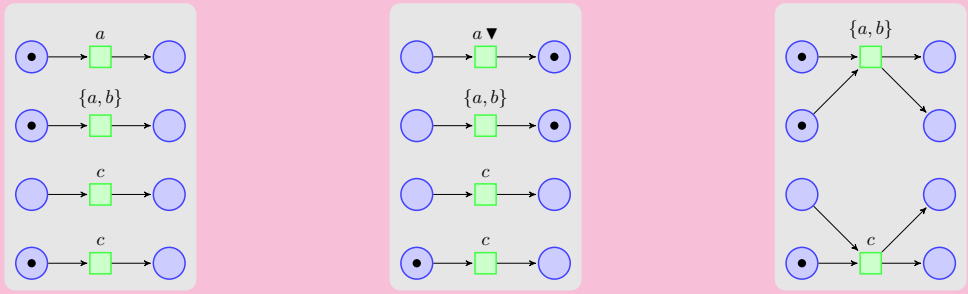
- Transition labels are like addresses for broadcastings the event of transition firings. This means that every time a transition t having label a wants to fire, this information is broadcast over the environment, and all the other transitions having a as a label are forced to fire as well. What happens if, say, one of those transitions cannot fire or is deadlocked? In that case the event cannot happen, meaning that *transitions sharing a label are forced to fire synchronously*. These two behaviors are illustrated in Figures 6.8a and 6.8b, where the transitions labeled with a fire



(a) A net. (b) Firing. (c) Runtime semantics for the net.

Figure 6.7: Runtime semantics for labeled places.

synchronously while the ones marked with c are deadlocked. The net in Figure 6.8a then behaves as the one in Figure 6.8c. The recursive thinking showed in the previous point also holds in the case of nets. If a net has multiple labels, then those labels have to be interpreted as aliases carrying the same information across the environment.



(a) A net. (b) Firing synchronously. (c) Runtime semantics for the net.

Figure 6.8: Runtime semantics for labeled transitions.

It follows that given an open Petri net, to understand its behavior at runtime we need to identify some places and transitions, according to their labels. We are going to define such an operation in the next subsection.

6.3.1 Quotienting

The behavior of a net at runtime can be completely characterized by introducing just two new operators, called *quotienting*. These operators act via controlled, parametrized identifications of places and transitions.

Definition 6.3.1 (Place-quotienting of open nets)**.** *Let*

$$N := (P_N, T_N, {}^\circ(-)_N, (-)^\circ_N, O_N, \lambda_{P_N}, V_N, \lambda_{T_N}, n_N, l_N, m_N, r_N)$$

Be an open Petri net. For each $X \subseteq \Sigma$ we define the place-quotienting of N by X*, denoted with $\bigstar^{Pl}_X(N)$, as the open Petri net defined as follows:*

- $P_{\bigstar^{Pl}_X(N)} := P_N / \simeq$*, where $\simeq$ is the equivalence relation generated by the condition:*

  $$p \simeq q \text{ if } p, q \in O_N \text{ and } \lambda_{P_N}(p) \cap \lambda_{P_N}(q) \cap X \neq \emptyset$$

  Elements of $P_{\bigstar^{Pl}_X(N)}$ are equivalence classes of $\simeq$, denoted as $[p]$. This means that the set of places is obtained by recursively identifying all the places sharing a label in X;

- $T_{\bigstar_X^{Pl}(N)} := T_N$;
- *For each* $t \in T_{\bigstar_X^{Pl}(N)}$, ${}^\circ(t)_{\bigstar_X^{Pl}(N)}$ *is the multiset defined pointwise by:*

$$ {}^\circ(t)_{\bigstar_X^{Pl}(N)}([p]) := \sum_{p' \in [p]} {}^\circ(t)_N(p') $$

  This means that the weight assigned to the edge from $[p]$ *to* t *by* ${}^\circ(t)_{\bigstar_X^{Pl}(N)}$ *equals the sum of the weights of the edges from each place in* $[p]$ *to the transition* t;

