Chapter 7
Petri Nets: A Formal Language to Specify and Verify Concurrent Non-Deterministic Event Systems

Didier Buchs, Stefan Klikovits, and Alban Linard

Abstract The study of concurrent and parallel systems has been a challenging research domain within cyber-physical systems community. This chapter provides a pragmatic introduction to the creation and analysis of such system models using the popular Petri nets formalism. Petri nets is a formalism that convinces through its simplicity and applicability. We offer an overview of the most important Petri nets concepts, analysis techniques and model checking approaches. Finally, we show the use of so-called High-level Petri nets for the representation of complex data structures and functionality and present a novel research approach that allows the use of Petri nets inside Functional Mock-up Units and cyber-physical system models.

Learning Objectives

After reading this chapter, we expect you to be able to:

- Use common Petri-net patterns to model concurrent processes
- Understand the semantics of Petri-nets in terms of state transitions systems
- Use model checking to systematically check for invariance and reachability properties of Petri-net models

7.1 Introduction

Since the early days of computing the modelling and verification of programs has been an important subject. Nowadays this subject is even more vital as computers are ubiquitous in our current way of life. Computers thrive in all kinds of environments, and some of their applications are life critical. Indeed, more and more lives depend on the reliability of airborne systems, rail signalling applications and medical device software for examples.

Given the importance of the matter, much effort has been invested to ensure the quality of the software. On the organisational side, project management techniques have been devised for the software development process, e.g. the RUP (Rational Unified Process) [174], the Waterfall Model [244], the Spiral Model [40], B-method [189], etc. Most recently, so-called "agile" methodologies, such as SCRUM [261], are taking over the

Didier Buchs
Faculty of Science, Computer Science Department, University of Geneva, Switzerland
e-mail: didier.buchs@unige.ch

Stefan Klikovits
Faculty of Science, Computer Science Department, University of Geneva, Switzerland
e-mail: stefan.klikovits@unige.ch

Alban Linard
Faculty of Science, Computer Science Department, University of Geneva, Switzerland
e-mail: alban.linard@unige.ch

P. Carreira et al. (eds.), *Foundations of Multi-Paradigm Modelling for Cyber-Physical Systems*,
https://doi.org/10.1007/978-3-030-43946-0_7

industry [249]. However, software development frameworks can improve software quality only up to a certain point. In fact, they cannot offer complete guarantees for critical systems by themselves as their effectiveness is only based only on empirical evidence [249].

These approaches have the fact in common that all of them require the description of what the system does without prescribing how to do it. That description is called the *specification*. Depending on the development process used, the specification can be informal (e.g. SCRUM), semi-formal (e.g. RUP) or formal (e.g. B-method). Getting the specification right is paramount for the software quality. On the one hand, the specification is used to check if the development team understood the requirements (to answer the question "Are we building the right thing?"). This process is called *validation*. On the other hand, it is used to check if the finished software does what it was meant to do ("Are we building the thing right?"). We call this step *verification*.

A very simple way to do verification is *testing*. In software testing, we use the specification to derive behaviours that we expect from the software. For each expected behaviour we write a *test*. A test is a procedure that exercises the software (or a part of it), and tells if the observed behaviour is as expected or not (according to the specification). Hence, a test can prove that there are errors in the software. However, proving the absence of errors is much more complicated. It implies to write a test for each possible behaviour of the software. The number of behaviours of even simple software is extremely large, meaning that testing is infeasible for proving the absence of errors. Nevertheless, there are some kinds of software that cannot afford to diverge from specification as human life or health depends on it. This need gave birth to a set of verification techniques that can guarantee the absence of errors in a given system: *formal verification*.

Formal verification techniques, a.k.a. *formal methods*, can guarantee the absence of errors in a system up to its modelling. There are several formal methods ranging from *theorem proving* to *model checking*. These techniques aim to build a formal mathematical proof of the program's correctness. This requires of course that the specification is also *formally* described. It further requires that the program itself has a formally specified semantics. In these sections we focus on the modelling phase and the model checking variant of formal methods.

Formally modelling complex systems requires languages that are adapted to the kind of system we are interested in and also must be defined with certain structuring mechanisms. In this chapter, we will mainly describe languages that provide features related to dynamic systems and data types. For structuring mechanisms we propose to consult publications on extensions of algebraic nets such as CO-OPN [32] and LLAMAS [197]. We will not describe them further as they are not absolutely necessary for the understanding of the basic concepts behind formal methods.

7.2 Modelling Concurrency

The modelling of concurrency requires specially adapted formal techniques. Among the numerous existing ones, we observe that all of them use constructs to either explicitly or implicitly describe events, states and synchronisation mechanisms. Moreover, the technique's semantics must exhibit the various behaviours of the modelled system because parallelism and concurrency inherently introduce activity non-determinism into a system. One of the most well-known formalisms is Petri nets, which we will introduce throughout this chapter. However, it is important to understand that most of the explained principles can be translated to other kinds of models such as process algebra [207], state charts [135] and similar.

In order to explain the essence of modelling with Petri nets, we use an example of a car engine throughout this paper. The system describes a very simple combustion engine consisting of foot pedal, engine, carburettor and fuel tank. The system is built from several elementary components, that are similar to automata used in some modelling tools [277]. We will see how all components are combined in a single Petri net that describes the entire system.

Although we could use a model of communicating automata for such problems, the use of a Petri net makes communication between components explicit, whereas it is often implicit and hence unclear when using automata. For instance, the choice of synchronous versus asynchronous communication is a meta-property in automata, rather than an explicit choice.

So, in summary, we will have the possibility to model systems with Petri nets in a condensed way. These systems will be composed of some non-deterministic entities communicating synchronously or asynchronously.

7.2.1 Petri Nets

Petri nets [237] are a graphical, formal modelling language dedicated to the representation of concurrent processes, including communication and synchronisation between them. Petri nets consist of four basic concepts: *places*, *transitions*, *arcs* and *tokens*. Places (represented by circles) model processes and resource containers (such as a fuel tank) of the system, while transitions (represented by rectangles) are used to model system evolution (e.g. the starting of an engine). Places and transitions are connected by arcs. Arcs describe how many tokens (represented as large dots) are taken from a place before and how many tokens are put into a place after a transition is executed. We usually refer to these arcs as the *pre-* and *postconditions* of a transition. Tokens are used to model resources (e.g. the petrol in our example) or control (whether an engine is on or off). Note, that in classic Petri nets only one kind of token is used, meaning that the process control and the resources are both represented equally, which might be confusing for beginners. This feature however offers great flexibility and renders Petri nets a very powerful formalism.

Figure 7.1 shows an example of a Petri net, that represents in a simple way the behaviour of a throttle foot pedal, engine and fuel tank within a car. It represents processes and resources as places: e.g. `up`, `off`, `fuel` or `filled`. The Petri net also contains transitions such as `press`, `stop` or `empty`. It can be easily observed that the amount of fuel is modelled by the number of tokens within the place fuel, but also that control of the engine is being handled by an individual token which is either in place `on` or `off`. An execution of a transition (e.g. `start`) will *consume* tokens from the precondition states (`off`) and produce tokens in postcondition states (i.e. `on` in our example).

The groupings (i.e. the big frames) shown for foot pedal, fuel tank, engine and carburettor have no semantic influence on the system, but merely help system description.

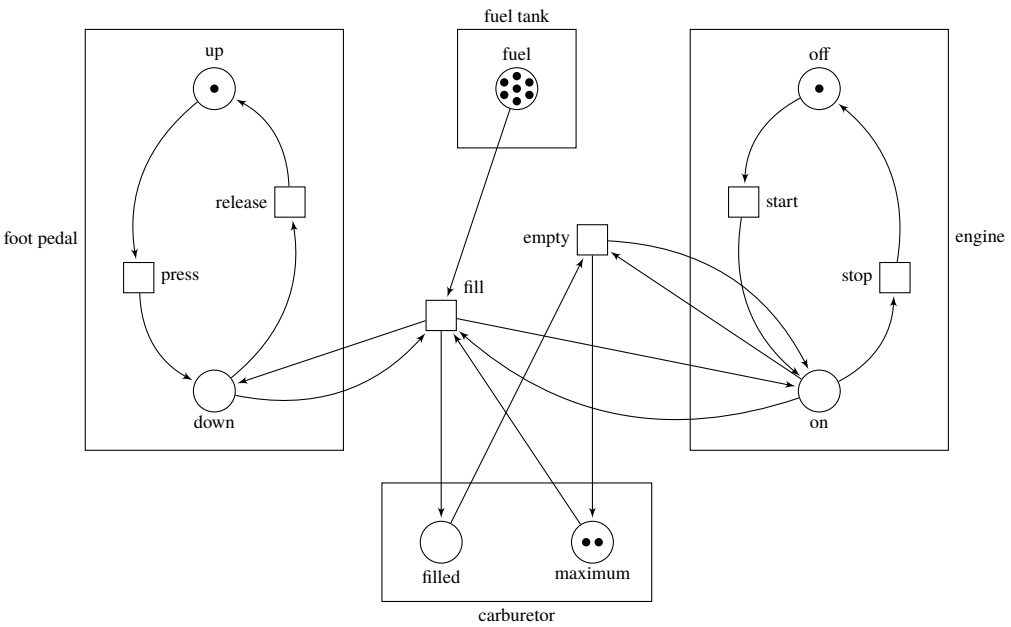


Fig. 7.1: A Petri net that represents a simplified functionality of a car's acceleration

Each places contains a (positive) number of *tokens* (0 by default), that describes how many processes or resources are in this state. We refer to the number of tokens in a place as its *marking*. The combination of markings of all Petri net places is also referred to as the Petri net's *marking* and describes the entire system's state.

Due to their well-studied semantics, they can be simulated, or used for static or dynamic analysis. Their simplicity makes it very easy to transfer them to other forms. For instance, Figure 7.2 shows the marking of the Petri net shown in Figure 7.1 encoded in the Lua programming language[1]. This Petri net can thus be embedded

[1] https://lua.org

and used within a software application. We will use the Lua language throughout this article to show how to implement and use Petri nets in practice. The `marking` of the Petri net is a structure that affects to each place its number of tokens. The `initial` marking is the one described initially in the Petri net.

```lua
function Marking.new (t)
  return {
    up      = t.up      or 0,
    down    = t.down    or 0,
    fuel    = t.fuel    or 0,
    off     = t.off     or 0,
    on      = t.on      or 0,
    filled  = t.filled  or 0,
    maximum = t.maximum or 0,
  }
end
```

```lua
initial = Marking.new {
  up      = 1,
  fuel    = 7,
  off     = 1,
  maximum = 2,
}
```

Fig. 7.2: Declaration of a Petri net marking as Lua code. The `new` function (left) of the `Marking` module is used to create a mapping that represents the Petri net marking. The code on the right shows the usage of this function for the creation of a new marking. Note, that it is only necessary to specify the places that contain tokens, as the other tokens are initialised with `0` by default.

A Petri net state (represented by its marking) is changed by "firing" transitions. In order to more efficiently describe system evolution we will from use markings, and operations thereon to express the behaviour of a Petri net. For convenience, we therefore define the following operators on markings: comparison, addition and subtraction. The comparison operator compares the number of tokens in each place of the two operands for equality or whether one of the operands is smaller/greater. Addition and subtraction operators perform place-wise addition or subtraction of the number of tokens of the operands. These operators allow the concise description of the transition firing semantics. Figure 7.3 provides the operator implementations in Lua.

As stated, the evolution of a Petri net is effected by firing transitions. Such transition firings are dominated by a simple rule: a transition can be fired if it is *enabled*. A transition is called enabled, iff there are enough tokens in all its input places (i.e. the places connected to the transition). When a transition is fired, it removes tokens from its precondition places, and adds new tokens into its postcondition places. It is important to understand that tokens do not "move" from one place to another, but are *consumed* and new (different) tokens are *produced*. The number of tokens consumed and produced are defined using annotations on arcs. By convention arcs annotations stating a weight of 1 token are omitted for legibility reasons.

For an illustration of a transition firing we can look at the `empty` transition from the car engine example above. The transition consumes one token from place `filled`, and another token from place `on`. Therefore it can be fired if the places `filled` and `on` contain *at least* one token each. Once fired, the transition produces one token in place `maximum`, and another one in place `on`. Figure 7.4 shows the Lua code that describes the firing of the transitions `press` and `empty`. It first checks if the input places contain enough tokens. If this condition is met, it performs firing by subtracting the precondition arcs and adding the postcondition arcs to the marking of the Petri net. If the transition is not fireable, the function returns **`nil`**, the Lua equivalent for `NULL` or `null` value.

7.2.2 Common Petri net patterns

Petri nets are better at expressing communication and synchronisation than the traditional automata formalism, as they make these concepts explicit. However, similar to automata, Petri nets have the aforementioned drawback to not distinguish processes and resources. It is up to the modeller to clarify the role of each place, for instance with naming or colour conventions. In order to introduce some common modelling practices we present a few standard patterns which are found in the Petri nets in Figure 7.5:

- Figure 7.5a represents the creation of two processes (q, r) from a process (p), or the release of a resource (r) from a process (p, q).

```
function Marking.__add (l, r)
  return {
    up      = l.up      + r.up,
    down    = l.down    + r.down,
    fuel    = l.fuel    + r.fuel,
    off     = l.off     + r.off,
    on      = l.on      + r.on,
    filled  = l.filled  + r.filled,
    maximum = l.maximum + r.maximum,
  }
end
```

Listing 8: Addition operator

```
function Marking.__eq (l, r)
  return l.up      == r.up
     and l.down    == r.down,
     and l.fuel    == r.fuel,
     and l.off     == r.off,
     and l.on      == r.on,
     and l.filled  == r.filled,
     and l.maximum == r.maximum,
end
```

Listing 9: Equality operator

```
function Marking.__sub (l, r)
  return {
    up      = l.up      - r.up,
    down    = l.down    - r.down,
    fuel    = l.fuel    - r.fuel,
    off     = l.off     - r.off,
    on      = l.on      - r.on,
    filled  = l.filled  - r.filled,
    maximum = l.maximum - r.maximum,
  }
end
```

Listing 10: Subtraction operator

```
function Marking.__le (l, r)
  return l.up      <= r.up
     and l.down    <= r.down,
     and l.fuel    <= r.fuel,
     and l.off     <= r.off,
     and l.on      <= r.on,
     and l.filled  <= r.filled,
     and l.maximum <= r.maximum,
end
```

Listing 11: Less-than-or-equals op.

