UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL AND APPLIED SCIENCES

Electronics and Computer Science

Compositional Specification and Reachability Checking of Net Systems

by

Owen Stephens

Thesis for the degree of Doctor of Philosophy

16th October 2015

Declaration of Authorship

I, Owen Stephens, declare that the thesis entitled *Compositional Specification and Reachability Checking of Net Systems* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as: [1, 2, 3]

Signed:.....

Date:.....

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL AND APPLIED SCIENCES
Electronics and Computer Science

Doctor of Philosophy

COMPOSITIONAL SPECIFICATION AND REACHABILITY CHECKING OF NET SYSTEMS

by Owen Stephens

Concurrent systems are frequently scrutinised using automated model checking, routinely using Petri nets as a model. While for small system models, it is often sufficient to give the system specification in a monolithic manner, for larger systems this approach is infeasible. Instead, a compositional, or component-wise specification can be used. However, while existing model checking techniques sometimes allow the specification of nets in terms of components, the techniques used for checking properties of the system all consider the composed, global net.

In this thesis, we investigate and advocate compositional system specification and an alternative approach to model checking that uses the structural compositional information to its advantage, vastly improving efficiency in many examples. In particular, we examine the categorical structure of component nets and their semantics, illustrating the functoriality of a map between the categories as compositionality. We introduce contextual Petri Nets with Boundaries (PNBs), adding read arcs, which naturally model behaviour that non-destructively reads the token state of a place. Furthermore, we introduce a type-checked specification language that allows us to compositionally construct systems to be modelled using PNBs, whilst ensuring that only correct compositions are expressible. We then discuss and implement compositional statespace generation, which can be used to check reachability. Via optimisations using weak language equivalence and memoisation, we obtain substantial speed ups and demonstrate that our checker outperforms the current state-of-the-art for several examples. A final contribution is the compositional specification of existing benchmark examples, in more natural, component-wise style.

Contents

List of Figures

Acknowledgements

I would like to thank my supervisors Dr. Julian Rathke and Dr. Paweł Sobociński for all of their support and advice throughout this project, their guidance was invaluable.

Thank you also to my lab-mates, friends and family, whose encouragement and support was always appreciated.

Finally, thanks to my examiners, Prof. Maciej Koutny and Dr. Corina Cirstea, whose careful reading and insightful comments lead to several improvements of this thesis.

Chapter 1

Introduction and Related Work

We introduce a technique for compositional checking of reachability in Petri nets. We show that by suitably exploiting a *compositional specification* of nets our technique can be more efficient than existing approaches. In this thesis, the introduction of our new technique is sub-divided into:

1. Investigation of compositional approaches to statespace generation and thus reachability checking of Petri nets, exploiting compositional specifications in order to avoid the statespace explosion and improve performance.
2. The application of programming language techniques to design a statically typed specification language for concise, well-defined compositional specifications.
3. Implementation of a tool for checking Petri net reachability using the techniques, with demonstrable performance increases vs existing state-of-the-art tools.

Model checking is the systematic process of determining whether a *model* of some system conforms to its expected behaviour or specification. For example, a system designer may wish to ensure that their system is *not* able to reach a problematic configuration: a lift system should not be able to be in the configuration where the doors are open *and* the lift is moving.

Due to their complex and often non-intuitive behaviour, concurrent systems are frequently scrutinised using automated model checking. Indeed, the vast number of possible interleavings of component behaviour, even for simple systems, makes it difficult to *manually* reason about the (lack of a certain) behaviour of the global system.

A prevalent modelling tool for concurrent or distributed systems is *Petri nets*. With an intuitive graphical presentation, yet rigorous underlying mathematical definition, Petri nets are powerful, yet accessible. As an example, the net illustrated in Fig. 1.1 models

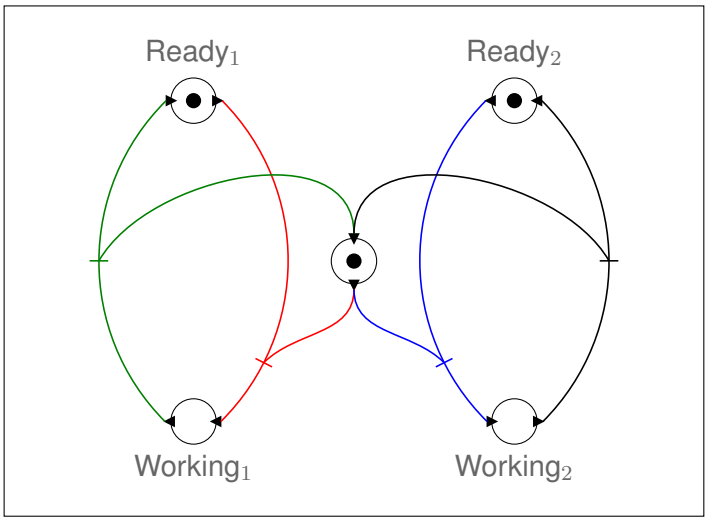


Figure 1.1: Example mutual exclusion system modelled as a Petri net

a *mutual exclusion* system, where two workers compete to obtain the token required to let them work. Either the first worker (red transition) or second (blue transition) can start first, but they cannot start at the same time. Once either worker has started, the other cannot. Once the worker has finished, it replaces the token, with the green or black transitions, allowing either worker to restart. An obvious property to check of this model is that both workers cannot be started at the same time.

For small system models, it is often sufficient to give the system specification in a monolithic manner: the global structure can be specified in one. However, for anything other than the most basic of systems, this approach is infeasible — instead, a *compositional*, or component-wise approach is often used. In a component-wise approach, a system is designed as a collection of logical *components*, which are composed to form the global system. By building systems in this way, designers need only reason about local behaviour of components and the interactions between components — the system is structured as a collection of maximally independent, decoupled entities.

The very reason to use automated model checking is to avoid having to explore large statespaces or check properties by hand. However, model checking is not immune to statespace explosion, and indeed, naively applying a compositional approach to system specification does not help alleviate the problem. For loosely coupled components, the numerous possible *interleavings* of their behaviour leads to large statespaces being generated and explored. While existing model checking techniques may allow the specification of nets in terms of components, the techniques used for checking properties of the system all consider the composed, *global* net—their approach is *monolithic*.