- *For each* $t \in T_{\bigstar_X^{Pl}(N)}$, $(t)^\circ_{\bigstar_X^{Pl}(N)}$ *is the multiset defined pointwise by:*

$$ (t)^\circ_{\bigstar_X^{Pl}(N)}([p]) := \sum_{p' \in [p]} (t)^\circ_N(p') $$

  The interpretation is analogous to the case above;

- $O_{\bigstar_X^{Pl}(N)} := \left\{ [p] \in P_{\bigstar_X^{Pl}(N)} \;\middle|\; \exists p' \in [p].(p' \in O_N) \right\}$, *meaning that a place is open if and only if in its equivalence class there is an open place of* N;
- $\lambda_{P_{\bigstar_X^{Pl}(N)}}$ *is defined pointwise as:*

$$ \lambda_{P_{\bigstar_X^{Pl}(N)}}([p]) := \bigcup_{p' \in [p]} \lambda_{P_N}(p) $$

  Hence labels are obtained by merging the label sets of places in the equivalence class;

- $V_{\bigstar_X^{Pl}(N)} := V_N$ *and* $\lambda_{T_{\bigstar_X^{Pl}(N)}} := \lambda_{T_N}$;
- $n_{\bigstar_X^{Pl}(N)} := n_N$ *and* $m_{\bigstar_X^{Pl}(N)} := m_N$, *meaning that the number of left/right ports of the net is the same as* N;
- $l_{\bigstar_X^{Pl}(N)} := n_{\bigstar_X^{Pl}(N)} \xrightarrow{id} n_N \xrightarrow{l_N} P_N \twoheadrightarrow P_{\bigstar_X^{Pl}(N)}$. *This means that the function assigning left interfaces to the net is morally equivalent to the one of* N, *composed with the obvious mapping of an element in its equivalence class to get the right codomain in the definition;*
- $r_{\bigstar_X^{Pl}(N)} := m_{\bigstar_X^{Pl}(N)} \xrightarrow{id} m_N \xrightarrow{r_N} P_N \twoheadrightarrow P_{\bigstar_X^{Pl}(N)}$. *The interpretation here is analogous to the case above.*

If N *has marking* $N^{\mathbb{N}}$, *the corresponding marking of* $\bigstar_X^{Pl}(N)$ *is calculated pointwise:*

$$ \bigstar_X^{Pl}(N)^{\mathbb{N}}([p]) := \sum_{p' \in [p]} N^{\mathbb{N}}(p') $$

Place-quotienting has the effect of merging all the places in a net sharing labels contained in a set specified by the user (X in Definition 6.3.1). We now define a similar operator, acting on transitions.

Definition 6.3.2 (Transition-quotienting of open nets)**.** *Let*

$$ N := (P_N, T_N, {}^\circ(-)_N, (-)^\circ_N, O_N, \lambda_{P_N}, V_N, \lambda_{T_N}, n_N, l_N, m_N, r_N) $$

Be an open Petri net. For each $Y \subseteq \Xi$ *we define the* transition-quotienting of N, *denoted with* $\bigstar_X^{Tr}(N)$, *as the open Petri net defined as follows:*

- $P_{\bigstar_Y^{Tr}(N)} := P_N$;
- *We define* $T_{\bigstar_Y^{Tr}(N)}$ *as follows:*

$$T_{\bigstar_Y^{Tr}(N)} := \bigcup_{[t] \in T_N/\simeq} \left(\prod_{t' \in [t]} t' \right)$$

where $\simeq$ *is the equivalence relation generated by the condition:*

$$t \simeq u \text{ if } t, u \in V_N \text{ and } \lambda_{T_N}(t) \cap \lambda_{T_N}(u) \cap Y \neq \emptyset$$

This definition may seem very cumbersome, but it is intuitively simple: First, we identify all the transitions that share a label in Y*, and partition the set* T_N *as such. What we got now is a bunch of equivalence classes, each a set of transitions of* N*. We want to regard transitions in each equivalence class as firing synchronously, and we want to represent this by replacing them with a tuple. If, for instance,* $[t]$ *consists of the transitions* t, t', t''*, we want to substitute these transitions in* N *with a unique transition* $\langle t, t', t'' \rangle$*. We do this by forming a tuple with all the transitions in a given equivalence class (hence the product sign) and we need to do this for each equivalence class (hence the union sign). Notice that if a transition is not synchronized with anything, its equivalence class will consist of only one element, and forming the tuple will trivially return the element itself.*