Fig. 7.3: Addition, subtraction, equality and less-than-or-equals operators used to work on Petri net markings. The evolution of a Petri net can be expressed using these operators.

```
function press (marking)
  if marking >= { up = 1 } then
    return marking
        - { marking.up   = 1 }
        + { marking.down = 1 }
  else
    return nil
  end
end
```

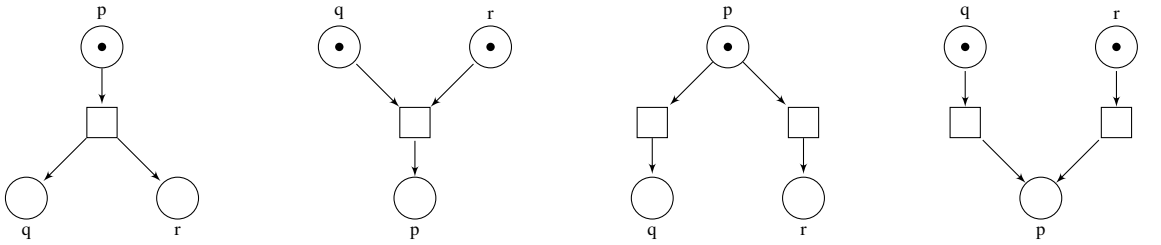
Listing 12: Code of the press transition

```
function empty (marking)
  if marking >= { filled = 1,
                  on     = 1 }
  then
    return marking
        - { marking.filled  = 1,
            marking.on      = 1 }
        + { marking.maximum = 1,
            marking.on      = 1 }
  else
    return nil
  end
end
```

Listing 13: Code of the empty transition

Fig. 7.4: Lua code as examples showing the implementation of transitions

- Figure 7.5b represents the synchronisation of two processes (q, r), or the acquire of a resource (r) from a process (q, p).
- Figure 7.5c represents a choice for process p, that can go either in q branch or in q branch.
- Figure 7.5d represents the collection of processes or resources (q, r) into one (p). When q and r are in mutual exclusion, it can also represent the end of a condition for process p.



(a) Control flow split or resource creation

(b) Synchronization or resource acquisition

(c) Control flow choice

(d) Control flow synchronisation point

Fig. 7.5: These four common Petri net patterns can be used to express various concepts related to control flow split, merge, choice or synchronisation, but also for the consumption, production, acquisition and release of resources.

7.2.3 Formal syntax and semantics

Since we introduced Petri nets as a formal method, we feel obliged to also provide the syntax and semantics in a more formal setting. We encourage the reader to bear with us through this section, as understanding of these concepts will be necessary for the more complex aspects of Petri nets. In return, we promise to keep ourselves short and only introduce the essential parts. It is also worth mentioning that these formal definitions can be used directly for the creation of tools that manage, simulate and analyse models.

Formally, a Petri net is a *labelled bipartite directed graph*, i.e., a graph whose vertices can be divided into two disjoint finite sets P and T (with $P \cap T = \emptyset$), such that every edge is directed and connects a vertex of P to a vertex of T (preconditions), or a vertex of T to a vertex of P (postconditions). This graph has natural number labels on vertices of P (place markings), no labels on vertices of T (transitions), and positive number labels on edges (number of consumed or produced tokens).

A Petri net is thus described as a tuple $\langle P, T, pre, post, m_0 \rangle$. The set of all possible markings is denoted $M = P \to \mathbb{N}$. It contains all possible functions that associate tokens to places of the Petri nets. The initial marking m_0 is an element of M that maps all places to their initial number of tokens. The $pre : T \to M$ and $post : T \to M$ functions represent arc valuations, and are called the $pre-$ and $post$conditions of the transitions. For every transition, they return a marking that corresponds to the valuations of input (resp. output) arcs of the transition. Note, that their signature is implicitly in curried form, as they can be rewritten as $pre : T \to (P \to \mathbb{N})$ and $post : T \to (P \to \mathbb{N})$.

The arithmetic and comparison operators $(+, -, =, \leqslant)$ that we introduced in the previous section are formally defined on markings as follows. We require that $\forall m_l, m_r \in M$

$$
\begin{aligned}
m_l + m_r &= \forall p \in P, (m_l + m_r)(p) \mapsto m_l(p) + m_r(p) \\
m_l - m_r &= \forall p \in P, (m_l - m_r)(p) \mapsto m_l(p) - m_r(p) \\
m_l = m_r &\equiv \forall p \in P, m_l(p) = m_r(p) \\
m_l \leqslant m_r &\equiv \forall p \in P, m_l(p) \leqslant m_r(p)
\end{aligned}
$$

where $a \mapsto x$ signifies that x is the image value of a.

In this chapter we will describe the evolution and behaviour of Petri net using Structured Operational Semantics (SOS) rules such as the example in Equation (7.1). We remind the reader that an SOS rule is composed of a conjunction of premises (above the line) and a conclusion (below the line). The rule states that it can be applied i.e. the conclusion transformation be executed, when the premises are satisfied. Both the premise(s) and the conclusion are expressions (usually called "term predicates").

$$
rule : \frac{tpred_1 \wedge tpred_2 ... \wedge tpred_n}{tpred} \tag{7.1}
$$

Each Petri net transition describes possible evolutions of the system. Formally it is a relation between Petri net states, i.e. markings. Thus, the behaviour is described as a transformation of the Petri net marking. The

conclusion of SOS rules that describe transition firings are written as $m \xrightarrow{t} m'$, where m and m' are Petri net markings and t is a transition. $m \xrightarrow{t} m'$ states that the system's current marking m leads to m' by firing t.

Using a model checking mentality we can also say that the SOS rule states that, given two markings m and m', firing t is correct if all the premises are correct. The semantics of firing one transition is given by the general rule in Equation (7.2).

$$transition_t : \frac{pre(t) \leqslant m}{m \xrightarrow{t} m - pre(t) + post(t)} \tag{7.2}$$

Example Let us consider the `release` transition in Figure 7.1. Its semantics is represented by the following rule:

$$\texttt{release} : \frac{(\mathsf{down} \mapsto 1) \leqslant m}{m \xrightarrow{\texttt{release}} m - (\mathsf{down} \mapsto 1) + (\mathsf{up} \mapsto 1)}$$

Remember that for a marking m to be greater or equal than another one, all individual place markings have to be greater or equal. For this specific premise to be valid, it means that all places in m need to have a greater or equal marking than $(\mathsf{down} \mapsto 1)$ (i.e. all places need to have at least 0 tokens, except for down which needs at least 1 token). If and only if this is the case, then firing the transition will lead to a new marking by removing one token from down and adding one token to up. □

Sequences of transitions also have simple semantics: beginning from a starting state, the transitions within the sequence are applied one by one. Formally, sequences of transitions (noted T^*) are inductively defined as lists. Note, that every transition $t \in T$ can also be seen as a sequence of transitions of length 1. Hence $T \subseteq T^*$. Further, we use a concatenation operator (.) such that $\forall t \in T, \forall s \in T^*, t.s \in T^*$, i.e. every concatenation of transitions is a transition, followed by a sequence of a transitions. Note that a the sequence s can be a single transition. This semantics is given by Equation (7.2) and Equation (7.3).

$$sequence : \frac{m \xrightarrow{t} m', m' \xrightarrow{s} m''}{m \xrightarrow{t.s} m''} \tag{7.3}$$

Figure 7.6 shows the Lua code that performs the computation of a sequence of transition. Each `transition` is itself a function, such as those we have already defined for `press` and `empty`.

```
function sequence (marking, transitions)
  for _, transition in ipairs (transitions) do
    marking = transition (marking)
    if not marking then
      return nil
    end
  end
  return marking
end
```

Fig. 7.6: The Lua code that handles the execution of a sequence of transitions on an initial marking. Note that `transitions` is a list of transition functions as defined above. The `ipairs` function is an ordered iterator over the elements of `transitions`.

7.2.4 Deduction Based on Rules

As we have seen above, we can compose new relations from existing rules, as shown for the sequences in Equation (7.3). We refer to this feature as *transitive closure*. However, in order to compute new relations based on existing SOS rules, we need to know exactly how to compute a rule's term predicates, as another rule's premises can be used as a conclusion of another rule.

This process, called deduction, is based on three principles: rule matching, variable instantiation and predicate reduction.These principles require the definition of additional operations on functional terms (the expressions including only application of function) and term predicates (the predicates applied on functional terms).

These principles are related to the use of variables. The operations that manage the values which can be taken by these variables can take must be thoroughly defined.

An assignment (or *substitution*) is defined as a function $\sigma \in \Sigma$, where Σ is the set of partial functions from variables to terms, or $\perp$ if no term is associated to the variable. For instance, a substitution σ can be the replacement of the variable a with the term $\texttt{down} \mapsto 1$, or by $\perp$ if the variable a is undefined. Substitution is extended to apply on terms instead of variables only: for all terms t, $\sigma(t)$ returns the term t where all variables that are defined in σ have been replaced by their associated term in σ. For instance if we take the term $pre(x)$ and apply the substitution $\sigma(x) = \texttt{release}$ onto this term we see that $\sigma(pre(x)) = pre(\texttt{release})$.

Matching refers to the fact that we would like to identify two terms t_1, t_2 modulo a substitution σ. This means that matching checks if there exists a substitution σ such that $\sigma(t_1) = \sigma(t_2)$. For instance, the substitution $\sigma = \{x \mapsto \texttt{release}, y \mapsto \texttt{release}\}$ can be used to match $pre(x) = pre(y)$.

We would like to introduce a simple example defining the comparison predicate over natural numbers. According to the rules given in Equation (7.4): $a, b \in \mathbb{N}$ are variables over natural numbers.

$$R1 : \frac{a \leqslant b}{a \leqslant b + 1} \qquad R2 : \frac{a \leqslant b}{a + 1 \leqslant b + 1} \qquad R3 : \frac{}{0 \leqslant 0} \tag{7.4}$$

We assume that the reader is familiar with the operations based on addition of natural numbers. Their very definitions can be given as an exercise to the reader. Deductions are computed using a special *Infer* function. $Infer(R)$ is defined as the deduction process for a set of rules R. In our specific case Infer will be defined as $Infer(\{R1, R2, R3\})$ and we will use Infer to compute the possible deduction on the predicate $\leqslant$.

In such deductions rules are applied on top of each other, the correspondence being the possible matching of one premises with the conclusion of another rule placed above. Correct deductions have to be finite and also must rely on a base case, i.e. a rule without premises (in our case R3).

Example The following examples show possible deductions, based on the rules R1, R2 and R3. By applying a rule whose conclusion that matches the provided term, we can find a premise, which we then use to match against another rule. This means we generally work bottom-up and try to find matching rules which will lead us closer to R3.

On the left, we try to deduce that $1 \leqslant 2$. We observe that $1 \leqslant 2$ can be the conclusion of the rule R1, leading us to infer that the premise of this rule must be $1 \leqslant 1$. Using rule R2 and the $1 \leqslant 1$ as conclusion, we see that the premise for this rule is $0 \leqslant 0$. Now we can apply R3, which does not have any premise and we successfully deduced that $1 \leqslant 2$.

On the other hand we can look at the example to the right. Our goal here is to deduce that $4 \leqslant 2$ is true. A first application of rule R2 will get us to $3 \leqslant 1$ and another application of R2 to the premise $2 \leqslant 0$. At this point, none of the rules can be applied any more This means that, according to the rules we defined, we cannot reach a base case and that $4 \leqslant 2$ must be incorrect.

Note, that in both cases we could have applied different rules than the ones that we actually used. On the left hand we could have swapped the rules and first applied R2 before using R1. On the right side we could have also used R1 instead of R2. In either case we would have reached the same conclusions. In general, inferences can be performed in any order by using any rule that is applicable. The trick is to use rules that quickly lead to an end (either a clash or a base-case).

$$\dfrac{R1\dfrac{R2\dfrac{R3\dfrac{}{0 \leqslant 0}}{1 \leqslant 1}}{1 \leqslant 2}}{} \qquad \dfrac{R2\dfrac{R?\dfrac{???}{2 \leqslant 0}}{3 \leqslant 1}}{R2\dfrac{}{4 \leqslant 2}}$$

According to the principles we gave, we can say that $Infer(R)$ is the least set that includes the results of all substitutions possible on all conclusions of all rules that do not have premises:

$$\sigma(post(r)) \in Infer(R) \text{ if } pre(r) = \emptyset, \forall r \in R, \forall \sigma \in \Sigma$$

and all rules' substituted conclusions where a substitution exists so that a rule premise is in the set:

$$\sigma(\phi(post(r))) \in Infer(R) \text{ if } \phi(pre(r)) = \phi(d), \forall r \in R, \sigma \in \Sigma \exists \phi \in \Sigma, \forall d \in Infer(R)$$

where $pre(r)$ are is the premise and $post(r)$ the conclusion of the rule

This last rule is called modus ponens, Example 7.2.3 shows such a deduction for a sequence of transitions. $Infer(R)$ is then obtained by fixpoint application of the modus ponens rule. For rules with positive numerator (no negation, only conjunction) this set always exists according to fixpoint theorem of Knaster-Tarski [264]. Nevertheless, the set can be infinite.