As an extreme example of the statespace explosion problem, consider a system formed of k components, each of which is able to reach l local states and *does not* interact with the other components; such a system has a reachable statespace with l^k states. An example of this system is illustrated in Fig. 1.2. Now consider checking reachability

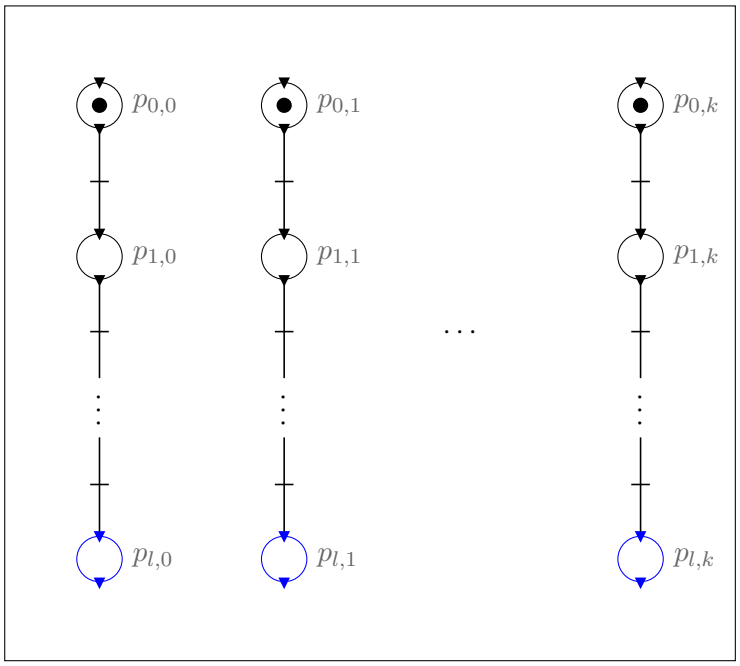


Figure 1.2: System of components with no interaction

of the global configuration that places a single token in each of the blue highlighted places; intuitively, to reach this global state, we must have fired *all* the transitions in the net, in some order that respects the individual components' dependency ordering. Due to interleaving, there are a large number of possible ways to reach the target state: if a reachability checker uses a breadth-first search strategy, or generates the entire statespace, a large majority of these interleavings will be needlessly explored.

An alternative approach is to take a *localised* view: the *global* target state can be considered as a *local* target state for *each component* — to reach the global target state, each component must reach its local target state, whilst correctly interacting with the other components. Indeed, from the knowledge of the interactions each component must make to reach its local target marking, we can check the compatibility of these component interactions, and determine if the global marking is reachable. For the example system we are considering, there are no interactions between components, so we simply check that each component can reach its local target state. By this change of view, we avoid exploring all possible interleavings, instead checking reachability of the components *in isolation*. Furthermore, since the components and target markings are repeated, we can check reachability in *a single component* and deduce that the global system *can* reach its target marking: a single component is able to reach its target marking without interacting with its neighbours, and all k components are the same.

However, in order to use a localised, component-based approach for Petri nets, we must be able to specify the *components* of a global net system. To do so, we use an *algebra of Petri nets*: we specify complex systems in terms of compositions of (small) component nets. By using an algebra that allows *partial specification* of transitions

by connecting to *boundary ports*, the full structure of composite nets can be built up through composition. As components are composed, the transitions connected to their boundary ports are fused to form fully-specified transitions. By giving a semantics that represents component reachability, while recording the corresponding boundary interactions, we can capture the reachability of the entire system, *without* first generating the composite net. Furthermore, we can *reduce* the representation of a component's behaviour, while *preserving* the representation of the interactions necessary to reach the target marking; when we compose the (reduced) component semantics, we *represent* global reachability, *without* generating the full statespace.

In other words, by taking advantage of information about the structure of the components, and how they are connected, we can check reachability vastly more efficiently.

Traditional approaches to model checking such systems instead take a global view, operating on the composite structure, and essentially ignoring any information about *how the system was formed*.

In this thesis, we investigate and advocate compositional system specification and an alternative approach to reachability checking that *does* use the structural compositional information to its advantage, in order to vastly improve efficiency in many examples.

1.1 Contributions and Thesis Structure

Summarising, the contributions presented in this thesis are as follows:

1. Categorial structure of PNBs and their semantics: we show the well-known property of compositionality in a new light, as the functoriality of a mapping between suitable categories.
2. We introduce *contextual PNBs*, adding *read arcs*, which naturally model behaviour that *non-destructively* reads the token state of a place.
3. Type-checked specification language: we show that by using a suitable programming language, we can compositionally construct systems to be modelled using PNBs, whilst ensuring that only correct compositions are expressible.
4. *Compositional* statespace generation for PNB-specified systems: we show that the statespace of a PNB-specified system can be compositionally generated, and furthermore, used to check reachability, *without* constructing the global net.
5. We show that compositional specifications can be exploited, to attack the statespace explosion problem, and improve the efficiency of reachability checking of systems modelled using PNBs. We show that by considering *weak language*

equivalence of PNB semantics, we can reduce the representation size of PNB semantics, whilst ensuring global behaviour is preserved. Furthermore, *memoisation* allows us to avoid repeated computations.

6. Compositional specification of existing benchmarks, in more natural, component-wise style, with formal, explicit specification of repeated structure.

These contributions have already been partially presented in three co-authored papers [1, 2, 3]. The individual contributions of the author of this thesis are the majority of the programming/implementation effort, while the theory was a joint effort.

The contributions demonstrated in this thesis are structured as follows:

In the following section, we explore related work from the literature. We introduce the required preliminaries in Chapter 2. In Chapter 3, we elucide the categorical structure of PNBs and the LTSs that form their semantics, exposing the notion of compositionality as functoriality. Chapter 4 introduces the example systems that we will use to demonstrate and evaluate our technique. We introduce a specification DSL that uses a static type system to ensure that only valid component-wise specifications can be constructed. In Chapter 5 we introduce a compositional technique for generating the statespace of systems specified using our DSL, and thus checking marking reachability. We prove the technique correct and give some example timings of a tool implementing the technique. Chapter 6 shows how to exploit the fact that language equivalence is a congruence to vastly improve the performance of our reachability-checking technique, introducing the notion of internal behaviour that we *ignore* in order to aggressively prune statespace. We prove the more-efficient algorithm correct. In Chapter 7, we discuss the implementation of our technique, and compare and discuss its performance relative to current state-of-the-art tools. Finally, Chapter 8, concludes and discusses future work.

1.2 Related Work

We survey and summarise the most-closely related techniques for avoiding statespace explosion, in particular when checking reachability in Petri nets, directing the reader to detailed surveys and discussions where they exist.

Before we do so, we briefly introduce each technique and give the structure of this chapter: in §1.3, we briefly discuss Petri nets and the complexity of their reachability checking. In §1.4, we discuss *partial-order approaches*, which attempt to generate only part of the statespace, while ensuring that the to-be-checked properties are preserved. In a similar vein, the *symmetry reduction* §1.5 technique attempts to generate a reduced statespace, by recognising symmetric states, which do not contribute new (up to symmetry) states to explore, i.e. the statespace is quotiented by symmetry. We then

move onto discussing other compositional (i.e. divide and conquer) approaches in §1.6, before taking a detour from statespace minimisation techniques to discuss *algebras* of nets related to that which we use in this thesis. We briefly discuss explicit structural reductions of nets that aim to reduce input nets in §1.8. In §1.9, we explore the use of *unfoldings* as a compact representation of a net's behaviour, and similarly the use of *decision diagrams* as a compact representation of a net's structure §1.10. Finally, we explore *linear programming* (LP) and *SAT solving* in §1.11, which encode Petri net problems into alternate domains possessing efficient algorithms.