- *For each* $t \in T_{\bigstar_Y^{Tr}(N)}$, ${}^\circ(t)_{\bigstar_Y^{Tr}(N)}$ *is the multiset defined pointwise by:*

$${}^\circ(t)_{\bigstar_Y^{Tr}(N)}(p) := \sum_{\{t' \in T_N \mid \exists i.(\pi_i(t) = t')\}} {}^\circ(t')_N(p)$$

This means that the weight assigned to the edge from p *to* t *by* ${}^\circ(t)_{\bigstar_Y^{Tr}(N)}$ *equals the sum of the weights of the edges from* p *to each component of the transition tuple* t*;*

- *For each* $t \in T_{\bigstar_Y^{Tr}(N)}$, $(t)^\circ_{\bigstar_Y^{Tr}(N)}$ *is the multiset defined pointwise by:*

$$(t)^\circ_{\bigstar_Y^{Tr}(N)}(p) := \sum_{\{t' \in T_N \mid \exists i.(\pi_i(t) = t')\}} (t')^\circ_N(p)$$

The interpretation is analogous to the case above;

- $O_{\bigstar_Y^{Tr}(N)} := O_N$ *and* $\lambda_{P_{\bigstar_Y^{Tr}(N)}} := \lambda_{P_N}$*;*
- $V_{\bigstar_Y^{Tr}(N)} := \left\{ t \in T_{\bigstar_Y^{Tr}(N)} \;\middle|\; \exists i.(\pi_i(t) \in V_N) \right\}$;
- $\lambda_{T_{\bigstar_Y^{Tr}(N)}}$ *is defined pointwise as:*

$$\lambda_{T_{\bigstar_Y^{Tr}(N)}}(t) := \bigcup_{\{t' \in V_N \mid \exists i.(\pi_i(t) = t')\}} \lambda_{T_N}(t')$$

- $n_{\bigstar_Y^{Tr}(N)} := n_N$, $l_{\bigstar_Y^{Tr}(N)} := l_N$, $m_{\bigstar_Y^{Tr}(N)} := m_N$ *and* $r_{\bigstar_Y^{Tr}(N)} := r_N$.

The marking of $\bigstar_Y^{Tr}(N)$ *is taken to be equal to the marking of* N*.*

Unsurprisingly, quotienting has some nice properties that are somehow similar to the ones of the hiding operator (Definition 6.2.8). We have the following:

Lemma 6.3.3 (Properties of quotienting operators)**.** *Given a net N and subsets $X, Y \subseteq \Sigma$, $X', Y' \subseteq \Xi$, we have:*

$$\bigstar^{Pl}_{X}(\bigstar^{Tr}_{X'}(N)) = \bigstar^{Tr}_{X'}(\bigstar^{Pl}_{X}(N))$$

$$\begin{aligned}
\bigstar^{Pl}_{X}(\bigstar^{Pl}_{Y}(N)) &= \bigstar^{Pl}_{Y}(\bigstar^{Pl}_{X}(N)) & \bigstar^{Tr}_{X'}(\bigstar^{Tr}_{Y'}(N)) &= \bigstar^{Tr}_{Y'}(\bigstar^{Tr}_{X'}(N)) \\
\bigstar^{Pl}_{X}(\bigstar^{Pl}_{X}(N)) &= \bigstar^{Pl}_{X}(N) & \bigstar^{Tr}_{X'}(\bigstar^{Tr}_{X'}(N)) &= \bigstar^{Tr}_{X'}(N) \\
\bigstar^{Pl}_{X}(\bigstar^{Pl}_{Y}(N)) &= \bigstar^{Pl}_{X\cup Y}(N) & \bigstar^{Tr}_{X'}(\bigstar^{Tr}_{Y'}(N)) &= \bigstar^{Tr}_{X'\cup Y'}(N) \\
\alpha \notin \bigcup \lambda_{P_N}(O_N) \Rightarrow \bigstar^{Pl}_{X}(N) &= \bigstar^{Pl}_{X\setminus\{\alpha\}}(N) & a \notin \bigcup \lambda_{T_N}(V_N) \Rightarrow \bigstar^{Tr}_{X'}(N) &= \bigstar^{Tr}_{X'\setminus\{a\}}(N) \\
\bigstar^{Pl}_{\emptyset}(N) &= N & \bigstar^{Tr}_{\emptyset}(N) &= N
\end{aligned}$$