Example Let us consider the sequence `press.release` in Figure 7.1. The semantics of this sequence are represented by the following simple deduction using the SOS rules of `press` and `release`. What we see is that the conclusion of the rule (the firing of `press.release` and creation of a new marking) has two premises: 1. the conclusion of the `press` rule, and 2. the conclusion of the `release` rule. Each of these is part of an SOS rule with it's own premises, which has to be satisfied for this rule to be applicable.

$$\dfrac{\dfrac{(\mathsf{up} \mapsto 1) \leqslant m}{m \xrightarrow{\texttt{press}} m' = m + (\mathsf{up} \mapsto 1) - (\mathsf{down} \mapsto 1)} \quad \dfrac{(\mathsf{down} \mapsto 1) \leqslant m'}{m' \xrightarrow{\texttt{release}} m'' = m' + (\mathsf{down} \mapsto 1) - (\mathsf{up} \mapsto 1)}}{m \xrightarrow{\texttt{press.release}} m''}$$

7.2.5 Reachability Graph

We saw that applying a transition to a Petri net marking generates a new marking. In order to study system evolutions in a more efficient manner, we can try to represent transitions and the thereby created markings as an automaton. The automaton's states are the markings and the automaton's transitions correspond to the Petri net's transitions. We refer to such an automaton as the *reachability graph* of a Petri net. This is due to the fact that it shows, starting from an initial state, the possible states of the system that can be reached.. To fully understand the definition of the automaton, it is noteworthy that the automaton's input alphabet is the Petri net's set of transitions.

Note, that depending on the initial marking, a Petri net has different reachability graphs, as different transitions might be enabled. To visualise this, we present the reachability graphs of the Petri net of Figure 7.1 in Figure 7.15 (at the end of this chapter). The two graphs correspond to the possible system evolutions with 0, 1, 2 and 3 initial tokens in `fuel` in Figures 7.15a – 7.15d, respectively. We can clearly see the repeated groups of markings that correspond to matching behaviour.

These groups of reachability graph nodes differ by the number of tokens available in the `fuel` place. To aid the legibility, we use different shades of gray to distinguish markings with different token numbers in place `fuel`, where the lightest is 0 and the darkest is 3. The last group at the bottom of the figures (almost white) shows the behaviour when `fuel` is empty. We can easily observe that the states of 0 tokens in Figure 7.15a is repeated in Figure 7.15b, but this time with 1 token (signalled by the darker shade of grey). In the same way, we can find the group of Figure 7.15b in Figure 7.15c, where however, every state is one shade darker, as there is one more token in state `fuel`. The reachability graph of the marking shown in Figure 7.1 with seven tokens in `fuel` has 84 states. We can observe from the graphs in Figure 7.15 that a Petri net with 0 tokens has 4 states

and 12 states with 1 token. The number of states increases by 12 for each additional token added initially to the `fuel` place.

The advantage of Petri nets is that, due to their better representation of concurrent processes and resources, they can be *orders of magnitude* smaller than their equivalent automata. For instance, [300] managed to simulate the state of several multi-threaded programs to perform intrusion detection. Simulation was performed on-the-fly during the program execution by wrapping system calls, and thus required an efficient representation and computation time. This goal was achieved using Petri nets, and it would not have been possible using the equivalent automata on the available resources.

Formally, the reachability graph of a Petri net is defined by a function $s : M \to T \to M \cup \{\bot\}$ that returns the successor for any marking and transition, or $\bot$ if none exists. This function is derived from the following basic rules:

$$\forall m \in M, t \in T, \begin{cases} s(m)(t) = \bot & \Leftrightarrow \nexists m' \in M, m \xrightarrow{t} m' \in Infer(\{transition_{t_i} | t_i \in T\}) \\ s(m)(t) = m' & \Leftrightarrow m \xrightarrow{t} m' \in Infer(\{transition_{t_i} | t_i \in T\}) \end{cases} \tag{7.5}$$

The relation can be restricted to the subset of states that are reachable from the initial state m_0 of the Petri net. This subset is easily defined using the sequences of transitions of Equation (7.3).

We define that the state space (SS) is the set of reachable markings starting from an initial marking m_0 for a given set of transitions T

$$SS(T, m_0) = \{m \in M \mid \exists s \in T^*, m_0 \xrightarrow{s} m \in Infer(\{transition_{t_i} | t_i \in T\} \cup \{sequence\})\} \tag{7.6}$$

By convention, for a given Petri net it is natural to implicitly define $SS(m_0) = SS(T, m_0)$.

The (naive) reachability graph generation algorithm is given in Figure 7.7. This function iterates over the reachable markings, initially only `initial`, and tries to apply all transitions. The function returns the set of reachable markings, each one annotated with the transitions that can be fired, and the corresponding successor marking. The `explored` set contains the already explored markings, whereas the `encountered` set contains markings that have not been explored, or that have been encountered again since their exploration. The code shows a simple implementation that iterates over all previously seen markings to ensure uniqueness. More efficient implementations may use a hash table.

7.2.6 Monotony

Petri nets exhibit an interesting property: monotony. Monotony means that if a transition can be fired for one marking, then it is also fireable for any marking greater than the evaluated one. Extended to the reachability graph, monotony means that all sequences of transitions that exist in the reachability graph of a Petri net also exist in reachability graphs of the same Petri net with greater initial markings. Trivially expressed we could say that adding tokens to a Petri net can only add new behaviours, but never inhibit existing ones.

Formally, monotony is defined by Equation (7.7), which can be derived from the rules in Equation (7.2) and Equation (7.3). This rule defines monotony on both, a transition and a sequence of transitions.

$$monotony : \frac{m \xrightarrow{t} m'}{m + \delta \xrightarrow{t} m' + \delta} \tag{7.7}$$

In our example monotony means that any sequence of actions in our engine can also be performed if we add tokens to any state of the Petri net (such as adding more `fuel`). We can observe the inclusion of reachability graphs in Figure 7.15 for initial markings `fuel` = $\{0, 1, 2, 3\}$. The sequence `stop.release.start.press` is observed several times, but with different initial markings, leading to a varying number of tokens in the `fuel` place for each one.

```
local markings = {}

function unique (marking)
  for _, m in ipairs (markings) do
    if marking == m then
      return m
    end
  end
  markings [#markings+1] = marking
  return marking
end
```

```
function reachable ()
  local explored    = {}
  local encountered = {
    [unique (initial)] = true
  }
  while next (encountered) do
    local marking = next (encountered)
    encountered [marking] = nil
    if not explored [marking] then
      explored [marking] = true
      for _, transition in ipairs (transitions) do
        local successor = transition (marking)
        if successor then
          successor = unique (successor)
          marking     [transition] = successor
          encountered [successor ] = true
        end
      end
    end
  end
  return explored
end
```

Fig. 7.7: Source code function that enumerates reachable markings (right). The code iterates over the already encountered markings and tries to find all possible successors by attempting to fire all individual transitions on that marking and adding the successful attempts into the list of encountered markings.
The `unique` function ensures that each marking corresponds to only one entry in the Lua memory. It is required by the `explored [marking]` and `encountered [successor]` lookups.

To summarise, we can see that the reachability graph of a Petri net is always included in the reachability graph of the same Petri net with a greater initial making. While the markings of the sequences' states may differ for each one, the transition sequences are preserved.

In fact, since monotony is a property of all Petri nets, we can observe the following lemma. It states that the state space (and thus the reachability graph) inferred from just the transitions is equal to the state space inferred by the transitions and the monotony rule that we can define as:

$$SSM(T, m_0) = \{m \in M \mid \exists s \in T^*, m_0 \xrightarrow{s} m \in Infer(\{transition_{t_i} | t_i \in T\} \cup \{sequence, monotony\})\} \quad (7.8)$$

So that the following statement holds:

$$SS(T, m_0) = SSM(T, m_0) \quad (7.9)$$

We also know that behaviours are preserved when extending markings:

$$m \leq m' \Rightarrow SS(T, m) \leq SS(T, m') \quad (7.10)$$

Where $\leq$ on markings compare markings one by one.

7.3 Properties of Petri Nets

In the previous section we introduced Petri nets and the rules for the exploration of the reachable state space of an initial marking. In this section we investigate the properties of Petri nets using representatives of state properties and transition properties.

7.3.1 Marking Properties

The state properties or in Petri net-lingo *marking properties* of a Petri net can be used to analyse one configuration of a Petri net. One of the most common representatives for these kinds of properties is the *bound*. A place's bound refers to the minimum (lower bound) and the maximum (upper bound) number of tokens that can be reached by any marking. The minimum and maximum number of tokens of all places are commonly referred to as the *bound* of a Petri net. This means that the Petri net bound is composed of

- the greatest marking that is contained by any reachable marking (lower bound),
- and the smallest marking that contains any reachable marking (upper bound).

Formally, we can define the lower and upper bound as follows:

$$lower = \max\left(\{m \in M \mid \forall m' \in SS(T, m_0), m' \geqslant m\}\right) \tag{7.11}$$

$$upper = \min\left(\{m \in M \mid \forall m' \in SS(T, m_0), m' \leqslant m\}\right) \tag{7.12}$$

Figure 7.8 shows the code that computes the bound of a Petri net, using its reachable markings. The code works by iterating over all markings and computing the minimum and maximum number of tokens for each place. There exist algorithms to compute the bound without the use of the set of reachable markings. These algorithms use structural properties of the Petri net instead, such as the one shown in [188].

The importance of bounds is paramount as they are often used to detect dead parts in the model, i.e. places that can never contain any tokens. Bounds computation can further detect places that can have an infinite number of tokens. However, as an infinite number of tokens would (theoretically) require an infinite reachability graph, such a bound cannot be detected using algorithms such as the one presented below. In order to detect an infinite bound, we have to use a *coverability graph*, an adaptation of the reachability graph for which a detection of repetitive sequences is added. This is explained for example in [238].

In the Petri nets community the term *k-boundedness* is often used, where $k \in \mathbb{N}$ is a natural number. This notion of bound refers to the upper bound at the place level and helps describing a Petri net. We say that a place p is k-bounded iff it contains at most k tokens in any reachable marking i.e. $\forall m \in SS(T, m_0), m(p) \leqslant k$. We can further refer to Petri net as being k-bounded iff all its places are k-bounded, i.e. $\forall p \in P, p$ is k-bounded.

```
function bounds (marking)
  local markings = reachable (marking)
  local bound    = {
    up      = { minimum = math.huge, maximum = 0 },
    down    = { minimum = math.huge, maximum = 0 },
    fuel    = { minimum = math.huge, maximum = 0 },
    off     = { minimum = math.huge, maximum = 0 },
    on      = { minimum = math.huge, maximum = 0 },
    filled  = { minimum = math.huge, maximum = 0 },
    maximum = { minimum = math.huge, maximum = 0 },
  }
  for marking in pairs (markings) do
    bound.up.minimum = math.min (bound.up.minimum, marking.up)
    bound.up.maximum = math.max (bound.up.maximum, marking.up)
    ...
  end
  return bound
end
```

Fig. 7.8: Source code stub of the function that calculates the Petri net's bounds

7.3.2 Sequence Properties

While marking properties provide useful information about the state of a system, they do not give any information about *how* the system evolves. In contrast, sequence properties express the possibility to fire transitions in the reachability graph. One such property describes the liveness of a transitions. This property assigns a liveness level ($l_i \in \{l_0, l_1, l_2, l_3, l_4\}$) to each transition of the Petri net, so that it provides information about whether a transition is fireable, can/will be fireable or will never be fireable. The individual levels are defined as follows:

l_0: the transition is *dead*, i.e. can never fire;
l_1: the transition can fire at least once in at least one path of the reachability graph;
l_2: the transition is *quasi-live*, i.e. can fire infinitely often in at least one path of the reachability graph;
l_3: the transition is *live*, i.e. can fire infinitely often in all paths of the reachability graph;
l_4: the transition can always fire.

Note that the liveness constraints are ordered and that if a transition is l_k-live, it is also l_{k-1} live, except for l_0 (dead). The liveness level of transitions is a useful information to debug models, as this level should in theory match the intended behaviour of the process using by the transition. In practice, l_0-live transitions should almost never occur in models: finding a l_0-live transition is usually a clear indication of a bug within the model as it should not appear at all if the model was correct. Transitions that are l_1-live can be fired at least once, and have no really interesting meaning. They can appear in any behaviour, such as the representation of scripts or programs that terminate. l_2-live transitions trigger infinitely often, and are thus found in the behaviour of programs that do not terminate, for instance web servers. They are distinguished from l_3-live transitions, because the latter can fire infinitely often in all execution paths. Thus, l_2-live transitions are used usually in looping processes that can terminate, whereas l_3-live transitions are used usually in looping processes that cannot terminate. l_4-live transitions are in practice very rare, as they represent an action that can always happen. They can be found in user interfaces, where one process records the inputs and has thus transitions that can always be fired.

In order to find the liveness level of transitions, we can employ the algorithm provided in Figure 7.9. The computation works on the set of reachable markings and further uses the information about a reachability graph's strongly connected components (calculated using *Tarjan's strongly connected components algorithm* [263]). The algorithm works in several stages, treating each transition individually as follows:

1. the transition is initially set as both, dead (l_0) and always fireable (l_4); these flags are unset as soon as the transition can respectively be fired or cannot be fired in the reachability graph;
2. by iterating over the reachable markings, the transition is set as l_1 live if and only if there is a marking in which it can be fired;
3. l_2 flag is set using the strongly connected components of the reachability graph: a transition is l_2 live if a component contains a marking in which the transition can be fired; and that stays within the component;
4. l_3 flag is set similarly using the strongly connected components: a transition is set as l_3 live if and only if it is l_2 live, and can be fired within a component that has no outgoing transitions (that is called a final component).