The techniques to avoid statespace explosion that we explore can be summarised as taking one of the following approaches:

1. Generating partial statespaces (partial order reductions, symmetry reduction)
2. Handling a reduced input net (structural net reductions)
3. Using compact representations of the net (unfoldings, decision diagrams)
4. Translating to an alternative domain to exploit existing algorithms (SAT, LP)

1.3 Petri nets and Reachability Checking

Petri nets are a well-known mathematical model with a vivid graphical presentation, used to model concurrent and distributed systems. A Petri net consists of a set of places, and a set of transitions that connect sets of places to sets of places. Petri nets were originally introduced by Petri in his PhD thesis [4], while Murata [5] gave a detailed introduction to the definition, properties and applications of Petri nets.

The problem of *reachability* is concerned with determining if, from a given starting marking, a Petri net is able to reach a particular target marking. For general Petri nets (i.e. those that can place any number of tokens on a single place), the problem was shown to be decidable by Mayr [6], whose algorithm was simplified by Kosaraju [7] and later further simplified by Lambert [8]. The reachability problem has a lower bound complexity of EXPSPACE, as proved by Lipton, in the equivalent setting of Vector Addition Systems [9]. The survey of Esparza and Nielsen [10] covers these results in detail.

In the restricted case of Petri nets with markings that assign at most 1 token to each place—known as 1-bounded, or safe nets—the problem of checking reachability is PSPACE-complete, as demonstrated by Cheng et al. [11]. In this thesis, we only consider safe nets.

1.4 Partial order reductions

Partial-order reduction methods aim to prevent the statespace explosion caused by representing all possible interleavings of truly-concurrent events. The method relies on determining those transitions that have low mutual interference; two transitions interfere if firing one enables or disables the other, or if the resulting marking changes when the order of firing of the two transitions is swapped. A reduced state space can be built and explored if sets of transitions with low mutual interference are only fired in only one of the possible orderings. Since the marking reached after firing the set of transitions is the same, regardless of the chosen order (if it were not, there would be a pair of transitions that interfere), the particular chosen order does not matter. The name *partial-order* methods is used since those transitions that do not interfere can be considered as unrelated by some *dependency* ordering. As an example, consider the example net shown in Fig. 1.3, originally due to Wolf [12]. The transition t_4 does not interfere with t_2 or t_3 and therefore we can only consider one of the three possible interleavings. The corresponding statespace, illustrated in Fig. 1.4, contains 6 redundant transitions and 4 redundant states, highlighted with dashed lines.

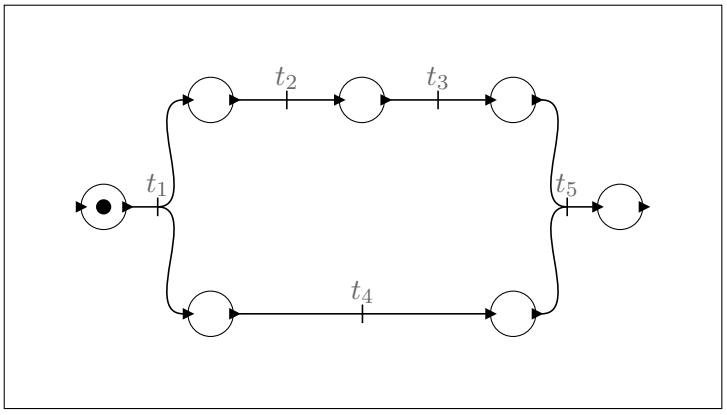


Figure 1.3: Example Petri net

To explore a reduced statespace using the partial order method, the essential idea is to fire only a subset of the enabled transitions at any particular marking. Indeed, the trivial case would be to take the set of all enabled transitions, but firing such a set offers no possibility of exploring a reduced statespace. However, it is not necessarily the case that firing smaller sets of transitions leads to a smaller explored statespace [13]. For a trivial example, consider the simple net in Fig. 1.5, both t_1 and t_2 are enabled (and do not interfere). If we fire both simultaneously, the explored statespace will have two states with one connecting transition; however, if we pick a smaller set of transitions (i.e. firing t_1 or t_2 alone), the explored statespace will have 3 states and 2 transitions since the transitions will have been interleaved.

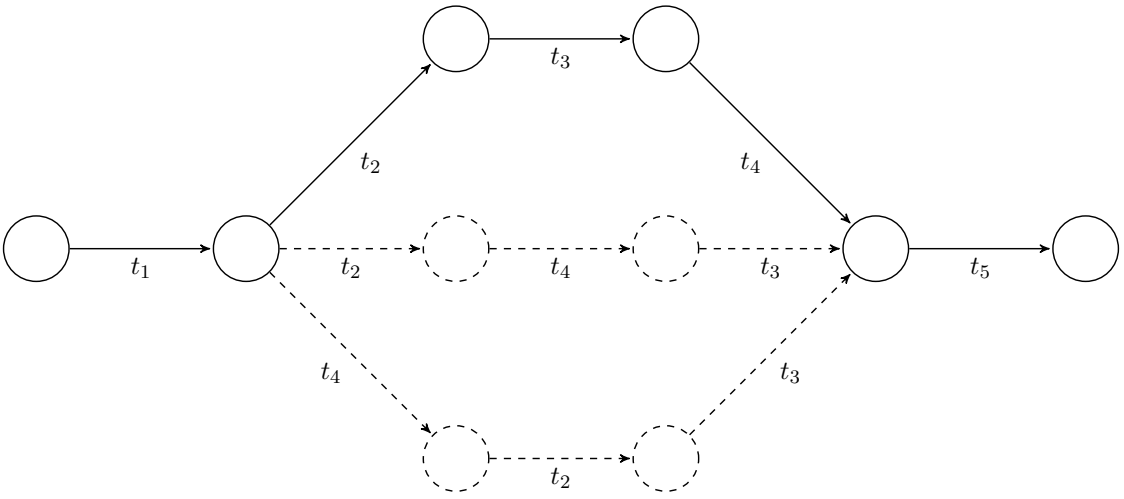


Figure 1.4: Statespace of the net in Fig. 1.3

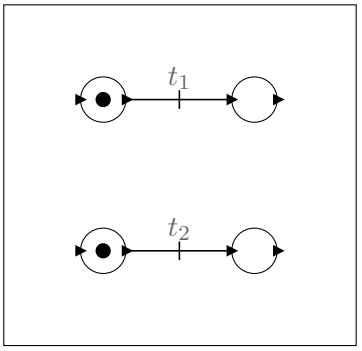


Figure 1.5: Example Petri net

Several related partial order reduction approaches exist in the literature, with slight variations on the set of transitions considered in each case. So called *stubborn* [14], *persistent* [15] and *ample* [16] set methods each specify the (sub)set of the enabled transitions for a state that should be explored. Here, we only present an intuitive overview of the stubborn set method; details of the precise differences between the approaches can be found in surveys by Valmari [13], Godefroid [17] and Clarke et al. [18]. Additionally, in recent research, Valmari and Hansen [19] investigated optimality of stubborn sets and the effect of particular choices of dependency relations on transitions.