In particular, motivated by the fact that transition and place quotienting operators commute, we will write $\bigstar_{X,X'}(N)$ to stand for $\bigstar^{Pl}_{X}(\bigstar^{Tr}_{X'}(N))$. Unpacking this definition we can moreover prove:

$$\bigstar_{X,Y}(\bigstar_{X',Y'}(N)) = \bigstar_{X',Y}(\bigstar_{X,Y'}(N)) = \bigstar_{X',Y'}(\bigstar_{X,Y}(N)) = \bigstar_{X,Y'}(\bigstar_{X',Y}(N))$$

$$\bigstar_{X,Y}(\bigstar_{X',Y'}(N)) = \bigstar_{X\cup X',Y\cup Y'}(N)$$

$$\alpha \notin \bigcup \lambda_{P_N}(O_N) \wedge a \notin \bigcup \lambda_{T_N}(V_N) \Rightarrow \bigstar_{X,Y}(N) = \bigstar_{X\setminus\{\alpha\},Y\setminus\{a\}}(N)$$

$$\bigstar_{\emptyset,\emptyset}(N) = N$$

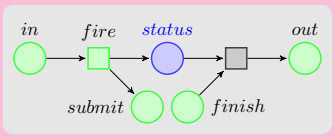
$$\bigstar_{X,Y}(N) = \bigstar_{X,\emptyset}(\bigstar_{\emptyset,Y}(N))$$

$$\bigstar_{X,\emptyset}(N) = \bigstar^{Pl}_{X}(N) \qquad \bigstar_{\emptyset,Y}(N) = \bigstar^{Tr}_{Y}(N)$$

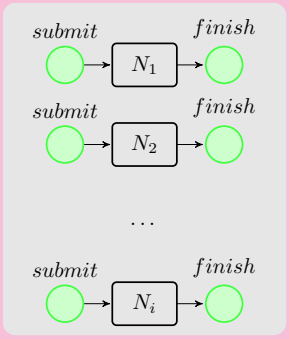
6.3.2 Examples

Now that we went through all the technicalities of the behavior of nets at runtime, we provide some examples of how this can be useful. Essentially, open places and visible transitions allow a net to communicate with other nets living in the same environment, and we can study the behavior of these communications using the quotienting operator.

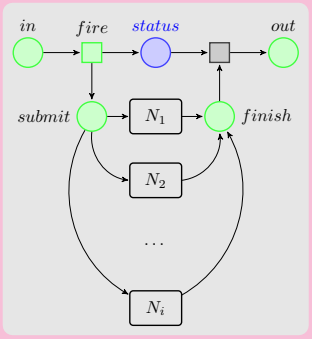
Example 6.3.4 (Delegated asynchronous computation)**.** Imagine you have a "master net" N, as in Figure 6.9a, and a bunch of "slave nets" $N_1, \ldots, N_i$, as in Figure 6.9b. The open place marked as *in* in Figure 6.9a stands for a memory pool where a user can deposit data to be processed, while the *out* place is where the user can collect the data after it has been computed.



(a) The master net N.



(b) Slave nets $N_1, \ldots, N_i$.



(c) Runtime semantics.

Figure 6.9: Delegated asynchronous computation.