7.3.3 Invariants

The actual aim of modelling is often to reason over certain problems. Invariants pose a convenient way of reasoning over Petri nets independent from their actual marking.

Invariant reasoning is part of the structural analysis of Petri nets. Generally, invariants are distinguished between place (P-)invariants and transition (T-)invariants. While the former is useful for deadlock detection [214], T-invariants serve the modelling of logic programs [215] and Horn clauses [187] as pointed out in [199].

The goal of invariant creation is the establishing of proofs based thereon. In general, proofs based on invariants have an algorithmic part that consists of computing invariants and a more analytic part, which is written by a human and describes the reasoning necessary for building the proofs. There is a large number proofs we can perform with invariants, ranging from state based proofs such as mutual exclusion to liveness properties. Such invariant-based proofs are interesting alternatives to the ones based on temporal logic properties. Although they tend to be less powerful, they can nevertheless provide parametric proofs more easily. This comes however at the cost of human contribution while temporal logic proofs are automateable.

```
function liveness (marking)
  local markings   = reachable (from)
  local components = tarjan (markings)
  for _, transition in ipairs (transitions) do
    transition.liveness = {
      l0 = true,
      l1 = false,
      l2 = false,
      l3 = false,
      l4 = true,
    }
    for marking in pairs (markings) do
      if marking [transition] then
        transition.liveness.l0 = false
        transition.liveness.l1 = true
      else
        transition.liveness.l4 = false
      end
    end
    for _, component in ipairs (components) do
      for marking in pairs (component) do
        if  marking [transition]
        and marking [transition].tarjan.component == component then
          transition.liveness.l2 = true
        end
      end
      transition.liveness.l3 = component.is_final and transition.liveness.l2
    end
  end
  return transitions
end
```

Fig. 7.9: Liveness level detection algorithm.

7.3.4 Formal Definition of Invariants

Each state $m \in SS(T, m_0)$ can be associated with an observation value. The observation is a summary of the state that contains only information relevant to the properties that are verified. Given a set of observations O, which observations are obtained through functions $\omega : M \to O$ that return an observation for any possible marking in M, ω is an invariant if and only if it returns the same observation for all the reachable states of the system. For instance, in Figure 7.1, the foot pedal is either up or down. In the marking, this means that there is one token in one of the places and none in the other at all times. Therefore $\omega : m \mapsto (m(\mathsf{up}) + m(\mathsf{down}))$ is a function that always returns the value 1 for all reachable states and is thus an invariant. More formally, ω is an invariant iff $\exists o \in O, \forall m \in SS(R, m_0), \omega(m) = o$.

The definition above for invariants is very general. It can be used if the users define the invariants themselves, but cannot help in deducing invariants from the structure of the Petri net. It is interesting to consider instead only functions that are linear combinations of elementary observation of places.

Usually, elementary observations are functions that for any place and marking return the marking of the given place. The elementary observations is thus defined as the function $o_e : P \to (M \to \mathbb{N})$ such that $\forall m \in M, \forall p \in P, o_e(p)(m) = m(p)$ These elementary observations can be instantiated using any place of the Petri net, for instance $o_e(\mathsf{up})$ is an observation. The set of all elementary observations is included into the observations. For convenience, we consider that writing the name of a place is equivalent to the elementary observation for this place. Hence we will use up to also refer to the function that returns the marking of this place: $m \mapsto m(\mathsf{up})$.

Observations can be composed as linear combinations of observations. For any observations $o_1, \ldots o_n \in O$, and constants $c_1, \ldots c_n \in \mathbb{N}$, an observation can be built to compute $c_1 * o_1 + \ldots c_n * o_n$, for instance $1 * o_e(\mathsf{up}) + 1 * o_e(\mathsf{down})$ (the places with coefficient 0 are not shown). The set of observations is thus defined by the linear combinations of elementary observations:

$$O = \{\Sigma_{p \in P}(c_p * o_e(p)) \mid \forall p \in P, c_p \in \mathbb{N}\}$$

Following this definition, if ω_1 and ω_2 are invariants, then $\omega_1 + \omega_2$ is also invariant. The addition of two invariants $\omega_1 + \omega_2 : M \to O$ is defined as: $\forall m \in M, (\omega_1 + \omega_2)(m) = \omega_1(m) + \omega_2(m)$. Also, if ω is an invariant then $\forall k \in \mathbb{N}, k * \omega$ is an invariant. The multiplication of an invariant with a constant is defined as: $\forall m \in M, (k*\omega)(m) = k*\omega(m)$. To compute invariants it is necessary to find the appropriate linear combination of elementary observations. The well-known Farkas algorithm can be employed for the finding of invariants, as explained in the next section.

7.3.5 Computing P-invariants

Discovering the P-invariants can be done using the Farkas algorithm [97]. The Farkas algorithm is an iteration process operating on a structure which corresponds to a Petri net's incidence matrix. For spatial reasons we do not provide the implementation of the algorithm but describe the (basic) functionality in textual form.

Its principle is to represent and manipulate a matrix where columns represent the transitions of the Petri net, and lines represent place markings or linear combinations of place markings. The algorithm works iteratively, performing the following actions:

1. add new lines to the matrix by building linear combinations of already existing lines, the aim of this process being to nullify columns (i.e. create columns that only contain zeros);
2. remove the lines that were used based on the summation;
3. remove the columns that are nullified;
4. remove all lines that are already expressed through other lines.

The process terminates when all columns are nullified. The solutions are linear combination of place markings that have a constant observation, and are thus invariants.

Example Below is an example of how to compute invariants following the given algorithm, on a very simple Petri net shown in Figure 7.10.

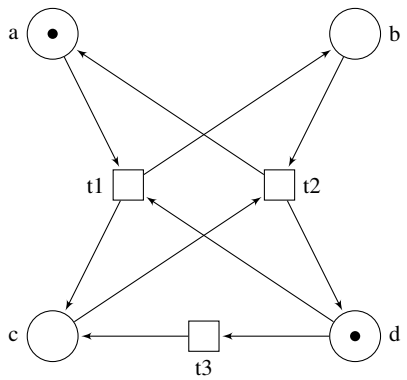


Fig. 7.10: Example of Petri net used to compute invariants

The successive F_i structures show the advance in Farkas algorithm. Each F_i is the result of one step, computed from the F_{i-1} structure. The first one (F_0) is the incidence matrix of the Petri net, that represents for each place and transition the difference between the weight on post arcs that rely the place and the transition, and the weight on pre arcs that rely them also.

$$F_0 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{ccc} t_1 & t_2 & t_3 \end{array} \\ \begin{bmatrix} -1 & 1 & 0 \\ 1 & -1 & 0 \\ 1 & -1 & 1 \\ -1 & 1 & -1 \end{bmatrix} \end{array}$$

The first iteration is to eliminate t_1. To do it, we have to find linear combinations of rows such that the column for t_1 is always zero. Obviously, both $a + b$, $c + d$, $a + c$ and $b + d$ are correct linear combinations. There is no need to represent greater coefficients, such as $2 * a + 2 * b$, thus only the minimal combinations are kept. Because all rows a, b, c, d have been used to compute the new rows, the rows a, b, c, d are removed from the matrix in F_1'. All columns that contain only zeros are also removed in F_1'', as they are useless for the remaining of the iterations.

$$F_1 = \begin{array}{c} \\ a \\ b \\ c \\ d \\ a+b \\ c+d \\ a+c \\ b+d \end{array} \begin{array}{c} \begin{array}{ccc} t_1 & t_2 & t_3 \end{array} \\ \begin{bmatrix} -1 & 1 & 0 \\ 1 & -1 & 0 \\ 1 & -1 & 1 \\ -1 & 1 & -1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{bmatrix} \end{array} \quad F_1' = \begin{array}{c} \\ a+b \\ c+d \\ a+c \\ b+d \end{array} \begin{array}{c} \begin{array}{ccc} t_1 & t_2 & t_3 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{bmatrix} \end{array} \quad F_1'' = \begin{array}{c} \\ a+b \\ c+d \\ a+c \\ b+d \end{array} \begin{array}{c} \begin{array}{c} t_3 \end{array} \\ \begin{bmatrix} 0 \\ 0 \\ 1 \\ -1 \end{bmatrix} \end{array}$$

As there is only one transition t_3 remaining, the second iteration is to remove it. Only one linear combination can do it: $a + b + c + d$, obtained by two ways: by adding the lines $a + b$ and $c + d$, and by adding the lines $a + c$ and $b + d$. But the Farkas algorithm requires to use only the rows that are not zero, thus $a + b$ and $c + d$ are not taken into account. As previously, the rows that have been used to compute the new ones are removed ($a + c$ and $b + d$) in F_2", and the nullified column is also removed in F_2''.

$$F_2 = \begin{array}{c} \\ a+b \\ c+d \\ a+c \\ b+d \\ a+b+c+d \end{array} \begin{array}{c} \begin{array}{c} t_3 \end{array} \\ \begin{bmatrix} 0 \\ 0 \\ 1 \\ -1 \\ 0 \end{bmatrix} \end{array} \quad F_2' = \begin{array}{c} \\ a+b \\ c+d \\ a+b+c+d \end{array} \begin{array}{c} \begin{array}{c} t_3 \end{array} \\ \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \end{array} \quad F_2'' = \begin{array}{c} a+b \\ c+d \end{array} \begin{bmatrix} \\ \end{bmatrix}$$

After this iteration, there are no more columns to remove. The remaining rows represent invariants of the Petri net: $a + b$ and $c + d$, that are named for later use:

$$i_1 = a + b$$
$$i_2 = c + d$$

As previously discussed, invariants are formulas that do not change applied on reachable markings. This means that for any sequence s and markings m, m', such that $m \xrightarrow{s} m'$, the invariant i applies at both ends of the sequence, and even at each state within the sequence: $i * m = i * m'$. In our example i is either i_1 or i_2, stating that the observation of $a + b$ and $c + d$ remains constant, independently of which transitions are fired.

7.3.6 Using Invariants for Proving Properties

Proving properties on models can be done by exploration of the reachability graph. Nevertheless, in most practical applications this method is difficult to use due to the state space explosion problem. The previous section defines invariants on Petri nets, that can be computed without the need to create the full reachability graph. In fact, invariants can be computed without even having to specify an initial marking. This section outlines the power of invariants by showing how to use them to prove properties on models. The usual way of performing these proofs is to establish invariants, and follow a case by case analysis of the possible state of the Petri net. This is done using in the reasoning the fact that invariants are respected in the whole reachability graph and also for the initial marking.

Proofs can usually be expressed as case analysis of the reachability graph of the Petri net. The proof process is then based on the following steps:

1. compute the set of invariants
2. apply invariants to the initial marking to obtain the corresponding observations
3. define an arithmetic reasoning based on the observations, invariants and case analysis, where the cases are in general markings of the state space that respect the invariants.

Using the invariants from Example 7.3.1, we can prove the principle of mutual exclusion of a and b in the Petri net above. The reasoning steps for this proof are as follows:

1. Application of the Farkas algorithm onto the incidence matrix provides us with the Petri net's invariants: $i_1 = a + b$ and $i_2 = c + d$.
2. Next, we calculate the invariant values using the initial marking $m_0 = \{a = 1, b = 0, c = 0, d = 1\}$. This leads to the inference that the value of $i_1(m_0) = m_0(a) + m_0(b) = 1$.
3. Since i_1 is an invariant, this observation needs to hold on any marking in the reachability graph, i.e. $\forall m \in M, m(a) + m(b) = m_0(a) + m_0(b) = 1$
4. Equipped with the knowledge that a place-marking cannot be negative ($\forall p \in P, \forall m \in M, m(p) \geqslant 0$) we can deduce that $m(a) \in \{0, 1\}$ and $m(b) \in \{0, 1\}$
5. Lastly, we can easily observe that the mutual exclusion has to hold since if $m(a) = 1$ and $m(a) + m(b) = 1$, then $m(b)$ has to have the value 0. Similarly, $m(a) = 0 \wedge m(a) + m(b) = 1 \Rightarrow m(b) = 0$

While this trivial proof can easily be computed by hand, the same principle can be used to create more complex, semi-manual, structural proofs. In practice such proofs are supported by powerful proof checkers such as Coq [30] to verify that hand written proofs are correct. The detailed description of this technique surpasses the scope of this book, but we encourage the reader to consult publications dedicated to this topic.

7.4 Techniques for Model Checking

Invariants can often be computed using less time and memory than a Petri net's the entire reachability graph. In some cases, however, such as during the computation of causality properties (for instance "if an event A occurs, then an event B must occur later"), the need to perform the generation of the state space is still required. Although the computation is straightforward (see the algorithm in Figure 7.7), this technique does not scale well since many models have an exponentially growing number of states with respect to to their size. It is not uncommon for a model's reachability graph to have more nodes than the number of atoms in the universe (10^{82}).

The size of the reachability graph of a Petri net usually depends on two factors: the number of places and transitions within the Petri net and the number of tokens in each place. For instance, the reachability graph of our example in Figure 7.1 increases with the number of tokens in the `fuel` place. It could also increase if we add more places, for instance to represent several positions of the foot pedal (up, middle, down for instance).

In order to still be able to calculate the state space, research led to the development of various approaches to overcome the increasing size of the state space. Below, we will briefly outline a few examples.