As a general definition (c.f. [13]), a set of transitions S is *stubborn* iff:

1. If firing some sequence of transitions *not* in S allows some $t \in S$ to be fired, reaching a marking m, then we can also fire that same sequence *after* firing t, to also reach m,

2. There is at least one $t \in S$ that is enabled after firing any length sequence of transitions *not* in S.

In other words, the firing of some transition in S commutes with the firing of transitions not in S. Thus, the search algorithm is justified in choosing the ordering where (the enabled transitions of) S are fired first: the marking reached in either order is the same.

However, to ensure that the possible statespace reduction enabled by the stubborn set method is sound w.r.t. a particular net property, the definition of stubborn sets must take into account the particular property being checked. For example, considering the trivial example net shown in Fig. 1.5, if the property to check is reachability of the marking with a token in top-left and bottom-right places only, then the set of stubborn sets cannot include $\{t_1, t_2\}$ since firing this set does not preserve reachability of the target marking. Whereas early papers related to stubborn sets focussed on preserving deadlock, Schmidt [20] detailed an encoding to efficiently check other Petri net properties, including reachability, using the stubborn set method.

The stubborn set method was later improved by Godefroid and co-authors [21, 22], particularly by combining stubborn sets and *sleep* [15] sets for additional improvements. The sleep set method attempts to avoid visiting states that have already been explored by an alternative interleaving of transitions, by tracking the set of transitions fired to reach a particular state. In later work, Alur et al. [23] demonstrated efficiency improvements when combining efficient representation via decision diagrams (see §1.10) and partial order reductions using ample sets. Recent work by Hansen and Wang [24] has shown that combining stubborn sets with compositional approaches (see §1.6) to determine transition dependencies can improve the reduction effect.

An alternative to standard reachability graph representations, due to Vernadat et al. [25] is the notion of a *covering step graph* (CSG). A CSG groups independent events into single transition step, which is fired atomically, and thus implicitly avoids the state explosion due to interleaving. CSGs can be combined with the persistent set method, as shown by Ribet et al. [26].

In-depth surveys of stubborn set-like methods were presented by both Valmari [13] and Godefroid [27, 17], examining the practical applicability and effectiveness of stubborn set methods for various Petri net model checking problems.

1.5 Symmetry reduction

Another technique, known as *symmetry reduction*, exploits symmetries in the structure of a net (and thus its reachability graph): the goal is, roughly, to build a reduced, quotiented reachability graph that satisfies the same temporal properties as the original, with only one representative of each equivalence class of states.

As a simple example of the symmetry-reduction technique, consider the input net illustrated in Fig. 1.6; the symmetry of this net is plain to see: the net is formed of two identical components. The LTS semantics of this net (omitting self-loops and labels for simplicity) is shown in Fig. 1.7. The statespace explosion problem is self-evident: the semantics contains all interleavings of the concurrent transitions. However, taking the symmetry of the input net into account, we may identify sets of LTS states, giving the LTS semantics shown in Fig. 1.8. Intuitively, since the "components" of Fig. 1.6 do not interact, we are free to "permute" them, removing the difference e.g. between the left component firing all its transitions before any in the right and vice-versa. Indeed, taking $[x]$ to be the partition of states equivalent (under the particular symmetry) to x, we have that marking y is reachable in the net if there is a path from the state representing the initial marking, i, to y in the LTS if there is a path from $[i]$ to $[y]$ in the quotiented LTS. Since the quotiented LTS has fewer states and paths, it is more efficient to check reachability using this LTS, compared to the full LTS.

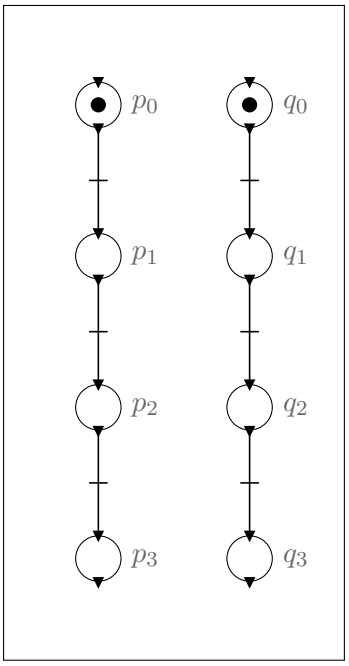


Figure 1.6: Example Petri net

The first technique for exploiting symmetry in model checking was due to Starke, in the context of checking reachability of Petri Nets [28]. Essentially, the reachability graph construction algorithm only considers new states (i.e. markings) if an equivalent state (w.r.t. the symmetry of the net) has not already been visited.

Further initial results on model checking using symmetry reduction were obtained by Clarke et al. [29], showing that symmetry reduction preserves satisfaction of CTL* formulas (preservation of formulas being the key property for the validity of symmetry reduction). Emerson and Sistla [30] presented theoretical results in the area of μ-calculus

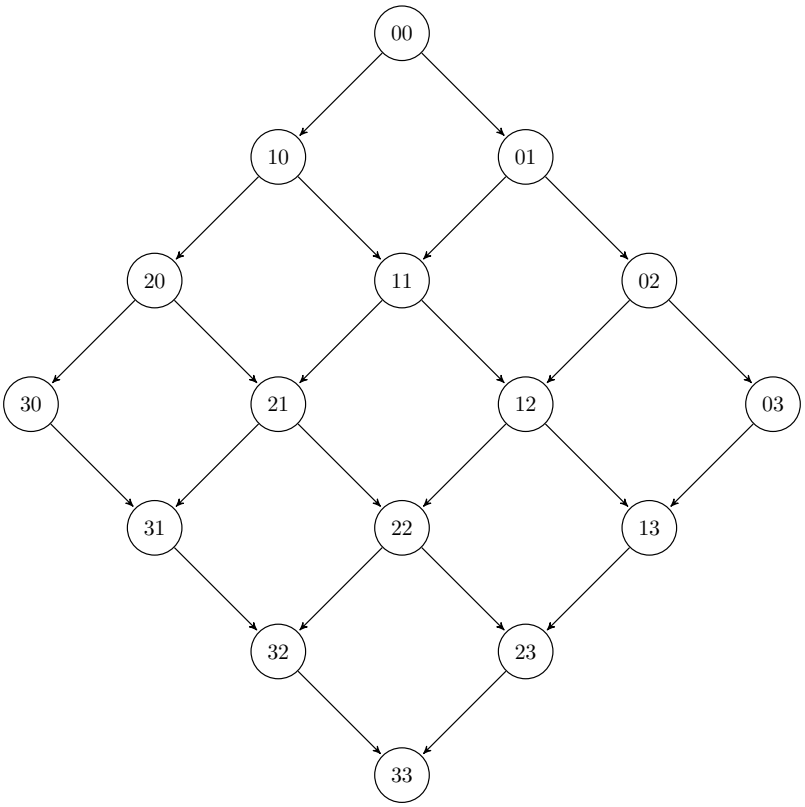


Figure 1.7: LTS semantics of the net in Fig. 1.6

model checking, while Ip and Dill [31] investigated the more practical problem of how to identify symmetries to be exploited.