The master net will delegate the actual computation to the slave nets in Figure 6.9b, that will process such information concurrently. When a firing message is broadcast on the address *fire*, the corresponding

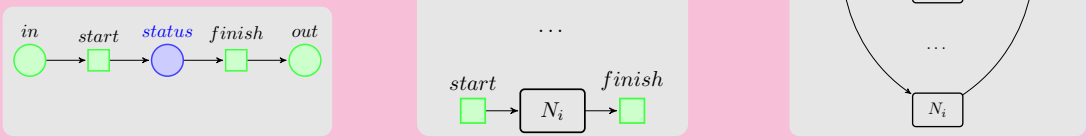
transition in the master net fires, effectively starting the computation. It produces a token in another memory pool, labeled as "submit", and a token in a (non-visible) place, named *status* (note that this is *not* a label, and we stressed this by coloring it blue and named it only for clarity of exposition). The *status* place keeps track, in the master net, of how many delegated computations are running. The slave nets $N_1, \ldots, N_n$ are all connected to the *start* memory pool, and have to act concurrently to use the token deposited there by the master net. Clearly only one of these slave nets will be able to claim the resource, and when the computation is finished, it will output it to the *finish* memory pool. At this point the rightmost transition of the master net will be enabled, and its firing will denote that the computation is finished. The result is output on the *out* memory pool, ready for the user to collect. The overall behavior of this net can be analyzed by applying the quotient operator to the parallel product of $N, N_1, \ldots, N_i$, that is, by looking at:

$$\bigstar_{\Sigma,\Xi}(N \square N_1 \square \ldots \square N_i) \tag{6.3.1}$$

The resulting net is displayed in Figure 6.9c. Notice that the net representing the runtime behavior of N along with $N_1, \ldots, N_i$ can be completely determined by Equation 6.3.1 *only if* the labels appearing in $N, N_1, \ldots N_i$ are not being used by other nets as well. This not being the case, another net M not represented in the picture could, say, "steal" the token from the *submit* memory pool before any of the slave nets claims it, causing unintended behavior. During implementation, we can ensure that such eventuality doesn't happen by making addresses reservable or private, i.e. by giving a set of rules determining which nets have access to a given memory pool/broadcast address. To model the fact that the environment cannot influence our nets in unexpected ways, viz. that no nets aside of $N, N_1, \ldots, N_i$ have to be involved in our analysis, we can study the properties of the net

$$\diamond_{\Sigma,\Xi}(\bigstar_{\Sigma,\Xi}(N \square N_1 \square \ldots \square N_i))$$

Together with an initial marking consisting of tokens only in the place originally marked as *in*. These last considerations will become considerably clearer once the relationships between design and runtime semantics will be made explicit.



(a) The master net N. (b) Slave nets $N_1, \ldots, N_i$. (c) Runtime semantics.

Figure 6.10: Delegated synchronous computation.

Example 6.3.5 (Delegated synchronous computation)**.** This example is very similar in flavor to Example 6.3.4, with the difference that this time we want the slave nets to run cooperatively instead of concurrently. We obtain this just by turning some places into transitions, as in Figures 6.10a and 6.10b. This time, the transition in the master net N labeled as *start* is syncronized with a corresponding transition in all the slave nets. Similarly, the *finish* label in the master net will not be able to fire until all the corresponding transitions in the slave nets are enabled. The runtime behavior of the nets is

displayed in Figure 6.10c, and the same caveat of Example 6.3.4 applies: If there are other nets with transitions sharing the labels *start* or *finish* the net may misbehave, for instance we could have a net with a transition labeled *start* that is permanently deadlocked, making the whole process described here deadlocked as well. This is again fixed during implementation by making addresses private or usable only by specified users/nets.

Notice that, in this case, the *in* place in the master net only has the function of "reserving" a computation. In fact, the master net doesn't pass any data to the slave nets, it merely orchestrates their executions so that they start together. This is not surprising if one thinks about nets as a resource theory. To achieve a "true" synchronous, delegated computation one should do the following:

- The token representing the data to be computed, left by the user in the place labeled as *in*, should be processed by a transition that copies it, and sends the copies to a memory pool where all the slave nets can fetch them. Alternatively, the slave nets should have direct access to the memory pool *in*, provided that every time they take a token from there they also "put it back";
- The token left by the user in *in* should be deleted when the transition labeled as *finish* fires;
- The tokens representing output data from the slave nets should be unified by the transition labeled as *finish*: The token in *out* would then represent the result of unifying all the parallel computations performed by the slave nets.