- *Symmetry reductions* [74] allow us to perform model checking on a smaller system, by analysing one representative component instead of several identical components. For instance, instead of analysing the behaviour of several databases, identified by their names, symmetry reductions focus on analysing the behaviour of one anonymised database.
- *Partial order reductions* [125, 163, 278] are based on the commutativity of concurrent actions. When performing model checking of asynchronous systems, action interleaving requires an arbitrary order between the events. To treat all possible cases, all permutations of the order must be considered, resulting in an exponential explosion of the number of traces and states. Partial order reductions allow to check only a subset of the behaviours, by removing executions that differ only by the order of independent transitions. This technique can only be applied when the property we want to check does not depend on the removed executions.
- *Büchi Automata* [296]: Some temporal logic formulæ (namely linear temporal logic ones) are checked against the set of infinite executions of the model. A way to perform this verification is to transform the formulæ into automata that accept the valid executions. Such automata are called Büchi automata. For

verification, the model and the formula are both encoded as Büchi automata which accept the languages that represent the executions of the model and the valid executions with respect to the property to verify. Model checking then consists of checking that the language of the executions is included in the language of the property.

- *Bounded Model Checking (BMC)* [33]: This approach is a kind of "degraded mode" of standard model checking, as the formula is checked only for executions of a maximum length k (where k is the sequence length from the initial marking). If no problem is detected, then k is increased until the formula does not hold any longer, or k reaches an upper bound (called the *completeness threshold of the design*). Note that there is no guarantee for executions longer than this upper bound. Since these kinds of problems can be reduced to propositional satisfiability problems, they can use very efficient SAT or SMT solvers.
- *Distributed Model Checking* [121, 22, 21]: One of the main problems when performing model checking is memory exhaustion. The idea of distributed model checking is to distribute the states over several computers in order to increase the overall available memory. This technique has a drawback: the transmission of data between computers poses as a bottleneck. Various algorithms have been designed to perform a "good" distribution of states so that most transitions are local to the same computer. They have reached their limits however due to the rise of modern, highly efficient CPUs. Distributed model checking is in practice most efficient when the time required to compute successor states is much longer than the time required to transmit states.
- *Parallel Model Checking* [257, 20]: With multi-core architectures, the trend is nowadays to perform parallel model checking instead of distributed model checking. In this approach, all the states are represented and computed on the same computer, but several execution processes are used to speed up the computation. This approach still grows more difficult with increasing parallelism, because of possible memory contention within storage for computed states. The same approach is used to perform parallel model checking on GPUs.
- *Symbolic Model Checking (SMC)* [58, 59, 77]: Instead of explicitly representing states and transition relations, this approach only represents and manipulates sets of states or sets of transitions using Decision Diagrams [55]. This representation allows at the same time to share some common parts of the states and reduce computations when applying the transition relation, in the best cases logarithmic with respect to the size of the model.

Some techniques have been developed that combine several approaches [92]. For instance, symmetry and partial order reductions are also often coupled with static reductions techniques that reduce the size of the specification with respect to the property to check, or with state compression as in [96], where some states are represented by the difference with their predecessor. Similarly, parallel model checking is easily used in combination with the other approaches discussed above. One exception is symbolic model checking because Decision Diagrams require a unicity table and computation caches, that need to be locked and are thus bottlenecks.

7.5 Data manipulation in Petri nets

The previous sections present the use of simple Petri nets models. While this type of Petri nets are sufficient to represent simple systems, they lack several features that are useful when dealing with complex cyber-physical systems, such as modularity, time constraints, and data manipulation. Such simple models encode data as a number of tokens. For instance, the fuel level is represented by a certain amount of tokens within the `fuel` place.

The basic Petri nets we introduced so far are commonly referred to as *Place/Transition* (P/T) nets. As we have seen, they are well-suited for the modelling of process control, synchronisation and resource flow within a system. However, in more complex situations the information used might not be easily representable as simple, black tokens. We can easily imagine cyber-physical systems where the transmission of more complex information is required. A P/T net modelling a very simple drone controller may have transitions `up`, `down`, `turn left`, `turn right`, `move forward` and `move backward`, which all modify the state. However, if we now add possibilities to perform multiple actions at the same time, we would have to add more transitions such as `move up & forwards`. The number of transitions within this relatively simple system would grow exponentially.

It would be more efficient to represent such information directly in the tokens. In order to do so, we need a token for vertical movement that could express to either move up, move down or stay, another one for changing directions (left, right, stay) and a third one for the forward/backward movement. We can then send these three tokens into a transition that would perform the action depending on the token values.

This simple scenario shows the necessity for *High-level Petri nets* (HLPN). HLPN were developed with the goal to attach a values to tokens, such as described before. Transitions and arcs are extended with *guards*, which evaluate the token values and assert a certain configuration.

7.5.1 Drone Controller

Our new High-level Petri net drone controller is presented in Figure 7.11 On the first view we can see three distinct groupings for `vertical movement`, `horizontal movement` and `orientation`. Each one of these three groups contains one place with an initial token with value `stay`. The behaviour of the groups follows the same scheme. From the initial place we can use transitions to create tokens with different values. The transitions' preconditions are guarded. This means that if the token's value is `stay`, we can only fire transitions whose guards require a `stay`-token (there are two in each group). Once we fire the transition, the `stay`-token is consumed and a new one is produced. For example from the place `altitude` we can either fire a transition `move up`, which produces an `up`-token, or `move down`, that creates a `down`-token. This behaviour matches the pushing of a joystick on the drone's physical remote to either direction. There are two more transitions within this group that are guarded by `up` and `down`, respectively. Firing these transitions will consume the token respective token and produce a `stay` token. In other words, we can use them to "reset" the token (i.e. to stop movement into that direction). On an actual drone remote, this behaviour matches the releasing of the drone's altitude joystick to neutral position. The groups that express the horizontal movement and orientation behave correspondingly.

Next to these three groups we can see the central `move` transition. This transition is responsible for the actual displacement of the drone `Move` consumes the tokens from `vertical`, `horizontal` and `orientation`. Interestingly, the arcs for these tokens are not guarded by concrete token values, but by variables (`a`, `d` and `m`). These variables are assigned when `move` fired and take the tokens' values. The transition further uses a token from place `drone` and a token from place `battery`. Note, that both `drone` and `battery` have different token types (i.e. can hold different valuations): While battery contains multiple, classical black tokens, the drone-place only holds one data-record token whose value stores information about the drone's current *position* (as $\langle x, y, z \rangle$-coordinates) and *angle* (stored as anti-clockwise deviation angle from north[2]).

Firing `move` will consume the three movement indicator tokens, one battery-token and the drone-token (assigned to variable `s`). It then produces equivalent tokens to the consumed ones in each one of the movement places, plus a new token in the `drone`-place, whose value is updated to match the new state of the drone[3]. Producing equivalent movement tokens to the one that has been consumed, has the effect that the behaviour of the drone will remain continuous until it is actively altered. In the example, `move`'s consumption of an `up` token will produce another `up` in the `altitude`-place. Hence, when firing `move` again, the drone will continue rising, unless the token has been modified (using the transitions within the `altitude` group)

Note, that since there is no way to produce tokens in the `battery`-place, this is the resource that limits the number of times we can trigger the central transition and hence move the drone.

Our drone controller further has a safety-mechanism integrated. This mechanism is shown as a guard on the central transition and expresses that we cannot fire this transition if the drone's position z-position is lower than 50 (centimetres) and the altitude token is specifies a downward movement (i.e. has value `down`).

7.5.2 Formalising High-level Petri nets

As we saw in the example, HLPN use various additional concepts such as variables and expressions. In this section we will focus on the adaptation of our existing Petri net formalism to integrate these concepts.

Variables and expressions To formally express the concepts that were used in the example above, we need to define matching and filtering of tokens. For this reason we introduce the notions of variables and expressions: We define a set of variables V (e.g. s,a,d,m), and a set of expressions over data and variables (noted E).

[2] I.e. the value 90 describes West, 180 South, and 270 and -90 stand for East

[3] In the figure the update is simply expressed as $s + f(a, d, m)$, where we assume the existence of a function f which can provide the $\langle x, y, z \rangle$-coordinate and orientation angle change produced depending on the group tokens.

Since variables are expressions or parts thereof, V is a subset of E: $V \subseteq E$. We further define a function $variables : E \to \mathcal{P}(V)$, which returns the set of variables that are used by an expression.

If an expression does not contain any variables, we call it a *ground expression*. The set of ground expressions $E^{\varnothing}$ is defined as $E^{\varnothing} = \{e \mid e \in E \wedge variables(e) = \varnothing\}$. In order to express concrete values (such as `stay` or `up`) this set has to be non-empty.

Binding Equipped with these tools we can now express the binding of variables (such as binding of the `altitude` token to the variable `a`). A binding $\sigma \in \Sigma$ is a partial function from variables to ground expressions: $\sigma : V \to E^{\varnothing} \cup \{\bot\}$. We remind ourselves that a substitution, as introduced in Section 7.2.4, is denoted noted $\sigma(e)$. This means that it is the application of a binding σ on an expression $e \in E$ and replaces variables with their value in the binding. Note, that a substitution does not have to replace all variables within an expression. Variables that do not appear in the binding remain in the expression. Formally:

$\forall e \in E, \exists e' \in E^{\varnothing}, e' = \sigma(e) \wedge variables(e') = variables(e) \setminus \{v \mid v \in V \wedge \sigma(v) \neq \bot\}$.

Transition guards In high-level Petri nets transition guards are used to prevent transitions to fire with unwanted token values and thereby stop unwanted behaviour. In the drone controller we use the guard $\neg(\mathtt{a} = down \wedge \mathtt{s}.position.x < 50)$ to prevent the drone from moving down when its altitude is below a certain threshold. Formally, transition guards are functions that evaluate to Boolean values ($\mathbb{B} = \{\top, \bot\}$). Transitions can only be fired iff the guard evaluates to $\top$.

Integration of the concepts Using the above definitions, we modify the definition of a Petri net to incorporate the new concepts. Specifically we change the following:

1. Add ground expressions to tokens within markings as a means to hold data
2. Add (variable) expressions to arcs in order to filter tokens or bind variables
3. Add Boolean guards over expressions to allow filtering of tokens and disabling transitions for certain token-values.

Formally, HLPN are described as tuples based on the structure of P/T nets (as defined before), but with added and modified fields: $\langle V, E, variables, P, T, pre, post, guard, m_0 \rangle$, where:

- $V, E, variables$ are the sets of variables and expressions, and the $variables$ function as introduced above;
- $M = P \to \mathcal{M}(E)$ is the set of all possible multisets over expressions. It replaces the former marking definition ($P \to \mathbb{N}$);
- $guard : T \times \Sigma \to \mathbb{B}$ defines the allowed bindings for each transition;
- $m_0 : P \to \mathcal{M}(E^{\varnothing})$ associates to each place a multiset of ground expressions as the initial marking.

This definition differs resembles the P/T definition from before, except for the introduction of variables and expressions and the new definition of marking, where a marking consists of a multiset of expressions rather than a natural number.

In fact, we can look at a classical P/T net is a special kind of high-level Petri net where the set of variables is empty ($V = \varnothing$) and only one expression ($E = \{\bullet\}$) exists. Note that since transition guards are usually defined as Boolean expressions over expression comparisons, a P/T net's $guard$ function always returns true: $\forall t \in T, \forall \sigma \in \Sigma, guard(t, \sigma) = \top$.

The semantics of a transition t is given in Equation (7.13). It differs from the previous transition semantics given in Equation (7.2) by adding the substitution of a binding within the pre and $post$ functions.

$$transition_t : \frac{\exists \sigma, \sigma(pre(t)) \leqslant m, guard(t, \sigma) = \top}{m \xrightarrow{t} m - \sigma(pre(t)) + \sigma(post(t))} \tag{7.13}$$

The use of expressions in HLPN requires the extension of the substitution to include all expressions in the function image. Formally, $\forall m \in M, \forall p \in P, \sigma(m)(p) = \sigma(m(p))$. This substitution also has to be extended to multisets, since HLPN markings are defined as such. Applying a substitution on a multiset is performed by applying the substitution to each element: $\forall es \in \mathcal{M}(E), \sigma(es) = [\sigma(e) \mid e \in es]$, where $[\ ]$ denotes a multiset by intention.