There are two key problems [32] that must be solved in order to use symmetries:

1. How to determine that two states, s and s' belong to the same orbit (set of equivalent states), known as the *orbit problem*,
2. How to determine the symmetry group, from the input model.

Indeed, for explicit-state representations, one can exploit a *canonical representative* for each set of equivalent states. The transition relation is then explored up-to representatives, ensuring that only one representative state of each equivalence set is explored. Model checking using explicit-state systems is efficient if it is efficient to compute a canonical representative, or similarly, if two states are equivalent. Unfortunately, under arbitrary symmetries, checking state equivalence is as hard as the graph isomorphism problem [32], a computationally difficult problem that has no P-time algorithm, yet has not been shown to be NP-complete [33]. Several successful explicit-state model checkers exploit symmetry [34, 35, 36].

To determine the symmetry group, one approach is to explicitly specify the symmetries in the input model, such as the *scalarset* data-type in the input language of Ip and

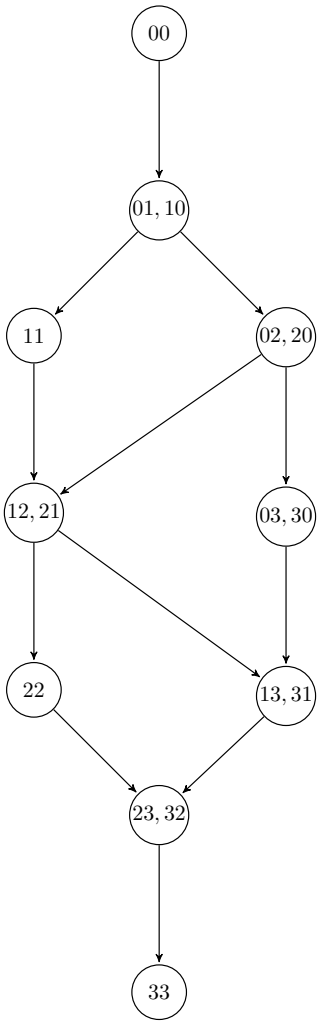


Figure 1.8: Quotient of LTS semantics in Fig. 1.7

Dill's [31], or components that may be permuted. Alternatively, an implicit approach attempts to determine the symmetry groups by examining the input model, for example, Schmidt [37] uses a partition refinement algorithm, to build the coarsest symmetry relation on input Petri nets that preserves the flow relation of the net in order to calculate the symmetries of a Petri net. The complexity of Schmidt's method is investigated by Junttila [38]. The performance of three alternative symmetry-reduction implementations is contrasted by Schmidt [39], who examines the trade off between reduction time and effect on reachability checking performance.

Summarising, for systems with inherent symmetry (i.e. those with collections of similar sub-systems), symmetry reduction can provide large reductions in the number of states that must be explored. However, the advantage can only be realised if efficient methods of detecting symmetry and checking equivalence of states under it are available.

Thorough survey articles covering symmetry-reduction have been compiled by both Wahl and Donaldson [40] and Miller et al. [41].

1.6 Compositional approaches

Compositional approaches to model checking decompose monolithic systems into smaller components, which are checked, giving *local* results. These local results are combined to form a global result for the original monolithic system. Such approaches are known as *divide and conquer* — the idea being that the smaller components are easier to check, and local results carry less information than the local component. Thus, by decomposing (checking, and composing local results), the amount of information that must be processed is (vastly) reduced.

Clarke et al. [42] describe a compositional approach to model checking systems with temporal logic specifications. They observe that the main difficulty is the preservation of local properties at a global level, and introduce a framework using *interface processes* to model the environment, when checking individual components, to ensure that components only exhibit globally-valid behaviours.

Compositional approaches using process algebra techniques date back at least to Milner's seminal CCS [43]. Yeh and Young [44] outline a reachability analysis that allows the simplification of "intermediate" components, that helps alleviate the state explosion problem, by replacing components with semantically-equivalent, but smaller components. Similarly, Valmari [45, 46] describes compositional state space generation, a divide and conquer approach to global state space generation that employs minimisation of local state spaces to avoid state explosion. A CSP model is used, modelling synchronous communication of labelled Petri nets; indeed, this method of decomposition restricts the possible minimisations, since the global synchronisations must be preserved. Furthermore, Valmari does not use any form of memoisation to prevent repeated work. Later, Valmari [47] presented a similar decomposition technique, using shared "boundary" places, instead of shared transition labels. This allowed for more natural LTSs: transitions were labelled with the effect on boundary places rather than net transition names. A similar approach was taken by Kindler [48], who introduced Petri net components, with distinguished input/output places that are fused to form larger components. Kindler gave a fully-abstract semantics, whereby the semantics of components under composition agrees with that of closed components. Very recent work by Baldan et al. [49] has elucidated compositional encodings between (a)synchronous process-calculi and a variant of Petri nets, with distinguished places/transitions used for synchronisation. Indeed, such an encoding gives "technology transfer": techniques for model checking properties in process-calculi can inform those in Nets and vice-versa.

An important consideration for compositional model checking approaches was noted by Graf and Steffan [50] and Cheung and Kramer [51]: local minimisation can be inefficient if it does not explicitly exclude behaviours that will never be performed by the context of a component. Both use interface constraints similar to the interface processes of Clarke

et al. [42] to prevent spurious behaviours of components, with Cheung and Kramer giving an algorithm for interface generation without requiring an LTS for the context. Krimm and Mounier [52] employed these ideas for compositional statespace generation for LOTOS programs, including large, complex examples.

In the area of Petri nets, Lee et al. [53], and Rakow [54] employ *slicing* to decompose a Petri net into concurrent units, in order to check reachability on a per-slice basis, thus avoiding some of the inherent state explosion due to interleavings. Whereas Lee et al. and Rakow use a *static* slicing (i.e. without considering the markings of a net, just its structure), Llorens et al. [55] propose a *dynamic* approach, which takes account of an initial marking of the net. However, slices are not *true decompositions* (i.e. disjoint: transitions (and places) of the original net should appear in only one slice), since they necessarily overlap, with transitions being shared among (many) slices. Furthermore, if the marking to be checked involves a large proportion of places in the net, little benefit will be gained by using slicing.