6.4 Relationship between static and runtime operations

Now we present what is probably the most important result of the whole Chapter. Suppose you have a number of nets which are to be used as components for designing a bigger net. This can be done by using the sequential ($\triangle$), parallel ($\square$) and hiding ($\diamond$) operators. At this point, the final net will most likely have some open places and visible transitions, and some of the labels will most likely appear more than once. We would like, at this point, to analyze the behavior of the Net at runtime, maybe using some of the techniques already developed in the Petri net literature. We face, alas, two problems:

- Our nets are a generalization of the ones already studied in the literature, and this may mean that some (if not all) of the optimization techniques to analyze reachability developed in the last decade may be underperforming or not apply at all in our case. This is because tokens can literally "teleport" from an open place to another sharing the same labels and, even worse, transitions that share a label are forced to fire together but considered as separate by standard software;
- Even if we were to ignore the previous point, or are able to develop a tool to precisely analyze our flavor of nets, we cannot rule out the possibility of side effects. For instance, how can we tell our software that we are sure that the labels we are using are reserved, and no other nets except the ones we specify will be able to share places and synchronize with our transitions?

Luckily enough, we are able to take open nets and "fall back" to a classic net by means of the runtime and hiding operators. In fact, the behavior of a net N at runtime, when no side effects are possible, is just:

$$\diamond_{\Sigma,\Xi}\left(\bigstar_{\Sigma,\Xi}(N)\right) \tag{6.4.1}$$

Notice how we used precisely this kind of equation in Example 6.3.4. The quotienting operator is identifying all the places sharing tokens and transitions firing synchronously (hence solving our first concern) while the hiding operator is giving us the behavior of the net in the absence of side effects (by closing all the places). The whole point of this is that if we know that side effects on the labels are not possible (e.g. because the labels are reserved) then our net N, at runtime, will behave *exactly* as the net in Equation 6.4.1. The operators $\diamond_{\Sigma,\Xi}(-)$ and $\bigstar_{\Sigma,\Xi}(-)$ are so important that we will denote them just as $\diamond(-)$ and $\bigstar(-)$, respectively.

The problem with this, on the other hand, is that the $\bigstar$ operator works by means of quotienting via equivalence relations, that can be quite a computationally expensive process if the net is big. What we want to do now is figure out if there is a clever way to compute it. The answer is yes, and it is not surprising: We compute it little by little, by letting it "slide" inside other operators. The following lemma will make such intuition precise.

Lemma 6.4.1. *Let N_1, N_2 be open Petri nets. Let X_1, X_2 the set of place labels that appear in N_1, N_2, respectively. With this we mean that, for example, if no place in N_1 has label α, then $\alpha \notin X_1$. Similarly, let Y_1, Y_2 the set of transition labels that appear in N_1, N_2, respectively. Let furthermore be $X, X' \subseteq \Sigma$ and $Y, Y' \subseteq \Xi$. We then have:*

$$\diamond_{X,Y}(N_i) = \diamond_{X\cap X_i, Y\cap Y_i}(N_i)$$
$$\diamond_{X,Y}(N_1 \triangle N_2) = \diamond_{X,Y}(N_1) \triangle \diamond_{X,Y}(N_2)$$
$$\diamond_{X,Y}(N_1 \square N_2) = \diamond_{X,Y}(N_1) \square \diamond_{X,Y}(N_2)$$
$$\diamond_{X,Y}(\bigstar_{X',Y'}(N_i)) = \diamond_{X\cap X_i, Y\cap Y_i}(\bigstar_{X'\cap X_i, Y'\cap Y_i}(N_i))$$
$$\bigstar_{X,Y}(N_i) = \bigstar_{X\cap X_i, Y\cap Y_i}(N_i)$$
$$\bigstar_{X,Y}(N_1 \triangle N_2) = \bigstar_{X\cap X_1\cap X_2, Y\cap Y_1\cap Y_2}(\bigstar_{X\cap X_1, Y\cap Y_1}(N_1) \triangle \bigstar_{X\cap X_2, Y\cap Y_2}(N_2))$$
$$\bigstar_{X,Y}(N_1 \square N_2) = \bigstar_{X\cap X_1\cap X_2, Y\cap Y_1\cap Y_2}(\bigstar_{X\cap X_1, Y\cap Y_1}(N_1) \square \bigstar_{X\cap X_2, Y\cap Y_2}(N_2))$$
$$\bigstar_{X,Y}(\diamond_{X',Y'}(N_i)) = \diamond_{X'\cap X_i, Y'\cap Y_i}(\bigstar_{(X\cap X_i)\setminus X', (Y\cap Y_i)\setminus Y'}(N))$$