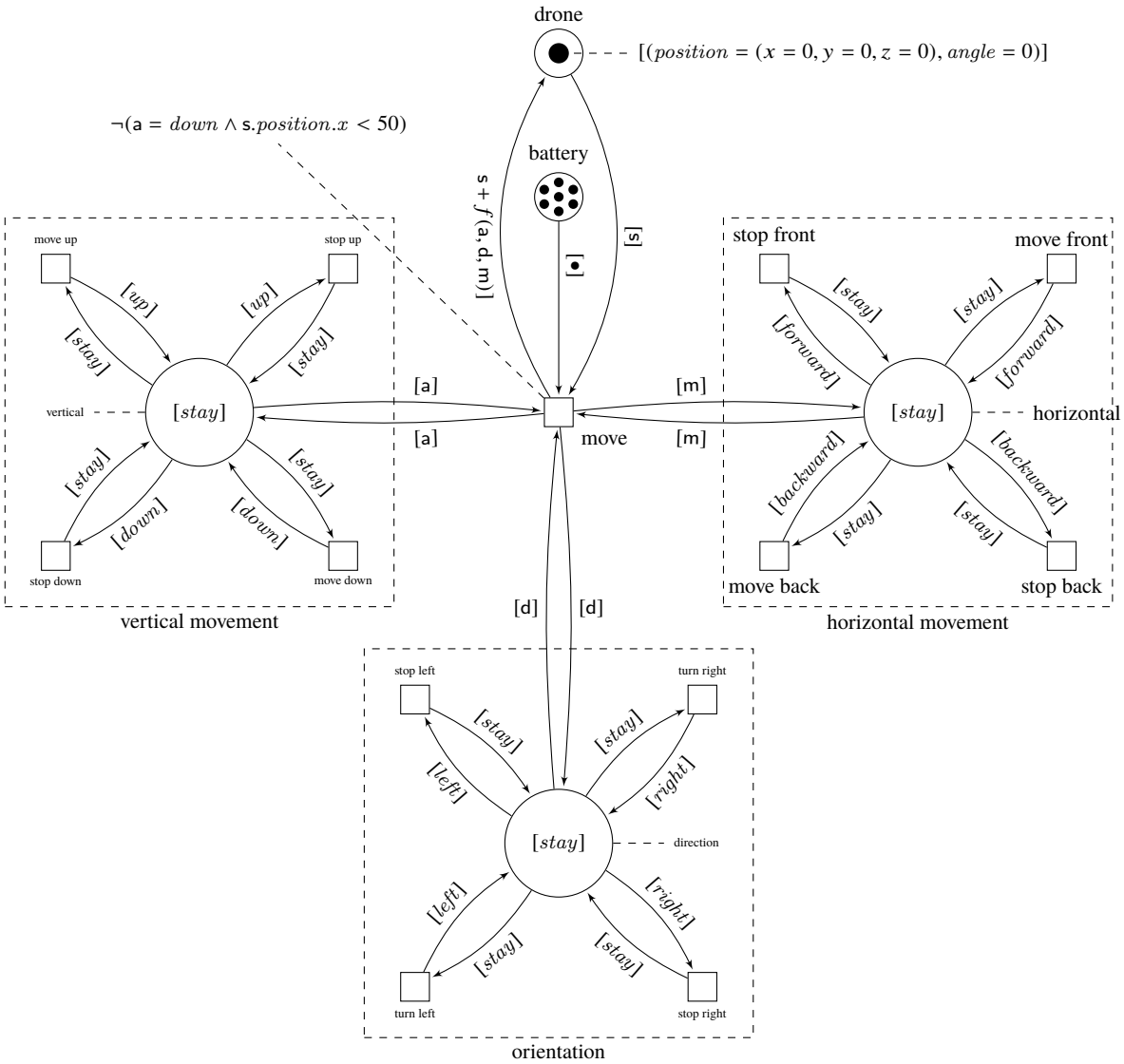


Fig. 7.11: Petri net representing the control of a drone, using data as tokens and arc valuations. The control is composed of three modules, that represent the state of the controller joysticks: one to move upwards or downwards, one to rotate left or right, one to move forwards or backwards. The drone moves when the $move$ transition is fired. It captures the state of the controller to move the drone. Places are labelled with multisets of tokens, shown as $[token_1, \ldots token_n]$. Each token is a structured data, that can be a simple token ($\bullet$), an atom ($stay$) or a record or subdata with named fields, such as $(position = (x = 0, y = 0, z = 0), angle = 0)$. The $position$ is a position relative to the initial position of the drone, and the $angle$ is an angle on the horizontal plane and signifies the deviation from north. Arcs are labelled by multisets of tokens, that can use variables, for instance x. Transitions have implicitly a guard that always returns $\top$, except the $move$ transition that has an explicit guard $\neg(\mathsf{a} = down \wedge \mathsf{s}.position.x < 50)$.

7.5.3 Other High-level nets

In the last couple of decades, researchers hae come up with numerous extensions and modifications of the P/T net formalism to ease the modelling of complex systems. This section introduces four representatives of such complex variants of Petri nets.

Coloured Petri nets (CPN) [158] CPN are an extension of Petri nets where each token can be of a certain *colour*. Each colour is a set of values, comparable to a type definition. The set of colours has to be clearly specified for a CPN. Arcs are modified to allow the specification of the colour to use in pre- and postconditions.

The drone example in Figure 7.11 is a CPN with the following colours:

- $CVertical = \{up, down, stay\}$
- $CHorizontal = \{forward, backward, stay\}$
- $CDirection = \{left, right, stay\}$
- $CFuel = \{\bullet\}$

Unfortunately, not all data types can be easily represented as colours, for instance, the drone state is a dictionary or tuple. The corresponding colour contains all the possible combinations of positions and angles. Since this set is infinite (x, y, z and $angle$ are rational), this solution is not usable in practice, and the domains have to be made discrete and bounded.

Symmetric nets [71] Symmetric nets, formerly known as "well-formed Petri nets", are a special kind of Colored Petri nets that use only simple data types: tuples of constants in finite and ordered domains. The data structure is thus very limited, and cannot easily represent data of varying size, as well as data in *a priori* unbound domains. Operations on data are also very restricted: it is only possible to obtain the successor or predecessor of a value, and test equality between two values. There are no operations allowed on the tuples themselves.

Despite all these limitations, symmetric nets are successfully used in practical problems because they offer a formalism that is convenient for efficient structural analysis and model checking. The limited expressiveness of these types increases the number of properties that can be verified, in particular for the structural analysis of models, such as computation of bounds or invariants.

Algebraic Petri nets (APN) [297] Algebraic Petri nets are a special form of Colored Petri nets, that allow the use of abstract algebraic data types [87] as colours. Such algebraic data types consist of a signature and axiomatisation and hence allow the user to represent custom data types. The advantage of APNs is that every data type used has a precise axiomatisation and consequently proofs can be done by theorem proving without the usual limitation of finiteness of model checking. Using APNs, parametric and under-specified systems can be modelled and also be verified in a more systematic way.

Timed Petri nets [231] The concepts of time and of Petri nets are quite opposite: while time determines the occurrences of events in a system, Petri nets consider only their causal relationships. Several variants of Petri nets have been defined with the notion of time. The three most common are: Time Petri nets, Timed Petri nets, and Petri nets with Time Windows. In Time Petri nets, transitions are labelled with time intervals, that define the time at which the transition can be fired, after it has been enabled (has all its preconditions met). In Timed Petri nets, time is also put within tokens, that have thus an age, and transitions are labelled with time intervals that define the age at which tokens can be consumed. In Petri nets with Time Windows, transition are given time intervals, this means that transition can fire (not mandatory) only in this time interval.

In this chapter we do not consider time with Petri nets for simplicity, and only focus on the causal relations associated to Petri nets.

7.6 Combining Model Semantics and Simulation

The recent rise of computers in everyday life is especially of importance when their purpose is to react to and act upon environment changes. We refer to such systems as cyber-physical systems (CPS). Such systems consist of a software part (e.g. a controller program) and a hardware side that usually consists of sensors and actuators.

While smaller systems, such as heating/light systems that measure presence have been installed and used for a long time, the trend towards Internet-of-Things applications, "smart systems" (such as new-generation cars and trains), and general large-scale systems that include hundreds, sometimes even thousands of components drives the need for means to verify and validate such systems.

The problem of these highly heterogeneous systems lies in finding the right means to model each part of the system. While former approaches to find *the one* modelling language or tool failed, nowadays the trend is reversed. Modern research is looking to model every part of the system with its most appropriate modelling formalism. Subsequently the individual components are combined and simulated together. This approach, called

co-simulation [126] has shown promising research results, but comes with one important question: How should we combine models that were developed using different syntaxes and semantics?

One of the approaches that aims to answer this question are *Functional Mock-up Units* (FMUs) [36]. The FMU formalism provides a homogeneous interface, the Functional Mock-up Interface (FMI). Each component is wrapped inside its own FMU and is henceforth executed using the FMI. The individual FMUs' inputs and outputs are connected to one another in order to allow information to be transmitted within a system. The semantics of such a composition are dominated by a so-called *master algorithm* that is responsible for passing control to individual components, relaying signals and choosing appropriate time step sizes which suit all units in the system.

In this section we introduce Petri Net Functional Mock-up Units (PNFMUs) [180]. This new type of FMU wraps a Petri net within the FMU in order to provide access to the efficient evaluation and calculations we introduced in the previous sections. Using Petri nets it is possible to detect deadlocks and possible system evolutions.

The FMI standard strictly defines the qualities of a valid FMU. In order to comply with this standard however, it is necessary that PNFMUs overcome three major challenges:

1. Time evolution Similarly to a Mealy or Moore machine, the Petri net formalism doesn't explicitly define a time concept. Often, the firing of a transition is accepted as a time unit. To overcome this limitation, the periodic or aperiodic wrappers presented in [272] can be used for this purpose. In addition, these wrappers has also been discussed in [83] when considering the semantic adaptation, giving the possibility to automatically generate FMUs from a domain specific language that solve this problem.

2. Inputs/Outputs adaptation Obviously, the inputs and outputs type and nature of a given formalism can differ from the simple value affectation of a variable defined by the FMI standard. Therefore, an adaptation must be performed as well for inputs and outputs of a FMU. The authors of [83] provided solutions with their domain specific language to overcome this limitation too.

3. Non determinism In the standard, FMI API functions are mathematically modelled as total functions [272]. This means that calling an FMU's API with the same parameters should always yield the same result. An initial, yet naive, approach to represent a Petri net state would be to consider it as a single marking. However, in situations where a Petri net has more than one fireable transitions at a given state, the evolution function of the FMI API (*doStep*) cannot yield a deterministic result. This particularity is clearly shown by the Petri net and its reachability graph in Figure 7.12. In this example, either t_1 or t_2 can be fired from its initial marking, leading to different system evolutions. This issue must be addressed to be able to represent a Petri net within an FMU.

Since the two first subjects of interest have been extensively discussed in the referenced publications, the non determinism of the formalism is considered here. Now that the problems related to the consideration of formalisms as FMUs has been addressed, let's discuss now of the PNFMU formalisation.

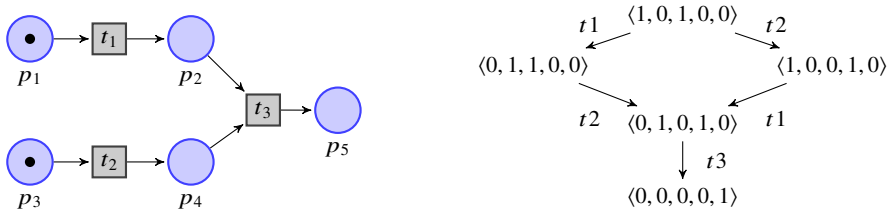


Fig. 7.12: A simple Petri net and its reachability graph. The markings are encoded as $\langle p_1, p_2, \ldots, p_n\rangle$, where p_i are the number of tokens within the places, which the ordered by their index. $\langle 1,0,1,0,0\rangle$ encodes the initial marking on the left, stating that there is one token in the first and third place each (i.e. p_1 and p_3).

7.6.1 PNFMU Formalisation

The need for Petri nets within an FMI system is closely tied to the need to analyse system evolutions and reachability of possible states. In order to do so, we need to define the formal basis of an PNFMU, which we base on the FMU formalisations presented in [272] and [51].

A standard FMU is defined as the structure $F = \langle S, U, Y, D, s_0, set, get, doStep \rangle$. Given a Petri net PN = $\langle P, T, pre, post, m_0 \rangle$, we define that a PNFMU is a tuple $PNFMU = \langle S, U, Y, D, s_0, set_{\mathbb{N}}, get_{\mathbb{N}}, doStep_{\mathbb{N}} \rangle$.

A PNFMU's internal states S are all possible markings of the Petri net, not just the reachable ones (i.e. $reach(s_0) \subseteq S$). U and Y are the Petri net's places which are writeable and readable, respectively. D remains the dependency of outputs on inputs and is used to avoid circular dependencies when composing FMUs. s_0 is set to the initial marking m_0.

The biggest difference between FMU and PNFMU is that the former only operates on one state. On the contrary, PNFMU are designed to explore and operate on a Petri net's reachability graph. This change is reflected in the three functions set, get and $doStep$, which are adapted to perform operate on sets of states as follows:

$set : \mathcal{P}(S) \times U \times \mathcal{P}(\mathbb{N}) \to \mathcal{P}(S)$ operates on a set of states (markings), modifies the marking of a certain place in each marking and returns a set of new states. Intuitively, the function iterates over the values to set and modifies returns a new state for each state that is modified. In total the function returns $n \times m$ markings, where n is the count of states and m the number of values entered into this function. Using the reachability graph from above, the call $set(\{\langle 0,1,1,0,0\rangle, \langle 1,0,0,1,0\rangle\}, p_5, \{2,4\})$ returns the set $\{\langle 0,1,1,0,2\rangle, \langle 1,0,0,1,2\rangle\}, \{\langle 0,1,1,0,4\rangle, \langle 1,0,0,1,4\rangle\}$. Note, that none of the four returned markings is reachable from s_0 by transitions.

$get : \mathcal{P}(S) \times Y \to \mathcal{P}(\mathbb{N})$ recuperates the set of place-markings of a set of states.
For example, $get(\{\langle 0,1,1,0,0\rangle, \langle 1,0,0,1,0\rangle\}, p_3)$ returns $\{1,0\}$, while $get(\{\langle 0,1,1,0,0\rangle, \langle 1,0,0,1,0\rangle\}, p_5) = \{0\}$.

$doStep : \mathcal{P}(S) \times \mathbb{N} \to \mathcal{P}(S) \times \mathbb{N}$ executes system evolutions. Given a set of states and a time step $h \in \mathbb{N}$, $doStep$ returns the length of the longest sequence h' (with $0 \leq h' \leq h$) that can be executed and the states that are reached. Note, that $doStep$ only operates states that are reachable from s_0, any states $s \notin S$ are not considered. Furthermore, contrary to standard FMUs, PNFMU's $doStep$ is defined over natural numbers and fails if $h \notin \mathbb{N}$. $doStep$ is defined as follows:

$$doStep(s, 0) = (s, 0) \tag{7.14}$$

$$doStep(s, 1) = \begin{cases} (succ, 1) & \text{s.t.} \quad succ = \left\{ s' | \exists t \in T, \exists s_i \in \left(s \cap SS(T, s_0) \right), s_i \xrightarrow{t} s' \right\} \wedge succ \neq \emptyset \\ (s, 0) & \text{otherwise.} \end{cases} \tag{7.15}$$

For $h \geq 2$: (7.16)

$$doStep(s, h) = \begin{cases} (s, 0) \text{ st. } (s, 0) = doStep(s, 1) \\ (s'', h'' + 1) \text{ st. } (s', 1) = doStep(s, 1) \wedge (s'', h'') = doStep(s', h - 1) \end{cases} \tag{7.17}$$

The $doStep$ function for the example Petri net net above is given by the equations above. To show its execution, here is the first four steps of $doStep$ for pn':

- $doStep(\{\langle 1,0,1,0,0\rangle\}, 1) = (\{\langle 0,1,1,0,0\rangle \langle 1,0,0,1,0\rangle\}, 1)$;
- $doStep(\{\langle 1,0,1,0,0\rangle\}, 2) = (\{\langle 0,1,0,1,0\rangle\}, 2)$;
- $doStep(\{\langle 1,0,1,0,0\rangle\}, 4) = (\{\langle 0,0,0,0,1\rangle\}, 3)$.