Christensen [56] uses a compositional approach to generate the statespace of Petri net components that synchronise via shared places or transitions. By generating a *synchronisation graph* capturing component communications, in addition to *local* reachability graphs of components, the full statespace can be implicitly represented, without generating all concurrent transition interleavings. A similar approach was described by Notomi and Murata [57], however, whereas at each step of Christensen's algorithm the local reachability graphs are only unfolded as far as the next *global* synchronisation, Notomi and Murata's approach generates full local reachability graphs, before reducing w.r.t. synchronisations and combining them. Thus Notomi and Murata's approach may explore local reachability graph states that are never explored in the global statespace, due to non-synchronisation. Furthermore, while Notomi and Murata describe reduction by merging states that are equivalent w.r.t. synchronisations, they do not give an algorithm for determining the set of such states. A similar reduction strategy is employed by Bucholz and Kemper [58], who abstract over *autonomous regions* (subnets delimited by synchronisations), simplifying the representation of local reachability graphs.

1.7 Algebras of nets

Compositional algebraic systems have the property that behaviour of composite systems is determined only by the behaviour of their component systems. While Petri nets are sometimes considered as being inherently non-compositional, early work by Mazurkiewicz [59] defined a compositional algebra of nets, based on fusion of named transitions. In the algebra of nets we use in this thesis, Petri Nets With Boundaries, (PNBs) [60, 61] it is also transitions that are fused, however, the composition operations are quite different in nature: PNB compositions are not commutative, and operate

on transitions connected to local *boundary ports* rather than by (global) name. Indeed, Mazurkiewicz's composition is a commutative parallel composition in the spirit of CSP and CCS.

Similar operations were used for the development of the Petri Box calculus (PBC), a process algebra of labelled Petri nets [62], with a compositional Petri net semantics. The PBC features two kinds of composition: the first is a control-flow-style sequential composition that utilises certain places labelled as *entry* or *exit* places in order to enforce a computation order, the second a synchronising composition, introducing new transitions based on the *global* fusion of transitions with conjugate labels, whilst preserving the original transitions. The composition operations of PNBs, instead, are closely related to the geometry of nets, with no control-flow style composition and only *local* synchronisation (that fuses transitions) through shared boundary ports. Best and Koutny [63] later introduced the Box algebra, a generalisation of the PBC, giving a (compositional) operational and denotational semantics and a general view of various composition operations as specific instances of generic transition refinement/relabelling operations. A tutorial-style overview of the Box algebra can be found in [64].

Reisig's [65] simple composition of nets (SCN) is an elegantly simple way of composing nets and is conceptually quite close to our work. His nets, similarly to PNBs, have left and right interfaces that are made up of ports and ought not to be confused with notions of input and output, rather reflecting the structural geometry of nets. Differently, in SCN the interfaces typically expose places, whereas PNB interfaces expose only transitions. Another difference is that in PNB, composition $N_1 \; ; \; N_2$ is only defined when the right interface of N_1 is equal to the left interface of N_2. While [65] demonstrates that the operation is very natural for composing real systems, the compositional semantic aspects of the theory have not, so far, been developed.

Component-wise construction of nets was emphasised by Kindler [48], who worked with a partial order semantics. The interfaces are a set of input and output places, which are connected with a transition when composed. The semantics was shown to be compositional with respect to this operation. Since the composition introduces additional transitions, it is not always clear how to *divide* a net into components with input and output places. A similar approach was taken by Baldan et al. [66], who introduced *Open nets*: nets with certain identified input and output places. Composition of Open nets is defined in terms of a pushout in the category of Open nets, realised as joining a common subnet. Another related approach is that of Priese and Wimmel [67], who use a combination of interface places *and* transitions to define *Petri nets with interface*, with composition operators that join interface places and transitions. An algebra of nets is defined, with combinators to form complex nets from basic components.

In this thesis, we will give a compositional semantics to the algebra of Petri nets with boundaries, by a translation into the algebra of non deterministic finite automata with

boundaries (§2.3.2). This algebra is an instance of the algebra of Span(Graph) [68], developed by Walters and collaborators: in fact, a translation from Petri nets to this algebra was already present in [69]. In more recent work [60, 70, 61], the algebra of Span(Graph) was lifted to the level of nets in a compositional way, and the resulting behavioural equivalences and connections with process algebra were explored.

1.8 Structural reductions

An alternative approach to aid the minimisation of a Petri net's reachability graph is to directly minimise or quotient the Petri net, intrinsically reducing its reachability graph. By removing redundant transitions or places, the net came become structurally smaller and thus lead to a smaller statespace to explore. Indeed, such simple fusion/elimination rules were discussed by Murata [5], which were used to quotient Petri nets, whilst preserving their behavioural properties.

Esparza and Schröter [71] introduced a technique for checking LTL properties of 1-safe nets that combined global and local reduction techniques. Globally, dead places/transitions (places that are never marked and thus never enable their outgoing transitions) and implicit places (places that are redundant w.r.t the enabling/disabling of a particular outgoing transition) are identified and removed by the (linear programming) marking equations they satisfy. Locally, generalisations of Berthelot's [72] *agglomeration* rules are used to remove certain intermediate places and their transitions, by bypassing such places. Haddad et al. [73] also generalised Berthelot's approach, but without the restriction to 1-safe nets. Furthermore, by deriving structural conditions for reduction from the behavioural properties (i.e. firing sequences) to be preserved, Haddad et al. were able to use weaker structural conditions, enhancing the effect of the reduction.

More recently, Rakow [74] also presented an approach for checking LTL properties on 1-safe nets, but used a decomposition into a "kernel" net, containing only the places mentioned in the LTL formula, and (reduced) environment subnets recording the remainder of the net's influence on the kernel.

Originally proposed by Olderog [75], the notion of *bisimulation* can be lifted from LTSs, to the places of a Petri net. Olderog's approach did not in fact lead to well-defined bisimulations, as Autant et al. [76] observed, while showing how to generalise the definition, and give a correct notion of place-bisimulation for Petri nets. Indeed, Autant et al. used the induced equivalence relation to define the quotient of a net (i.e. a structurally smaller, but bisimilar net), however, they gave no algorithm for calculating such quotients in order to reduce a given net. In a later paper [77], Autant et al. did provide a simple algorithm to calculate such bisimulations, extending their approach to labelled nets with τ (internal) transitions. Schnoebelen and Sidorova [78] refined Autant et al.'s

approach, showing that by considering a set of markings as relevant (as opposed to all markings), the effectiveness of the reduction can be improved.

1.9 Unfoldings

The *unfolding* of a Petri net is a structure representing all of its possible behaviours, originally introduced by Nielsen et al. [79] under the name of *Occurrence Nets*. Unfoldings were later described in more detail by Engelfriet [80], who used the name *branching processes*. Essentially, a branching process represents some partial "history" of a Petri net from some initial marking, as another Petri net, but with the condition that at each conflict of transitions that could be fired, each alternative is represented by a copy of the (thus-far constructed) branching process. A branching process is thus an acyclic net without "backward conflict", where two transitions output to the same place; the lack of backwards conflict implies that each set of places in the unfolding has a unique trace of transitions that were fired to place a token in each place. An unfolding is therefore a compact representation of all possible computations of its underlying net.