These equations are cluttered and quite difficult to read, but in the case of the operators $\diamond(-)$ and $\bigstar(-)$ (the most ones commonly used) they can be simplified to:

$$\diamond(N_i) = \diamond_{X_i, Y_i}(N_i)$$
$$\diamond(N_1 \triangle N_2) = \diamond(N_1) \triangle \diamond(N_2)$$
$$\diamond(N_1 \square N_2) = \diamond(N_1) \square \diamond(N_2)$$
$$\diamond(\bigstar(N_i)) = \diamond_{X_i, Y_i}(\bigstar_{X_i, Y_i}(N_i))$$
$$\bigstar(N_i) = \bigstar_{X_i, Y_i}(N_i)$$
$$\bigstar(N_1 \triangle N_2) = \bigstar_{X_1\cap X_2, Y_1\cap Y_2}(\bigstar_{X_1, Y_1}(N_1) \triangle \bigstar_{X_2, Y_2}(N_2))$$
$$\bigstar(N_1 \square N_2) = \bigstar_{X_1\cap X_2, Y_1\cap Y_2}(\bigstar_{X_1, Y_1}(N_1) \square \bigstar_{X_2, Y_2}(N_2))$$
$$\bigstar(\diamond(N_i)) = \diamond(N_i)$$

6.5 Why is this useful?

The main goal of this chapters is to endow nets with interfaces that allow communications through channels via addresses. This is useful since it allows us to represent, say each smart contract/piece of code as a net of its own. Sequential and parallel composition makes drafting processes (which can be programs such as smart contracts) easy, while the quotienting operator tells us how they behave when put together. This gives a very nice trade-off between making our nets more powerful and preserve the analysis tools we used up to now.

In the next chapters, our goal will be to extend our categorification efforts to open nets. We already saw in Chapter 4 how symmetric monoidal categories are important in representing all the possible computation of a net, which can in turn be mapped to some other semantic category as we did in Chapter 5. This technique will have to be generalized, while other categorical approaches will be employed to obtain better analytical tools to frame problems such as reachability and liveness.

Bibliography

[1] P. Baldan, F. Bonchi, F. Gadducci, and G. V. Monreale, "Encoding asynchronous interactions using open petri nets," in *CONCUR 2009 - Concurrency Theory*, M. Bravetti and G. Zavattaro, Eds., vol. 5710, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 99–114. [Online]. Available: `http://www.math.unipd.it/~baldan/Papers/Soft-copy-pdf/ModularEncoding.pdf` (cit. on p. 59).

[2] F. Borceux, *Handbook of Categorical Algebra: Volume 1, Basic Category Theory*. Cambridge: Cambridge University Press, 1994, p. 358 (cit. on p. 31).

[3] E. Brady, "Idris, a General-Purpose Dependently Typed Programming Language: Design and Implementation," *Journal of Functional Programming*, vol. 23, no. 5, pp. 552–593, 2013 (cit. on p. 55).

[4] ——, *Type-Driven Development With Idris*. New York, NY: Manning Publications, 2017, p. 480 (cit. on p. 55).

[5] V. Buterin, "A next-generation smart contract and decentralized application platform," *Etherum*, no. January, pp. 1–36, 2014. [Online]. Available: `http://buyxpr.com/build/pdfs/EthereumWhitePaper.pdf` (cit. on p. 51).

[6] B. Coecke and A. Kissinger, *Picturing Quantum Processes. A First Course in Quantum Theory and Diagrammatic Reasoning*. Cambridge: Cambridge University Press, 2017 (cit. on pp. 28, 49).

[7] E. Czaplicki, "Elm: Concurrent FRP for Functional GUIs," *Master thesis, Harvard University*, no. March, p. 55, 2012. [Online]. Available: `https://www.intranet.seas.harvard.edu/academics/undergraduate/computer-science/thesis/Czaplicki.pdf` (cit. on p. 56).