Table 7.1 compares the definitions of the individual components of both, the FMU and PNFMU.

7.6.2 PNFMU Example

Figure 7.13 shows some possible evolutions of a PNFMU that wraps the Petri net of the above example. Out of the infinite sequences of actions possible, we choose five traces that are being presented as an evolution tree.

- First, the main branch (center) shows the $doStep$ evolution of the system. Note that after the third $doStep$ the returned state remains unchanged and the returned h' is 0. This indicates a deadlock, as no further evolution is possible.

Component	FMU	PNFMU
S	a set of internal states of F	the set of all possible markings; S
U	a set of input variables over values $\mathbb{V}$	a set of input places; $U \subseteq P$, $\mathbb{V} = \mathbb{N}$
Y	a set of output variables over values $\mathbb{V}$	a set of output places; $Y \subseteq P$, $\mathbb{V} = \mathbb{N}$
$D \subseteq U \times Y$	a set of input-output dependencies specifying which outputs depend on which inputs	
$s_0 \in S$	the initial state of F	the initial marking of PN; $s_0 = m_0$
set	sets the value of an input variable, returns the new state; $set : S \times U \times \mathbb{V} \to S$	sets the value of an input place in the given states, returns the new states; $set : \mathcal{P}(S) \times U \times \mathcal{P}(\mathbb{N}) \to \mathcal{P}(S)$
get	returns the value of an output variable; $get : S \times Y \to \mathbb{V}$	returns the an output place's values in all given markings; $get : \mathcal{P}(S) \times Y \to \mathcal{P}(\mathbb{N})$
$doStep$	attempts simulation step, returns actual step size and new state; $doStep : S \times \mathbb{R}_{\geq 0} \to S \times \mathbb{R}_{\geq 0}$	attempts simulation step, returns actual step size and new states; $doStep : \mathcal{P}(S) \times \mathbb{N} \to \mathcal{P}(S) \times \mathbb{N}$

Table 7.1: Comparison of the structures of FMU and PNFMU.

- We observe another trace using only *doStep* actions. The $doStep(s, 4)$ attempts to find the states reachable with a sequence lenght of 3. However, the reachability graph dictates that only two steps are possible. Hence the action returns the end of the longest sequence that has been reached (in this case 1) and
- On the left branch, we see the creation of two markings of which one is not in the reachability graph of m_0. The subsequent *doStep* therefore ignores this state when performing the calculation.
- The right branch describes the evolution after a *set* action on the initial state. This *set* creates one single, non-reachable marking. Therefore, the *doStep* action on this state returns the empty set of states.
- Executing another *set* creates a reachable marking. This means that the *doStep* is possible and succeeds.

We can easily see that, using PNFMUs, it is possible to study the possible evolutions of a system and analyse whether the execution of certain sequences leads to a reachable marking (i.e. a "good state"). We can also find deadlocks situations, i.e. states where a *doStep*) returns 0.

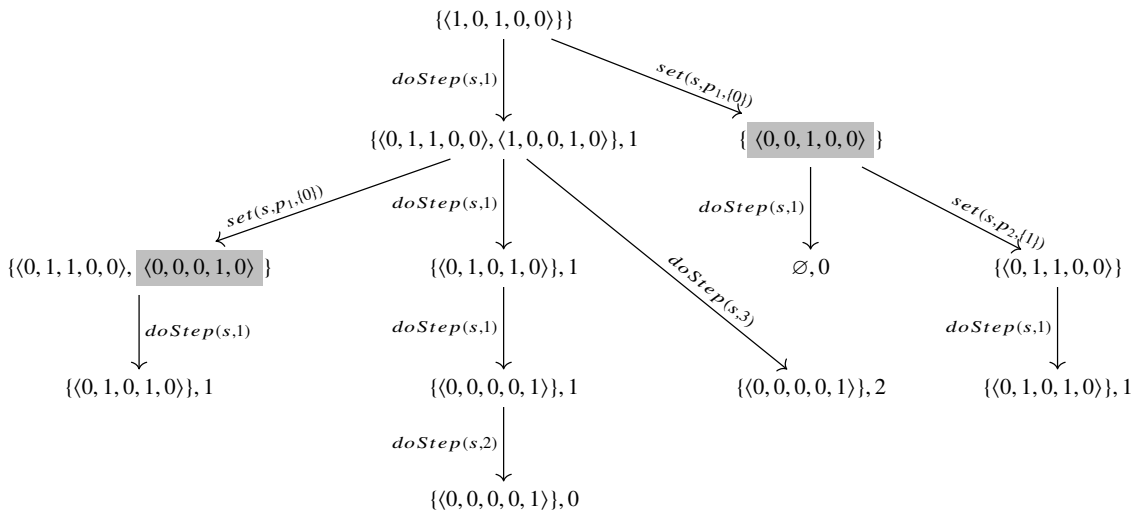


Fig. 7.13: A tree displaying possible system evolutions. Nodes are the (sets of) states and arcs are annotated with the action performed (where s is the variable that stores the previous states). Markings that are not in the reachability graph of s_0 are gray.

7.6.3 PNFMU Composition

The FMI standard defines FMU composition in a very simple manner. An FMU's outputs are directly connected to another FMU's inputs. This is usually performed by the *master algorithm* using the *set* method. Generally, the master algorithm performs such updates in two steps: 1. update all FMUs' input values; 2. perform system evolution by calling *doStep* on each FMU.

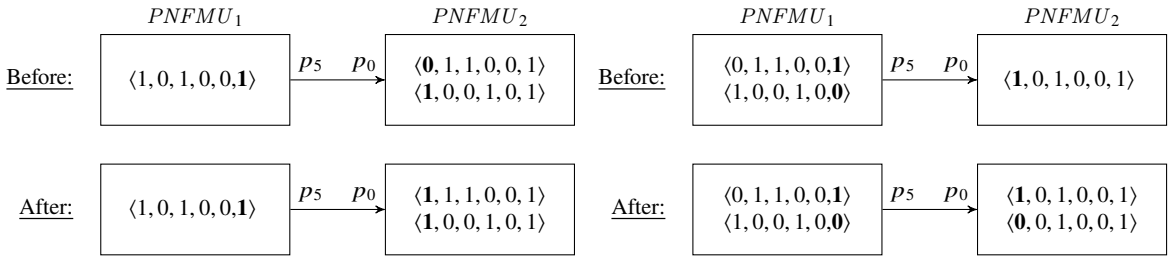
The composition of PNFMUs is slightly more complex, as it is necessary to handle sets of states. In general, three main scenarios can be distinguished:

One-to-One/One-to-Many Connecting a PNFMU that is currently in one, single state to another PNFMU is most trivial of the possibilities. Upon update, the marking(s) of the second PNFMU's inputs are set to the first PNFMU's output places' values. If the second PNFMU has multiple states, all states are updated.

As represented in Figure 7.14a the place p_5 of $PNFMU_1$ is linked to the place p_0 of $PNFMU_2$, drawn in bold. The execution scenario is quite trivial. In fact, for every marking of $PNFMU_2$, the number of tokens of place p_0 is set to the number of tokens of the place p_5 from the $PNFMU_1$.

Many-to-One The case represented Figure 7.14b is slightly more complicated. $PNFMU_1$, which has multiple markings, is connected to $PNFMU_2$ that has one single marking. In other words, there is two distinct numbers of tokens in the place p_5 in $PNFMU_1$, respectively 0 and 1. In consequence, the variables update step should create one marking for each values of $PNFMU_1$'s output variable. In the current case, $PNFMU_2$ should have two markings after the update, one for $M(p_1) = 0$ and another for $M(p_1) = 1$.

Many-to-Many After a step evolution, it is possible that both PNFMUs contain more than one marking. In fact, the *Many-to-Many* relationship is a generalisation of the *Many-to-One* case. This case requires to write all output values of one PNFMU to all inputs of the other. Effectively, the Cartesian product of possible states is created, where all combinations of $PNFMU_1$'s values are applied to all input places of $PNFMU_2$.



(a) One-to-One/One-to-Many relationship (b) Many-to-One relationship

Fig. 7.14: FMU relationships before and after the update

7.6.4 Advanced Composition Mechanisms

Modularity is a mandatory principle to apply Petri nets to real world-sized systems. Modular extensions of Petri nets allow to create complex models by combining smaller entities. They facilitate the modelling and verification of large systems by applying a divide and conquer approach and promoting reuse. Modularity includes a wide range of notions such as encapsulation, hierarchy and instantiation. Over the years, Petri nets have been extended to include these mechanisms in many different ways. The heterogeneity of such extensions and their definitions makes it difficult to reason about their common features at a general level. An approach has been proposed to standardise the semantics of modular Petri nets formalisms, with the objective of gathering even the most complex modular features from the literature. This is achieved with a new Petri nets formalism, called the *LLAMAS* Language for Advanced Modular Algebraic Nets (LLAMAS)[197]. This framework can be envisioned to extend current work on $PNFMU$ and for the abstract description of the master algorithm.

7.7 Tooling

The vast amount of research invested in the Petri nets domain not only led to many important, theoretical findings but also saw the development of tools and software that can be used to model, analyse and simulate Petri nets. In the following section we will describe some tools that can be used to develop models based on Petri nets and to perform model checking thereon. Some of the tools described below are currently unavailable or are not able to perform model checking for Petri nets directly. Nevertheless, we will introduce them since they contribute interesting concepts and serve as potential candidates for extension to the formalism. Note, that since the domain is subject of intensive research, new ideas are often presented in in academic tools which oftentimes remain proof-of-concept implementations and not advanced into mature, reliable software.

7.7.1 Tools for Petri net Modelling and Verification

Since the Petri net formalism is based on a graphical syntax, matching model editors, composers and visualisers are an important important part of the Petri nets tool chain. Over the years, a lot of tools have been implemented to perform these tasks, but only a few have survived long enough to be well-recognised by the Petri nets community, or to be used in industry.

CPN Tools [206] (http://cpntools.org) is a tool for editing, simulating, and analysing high-level Petri nets. It supports basic Petri nets, timed Petri nets and Coloured Petri nets. It features a graphical editor, a simulator and includes a state space analysis component.

CPN-AMI [132] (http://move.lip6.fr/software/CPNAMI) is a formal modelling platform based on Petri nets, more specifically targeting Symmetric nets. It provides a graphical user interface to create and edit models, run tools built upon the platform and obtain their results. The CPN-AMI platform uses tools developed by several research teams to perform structural and behavioural analysis of models.

The platform features software to graphically create Petri nets modularly or using a scripting interface. Other tools allow to compute structural properties, such as bounds, invariants, syphons, traps, and liveness. The platform is also able to generate the state space of a Petri net, and perform CTL and LTL verification using various model checkers.

ITS-Tools ITS-TOOLS [267] (http://ddd.lip6.fr/) is a successor of CPN-AMI. It allows its users to create models using a textual language or a graphical editor for Petri nets. This tool is able to perform behavioural analysis (safety, CTL and LTL model checking) on models expressed in Place/Transition, Symmetric and Time Petri nets, as well as some other formalisms.

AlPiNa & StrataGEM: Algebraic Data Types and Term Rewriting tools In recent years, the University of Geneva's SMV group has produced two tools for model checking and model editing: the Algebraic Petri Nets Analyzer (AlpiNa) [56, 57, 151] and StrataGEM [4] [190, 39]. AlpiNa is a model checker dedicated to Algebraic Petri Nets. StrataGEM marries the concepts of Term Rewriting to the efficiency of Decision Diagrams, in order to perform efficient model checking. While AlPiNA is adapted to high-level specifications using Petri nets, StrataGEM focuses on the low-level ones.

These two tools heavily rely on algebraic data types and term rewriting techniques to represent systems and their semantics. They are able to compute large state spaces, and to evaluate reachability properties, as proven in the Model Checking Contest [171].

SNAKES [230] (https://snakes.ibisc.univ-evry.fr) is a Python library that provides a framework to define and execute many sorts of Coloured Petri nets. A key feature of SNAKES is the ability to use arbitrary Python objects as tokens and standard Python expressions in many points, for instance in transitions guards or arcs.

Renew: Renew [62] (http://www.renew.de) is a tool that supports the development and execution of object-oriented Petri nets, a specific kind of Coloured Petri nets. Its main feature is its integration with the Java programming language: Petri nets can be labelled by Java code, and thus call Java methods in transition guards or on arcs.

Petri net kernel (ePNK): [164] (http://www.imm.dtu.dk/~ekki/projects/ePNK/) is a platform for developing Petri net tools based on the PNML [147] transfer format. Its main idea is to support the definition of

[4] https://github.com/mundacho/stratagem

Petri net types, which can be easily integrated into the tool, and to provide a simple, generic graphical editor, which can be used for graphically editing nets of any plugged in type.