Intuitively, an unfolding is an tree-like structure: branching points in the net signify *choices* of enabled-transition firing. The advantage of an unfolding (in fact, itself a Petri net), over a state graph, is that concurrency is represented explicitly, rather than implicitly. Simple structural properties of an unfolding determine whether or not a given pair of transitions can fire concurrently in the original net, or are causally related (i.e. the firing of one can lead to the other becoming enabled). While these properties can be derived from the state graph, it is certainly a more arduous task to do so. Furthermore, the well-known "state-explosion" problem leads to interleaving semantics (e.g. state graphs) of concurrent systems being large, even for small or simple systems. By explicitly representing points of choice and synchronisation (and not representing all possible interleavings of concurrent transition firings), unfoldings give much smaller representations. Indeed, unfoldings are often referred to as *partial-order* semantics, since arbitrary transitions may not be causally related (i.e. firing one eventually enables the other), when the transitions are concurrent.

Example 1.1.
An example Petri net is shown in Fig. 1.9, and its unfolding is illustrated in Fig. 1.10. Note that the unfolding is infinite — we only show a small, finite prefix of it. The graphical notation we use is not-standard, but should be intuitive; we introduce this notation later in the thesis, in §2.6.1. Consider the transitions t_1 *and* t_2*; these transitions* can *be fired concurrently, and in the induced partial (causal) order of the unfolding, there is no relation between the transitions, which witnesses this fact. Intuitively, this fact is observed since there is no path from the post-set of* t_1 *to the pre-set of* t_2*. Note however, that* t_3 is *causally-related to both of these transitions. Furthermore, since there*

is a (forward) conflict (i.e. firing choice) between t_6 and t_7, the unfolding contains a branching point at these transitions.

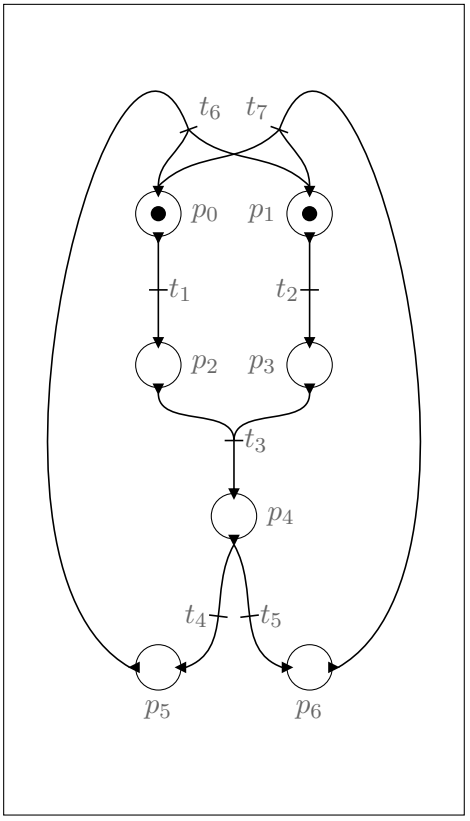


Figure 1.9: An example Petri net

Using unfoldings as a method of checking reachability in Petri nets was pioneered by McMillan [81], using the semantics proposed by Nielsen et al. [79] and Engelfriet [80].

McMillan observed that by considering a suitable *prefix* of the unfolding, which represents all reachable markings of the original net, the full (often infinite) unfolding need not be constructed. Such prefixes are known as *finite (marking-) complete prefixes*. McMillan's algorithm used such prefixes to check reachability (precisely, coverability) of a set of places, T, by supplementing the original net with a new transition, t, whose pre-set was T. If, during the execution of the algorithm (that is, while the constructed unfolding was not a *complete* prefix) t was found to be enabled, then T *was* indeed coverable. If the complete prefix was constructed before t was encountered, the marking was not coverable.

McMillan's algorithm, while correct and simple, is inefficient in terms of the size of the complete prefix generated, and has been subsequently improved by several authors. Esparza et al. [82] improved the algorithm to construct smaller (complete) prefixes, by showing that more transitions in the unfolding can correctly be considered as "cutoff" points of the unfolding, which are not unfolded further. Heljanko [83] further refined the work of Esparza et al. to generate even smaller prefixes in some examples (though the