[8] ——, *Elm Homepage*, 2018. [Online]. Available: `http://elm-lang.org/` (cit. on p. 56).

[9] S. Eilenberg and S. Mac Lane, "General Theory of Natural Equivalences," *Transactions of the American Mathematical Society*, vol. 58, pp. 231–294, 1945 (cit. on p. 18).

[10] B. Fong, "Decorated Cospans," *Theory and Applications of Categories*, vol. 30, no. 33, pp. 1096–1120, 2015 (cit. on p. 64).

[11] F. R. Genovese and J. Herold, "Executions in (semi-)integer petri nets are compact closed categories," 2018. [Online]. Available: `http://arxiv.org/abs/1805.05988` (cit. on pp. 3, 48, 50).

[12] N. Ghani, J. Hedges, V. Winschel, and P. Zahn, "Compositional Game Theory," 2016. [Online]. Available: `http://arxiv.org/abs/1603.04641` (cit. on p. 51).

[13] Haskell.org, *Haskell Homepage*, 2018. [Online]. Available: `https://www.haskell.org/` (cit. on p. 20).

[14] HaskellWiki, *The Category Hask*, 2012. [Online]. Available: `https://wiki.haskell.org/Hask` (cit. on p. 54).

[15] C. A. R. Hoare and Elliott Brothers Ltd., "Algorithm 64: Quicksort," *Commun. ACM*, vol. 4, no. 7, p. 321, 1961. [Online]. Available: https://dl.acm.org/citation.cfm?doid=366622.366644 (cit. on p. 52).

[16] G. M. Kelly and M. L. Laplaza, "Coherence for Compact Closed Categories," *Journal of Pure and Applied Algebra*, vol. 19, pp. 193–213, 1980 (cit. on p. 49).

[17] M. Köhler-Bußmeier, "A survey of decidability results for elementary object systems," *Fundamenta Informaticae*, no. 1, pp. 99–123, 2014. [Online]. Available: https://www.researchgate.net/publication/262001792_A_Survey_of_Decidability_Results_for_Elementary_Object_Systems (cit. on p. 59).

[18] S. Mac Lane, *Categories for the Working Mathematician*, S. Axler, F. W. Gehring, and K. Ribet, Eds., ser. Graduate Texts in Mathematics. New York: Springer, 1978, vol. 5, p. 262. [Online]. Available: http://link.springer.com/10.1007/978-1-4757-4721-8 (cit. on pp. 4, 19, 20, 27, 31).

[19] S. MacLane, *Mathematics: Form and Function*. New York, 1986, p. 476 (cit. on p. 19).

[20] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," *Www.bitcoin.org*, pp. 1–9, 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf (cit. on p. 51).

[21] M. Nielsen, "Models for concurrency," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 520 LNCS, 1991, pp. 43–46 (cit. on p. 7).

[22] C. Petri and W. Reisig, "Petri net," *Scholarpedia*, vol. 3, no. 4, p. 6477, 2008. [Online]. Available: http://www.scholarpedia.org/article/Petri%7B%5C_%7Dnet (cit. on p. 7).

[23] R. Riemann, *Modelling of Concurrent Systems: Structural and Semantical Methods in the High Level Petri Net Calculus*. Munchen: Herbert Utz Verlag, 1999, p. 252 (cit. on p. 7).

[24] V. Sassone, "On the Category of Petri Net Computations," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 915, 1995, pp. 334–348 (cit. on pp. 37, 38, 45, 46).

[25] P. Selinger, "A survey of graphical languages for monoidal categories," in *New structures for physics*, vol. 813, Springer, 2010, pp. 289–355 (cit. on p. 29).

[26] P. Sobociński, "Representations of Petri net interactions," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6269 LNCS, 2010, pp. 554–568 (cit. on p. 51).

[27] V. Winschel and M. Kraetzig, "Solving, Estimating, and Selecting Nonlinear Dynamic Models Without the Curse of Dimensionality," *Econometrica*, vol. 78, no. 2, p. 803, 2010. [Online]. Available: http://dx.doi.org/10.3982/ECTA6297 (cit. on p. 51).