Tina [29] (http://www.laas.fr/tina) The TIme Petri Net Analyser is a toolbox for the editing and analysis of Petri nets, and Time Petri nets. It features a graphical editor, and a set of tools to perform structural analysis (e.g. invariants), behavioural analysis (e.g. reachability and coverability graphs) and LTL model checking.

TAPAAL [157] (http://www.tapaal.net/) is a tool for the modelling, simulation and verification of Timed Petri nets. It offers a graphical editor for drawing Petri nets, a simulator for experimenting with the designed nets, and is able to check the bound of the model, and to verify properties expressed in a subset of CTL. TAPAAL can translate its models to the format of the UPPAAL tool. In the same domain, Open-Kronos [298] uses the Büchi automata approach to verify real-time systems.

The Petri nets repository [146] (http://pnrepository.lip6.fr) is a recently created repository of Petri net models. It includes models imported from several sources, such as the Model Checking Contest [170, 169], the former PetriWeb repository [127], and the Very Large Petri nets benchmark suite (http://cadp.inria.fr/resources/vlpn/). Its main feature is that this repository provides access to models and corresponding, computed properties through both, a web interface and an API.

Many other tools exist to perform model checking on systems specified with formalisms that are not Petri nets, such as for instance automata or communicating processes. Among them, the following tools are worth mentioning:

SPIN [149] (http://spinroot.com) is a software model checker that verifies specifications written in PROMELA (PROcess MEta LAnguage), adapted to the representation of asynchronous distributed systems. The tool uses the Büchi automata approach to verify LTL properties on the models.

SPOT [192] (https://spot.lrde.epita.fr) is a Büchi automata library rather than a full model checker. It is intended to be coupled with an engine able to compute the transition relation of the system, in order to build a LTL model-checker. This design allows it to be used with any kind of formalism. This library implements great number of useful algorithms related to LTL model checking: LTL parsers, LTL formulae syntactical simplification, translation to several flavours of Büchi automata, automata simplification, and of course emptiness check algorithms for them. It is considered one of the best candidates for operational model checking [245].

UPPAAL [27] (http://www.uppaal.com) UPPAAL is an integrated tool environment for modelling, simulation and, verification of real-time embedded systems. Typical application areas of UPPAAL includes real-time controllers and communication protocols in particular, those where timing aspects are critical.

SCADE [265] (http://www.esterel-technologies.com) is an industrial-grade environment for the development of critical embedded systems. It is coupled with a model checker and used in industry in the domain of synchronous systems.

Note that this list is non-exhaustive and the provision of a complete list is out of the scope of this chapter. Its purpose is to provide a short overview over some of the more popular tools to help the decision of an appropriate one. The choice of a tool should however depend on various criteria, such as the Petri net class (P/T net, Coloured Petri nets, Timed Petri net), the type of properties that should be tested by the tool, the required efficiency of the tool and the execution environment. We encourage the reader to consult other resources[5] to find additional guidance towards a better informed decision process.

7.7.2 Evaluation of Model Checking Techniques

The Petri nets community is eagerly hosting and participating in various model checking contests in order to evaluate the various model checking techniques and tools and to discover approaches that might be particularly well-suited for certain model classes and types. Some of the more prominent ones are following, but note that this list is non-exhaustive.

The Hardware Model Checking Contest [34] was first held in 2007, and is now associated with the CAV (Computer Aided Verification) and FLOC (Federated Logic Conference) conferences. It focuses on circuit verification by means of model checking based on SAT-solvers. This event ranks the three best tools according to a selected benchmark.

[5] such as https://en.wikipedia.org/wiki/Model_checking

The Verified Software Competition [165] takes place within the Verified Software: Theories, Tools and Experiments (VSTTE) conference. This competition is a forum where researchers can demonstrate the strengths of their tools through the resolution of five problems. The main objective of this event is to evaluate the efficiency of theorem proving tools against SAT-solving. Started in 2010, it has now become a yearly event.

The Competition on Software Verification [4] is an event associated with the conference on Tools and Algorithms for the Construction and Analysis of Software (TACAS). Aimed at the verification of safety properties of C programs, it has been held yearly since 2012.

The Satisfiability Modulo Theories Competition [54] takes place within the context of the CAV conference. Held yearly since 2005, its objective is to evaluate the decision procedures for checking the satisfiability of logical formulas.

The SAT Competitions [159] proposes to evaluate the performance of SAT solvers. Initiated in 2002, it is held every two years since 2007, and identifies new challenging benchmarks at each edition.

The Model Checking Contest [169] at the Petri Nets conference puts emphasis on the specification of parallel and distributed systems, and their qualitative analysis.

The main problem of these contests is the difficult choice of comparison metrics. It is seemingly easy to compare tools for a set of models. Drawing conclusions from the achieved tool results in order to compare of model checking *techniques* is a much more complex endeavour, as each tool merges several techniques to be efficient. Moreover, the efficiency of model checking techniques highly depends on each individual model's characteristics. For instance, partial order reductions are showing good results for highly parallel systems with few synchronisations, but are less efficient when the degree of synchronisation increases. However, since verification techniques are often combined with other ones, it is difficult to clearly attribute good performance to an individual technique.

7.8 Summary

In this chapter we have shown a way to develop cyber-physical systems that include concurrent dimension. Our proposal is to consider to model first our system as many entities cooperating. Each entities can be modelled using the most convenient formalism to the problem that is considered. We propose for concurrent systems the use of Petri nets. We explain the interest and capabilities of Petri nets for modelling complex systems in an abstract way, i.e. by masking details that are not useful. Such modelling techniques hide concerns related to distribution, naming, communication mechanism and representation of data structures. We provide also their semantics through deduction systems in order to explain precisely all notions we are using such as states, marking and state space. It must be noted that the key aspect of the semantics of Petri net is the potential non-deterministic behaviours that we can exhibit from a model. The impact of this dimension is the potentially very large number of states of a system. We then develop some formal ideas to analyse Petri nets such as marking exploration and the use of invariants. While there is a complexity barrier for marking exploration, the use of invariants need human intervention.

In the rest of the chapter we have been interested into the idea of managing complexity by applying the *divide to conquer* principle. It means to consider separately the modelling and verification into small units. We also promote by this model decomposition the possible integration of heterogeneous modelling which is very important for the intrinsically heterogeneous cyber-physical systems.

The need for simulation of heterogeneous systems is also covered by our approach. It is based on the construction of functional units (FMU) and their cooperation into a larger system by specific master algorithms. Petri nets being a formalism often used to represent concurrent executions, defining its state as a set of markings was discussed and formally presented, avoiding the violation of the FMI API determinism. Furthermore, the API standard was extended with functions to add more observability and functionalities on the considered PNFMU. This study showed that deadlocks can be detected when simulating a PNFMU with the standard API, yet without being able to know the actual markings in which the Petri net is in a deadlock. Then, the composition of PNFMUs was introduced, yet partially leaving out of the study time evolution and outputs/inputs adaptation. Moreover, since the state of a PNFMU is now defined as a set of markings, a multi states relationship of the composition between FMUs was presented.

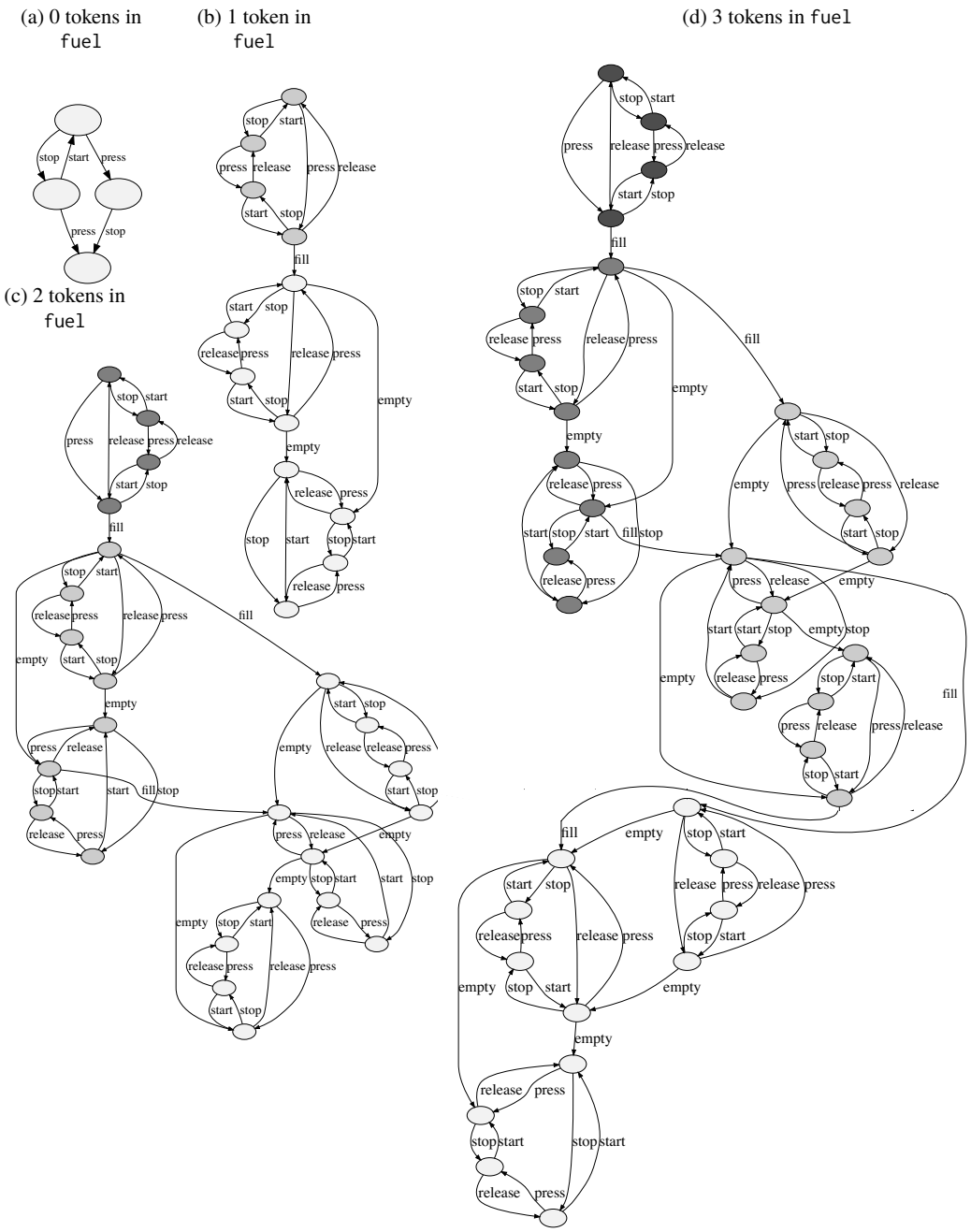


Fig. 7.15: The reachability graphs of the Petri net in Figure 7.1 with different initial markings. Note how the smaller graphs can be found as subgraphs inside the larger ones. This is due to the monotony of Petri nets as discussed above. Moreover, the node colour is representative of the number of tokens in place `fuel`, ranging from 0 (lightest) to 3 (darkest).

7.9 Literature and Further Reading

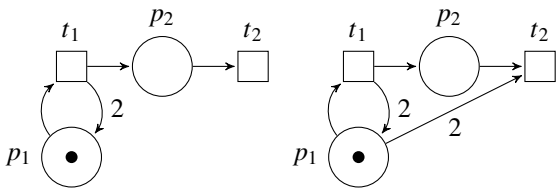
For a deeper understanding of concurrency modelling and Petri nets, we refer the interested reader to the following works. An excellent book by Reisig [239] is explaining fundamental analysis techniques for Petri nets. Through well-chosen examples, it also shows how to model several well-known systems that can be useful for beginners and newcomers to the domain of discrete dynamic system modelling and analysis. M. Diaz [85] also

provides a look at basic techniques for modelling with Petri nets. Several extensions of place transition nets are explained and their practical use for concrete applications detailed. This book will probably be of more interest for the curious Petri-netter who wants to dive into various modelling options such as stochastic Petri nets, Timed Petri nets and their corresponding formal techniques.

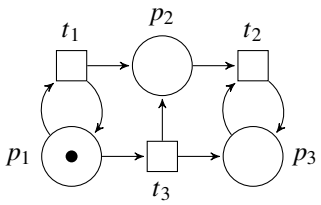
Finally, [26] is a reference for people who are interested in exciting and practicable directions related to performance modelling and cost estimations of system through quantitative methods. The authors presented useful methods based on Markov chains and develop these techniques for the purpose of stochastic Petri net analysis.

7.10 Self Assessment

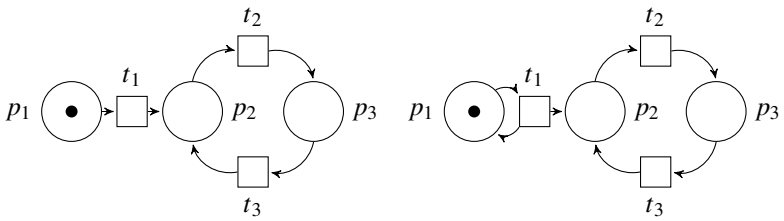
1. What is a Petri net marking? What is the initial marking?
2. What is the difference between a reachability and a coverability graph?
3. Can you list some interesting properties of Petri nets related to states? And related to transitions?
4. What does monotony express in the Petri net context?
5. What are Petri net invariants? What purpose do invariants serve? Can you name the two types of invariants and state their difference?
6. Can you name different types of High-level Petri nets and state their purpose? In these Petri nets, can you say which of the transitions are live or not?
7. For these two Petri nets, can you say which transitions are live and which ones are not?



8. In these Petri nets, what are the bounded places. What are the place bounds of this Petri net?



9. Can you identify which of these transitions are live, and which ones are not?



10. For which initial marking of P_1 this Petri net has no dead state or has dead state?