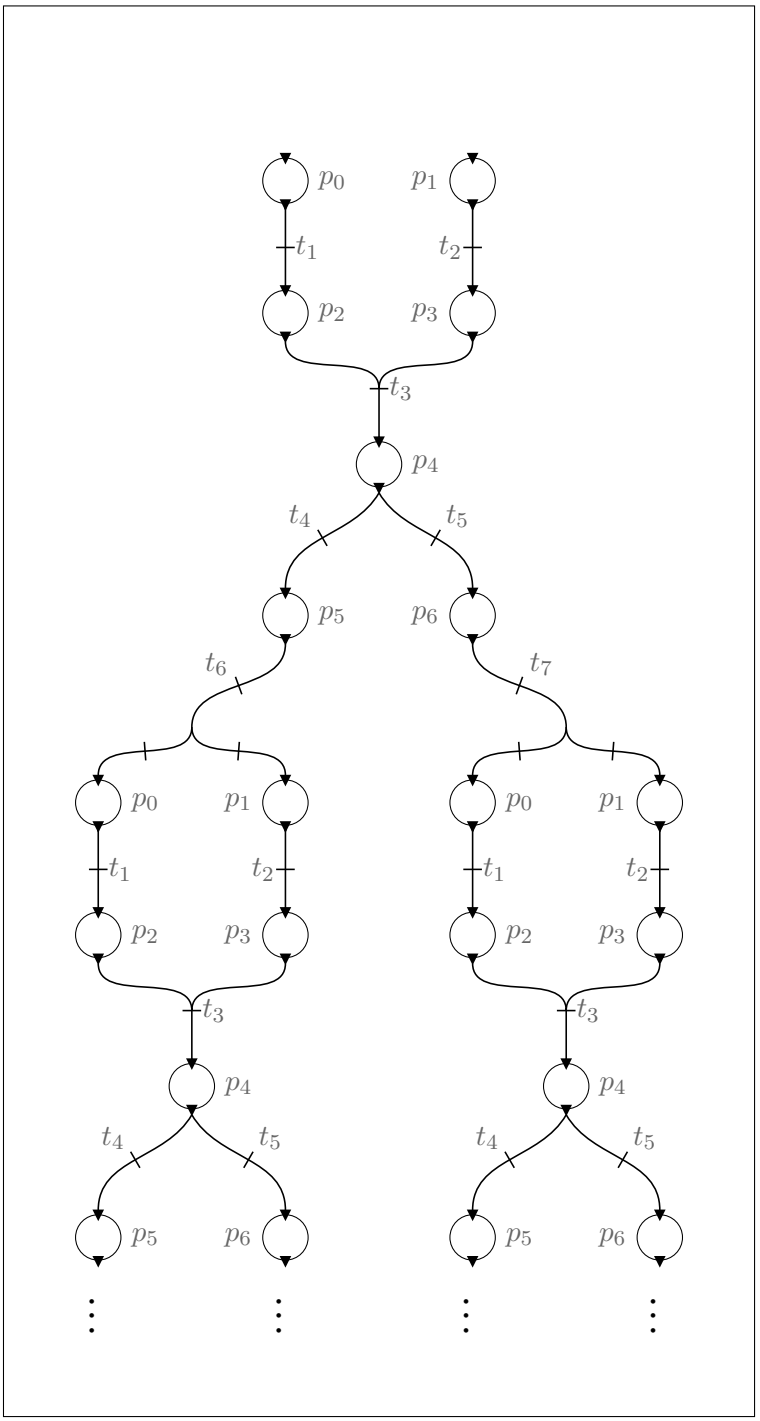


Figure 1.10: Unfolding of the Petri net in Fig. 1.9

author questioned the benefits of the size reduction vs. the creation effort involved). Khomenko and Koutny [84] suggested an alternative to the algorithm of Esparza et al., using a different method of computing possible extensions of a (incomplete) unfolding, that is more efficient in the examples they present. Heljanko et al. [85] showed how the approach of Esparza et al. may be parallelised for further increases in efficiency. More recent work by Bonet et al. [86] employed heuristic search algorithms in an attempt to "direct" the unfolding procedure towards the target transition, rather than existing

approaches using simple search strategies. Khomenko et al. [87] generalised the definition of complete prefix of an unfolding, that is applicable to nets with e.g. equivalent markings, or extra data that is to be considered along with markings. Finally, Neumair et al. [88] show that even unbounded Petri nets (those without finitely many reachable states) may still have their statespaces by suitable extensions of unfoldings.

In addition to McMillan's approach (and its improvements), several methods exist for checking the reachability problem in nets that use (complete prefixes of) unfoldings. Esparza and Schröter [89] surveyed 4 such algorithms. They recommend an algorithm to use based on whether the marking is expected to be reachable or not. If reachability is not expected, they suggest the algorithm of Heljanko [90], which translates the reachability problem into a logic program and checks for the existence of a stable model of the program. If the marking is expected to be reachable, they suggest using (an improvement of) McMillan's algorithm.

An important point to note regarding unfoldings, and indeed, finite complete prefixes, is that they carry more information about the computations of nets, than merely marking reachability. For instance, it is possible to check for deadlock, as shown in McMillan's seminal paper [81]. Furthermore, automata-theoretic approaches to LTL model checking can be adapted [91, 92] to unfoldings.

For an overview of the extensive field surrounding unfoldings and the corresponding model checking approaches, see [93] and [94]. Since unfoldings represent choice between transitions as copying, it is possible to obtain unfoldings that are exponentially larger than the net from which they are generated. For example, if a net consists of a sequence of choices between transitions (an example of such a net is given later in this thesis: §4.2.3), then each choice introduces a branching point in the unfolding, leading to an exponential blow-up in unfolding size. Indeed, unfoldings cope poorly when state explosion is caused by a sequence of choices, rather than inherent concurrency [95]. Recently introduced by Khomenko et al. [96, 95], merged processes are a condensed representation of a Petri net's behaviour, remedying the problem of sequenced choices, whilst being amenable to (generalised) unfoldings-based model checking. Merged processes can be thought of as reduced unfoldings, where certain conditions are identified, and duplicate events are removed, quotienting the unfolding.

In the original presentation, a merged processes was obtained by reducing a precomputed unfolding; however, in recent work, algorithms have been developed that directly compute merged processes [97].

1.10 Decision diagrams

In this section we move on to lower-level optimisation techniques, concentrating on efficient symbolic representation of nets and their semantics.

Symbolic state space representations aim to be more efficient (in terms of memory-requirements) than explicit representations, often leading to vast improvements in the performance of corresponding model checkers. Rather than storing and exploring states one-by-one, symbolic model checkers store and explore multiple states at once. The most prominent initial work on symbolic model checking is due to McMillan, in his PhD thesis, and later published together with Burch et al. [98]. Burch et al. gave a symbolic method of model checking μ-calculus formulae, using (Ordered) Binary Decision Diagrams (OBDDs); by checking μ-calculus formulae, they were able to generalise previous BDD-based model checking techniques targetting concrete instances: Coudert et al. [99] model checked Mealy machines [100] (automata with state-dependent output), while Browne et al. [101] described a technique for model checking circuits against CTL specifications.

Ordered Binary Decision Diagrams (OBDDs, more commonly just BDDs) were popularised by Bryant [102], who showed that BDDs canonically represented functions over the Booleans, whilst allowing efficient manipulation, e.g. performing conjunction or checking for tautology. A simple example is illustrated in Fig. 1.11, showing a representation[1] of the formula $(a \wedge b) \vee (a \wedge c)$, which could represent the collection sets of states $\{\{a, b\}, \{a, c\}\}$ where a particular property holds.

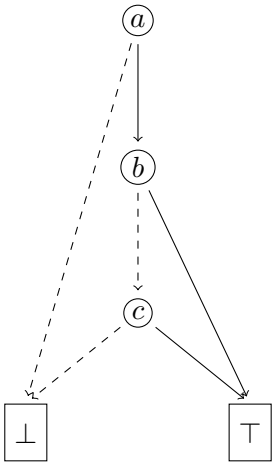


Figure 1.11: BDD representing $(a \wedge b) \vee (a \wedge c)$: dashed edges represent false, solid represent true.

Using suitable encodings of states and transitions, entire state graphs can be represented, and properties (for example, reachability) can be checked using a fixed-point

[1] A dashed line represents that the source variable was false, a solid line, true.

operation on the underlying BDDs. Detailed introductions to BDDs, their use and algorithms can be found by Andersen [103] and Drechsler and Sieling [104].

Symbolic BDD representations can be applied to Petri nets. Pastor et al. [105] gave encodings of the markings, transitions and firing rules of 1-safe Petri nets as BDDs. These encodings were used to give the first symbolic method of calculating all reachable states of the net, and furthermore, verification of liveness and safeness properties. Pastor and other collaborators later improved the algorithm [106, 107], using more efficient representations of Petri net markings (using a suitable encoding method, structurally-related places could be represented using fewer variables). Semenov and Yakelov [108] took a different optimisation approach, combining the symbolic exploration method with the unfolding technique to discover "clusters" of places that are never simultaneously marked; such clusters have minimal BDD size, regardless of variable ordering and thus assigning "close" variables to elements of clusters helps minimise the size of the resulting BDD.

Miner and Ciado [109] improved the efficiency of the statespace generation technique of Pastor et al.. Using Multi Decision Diagrams (MDDs)—a generalisation of BDDs introduced by Srinivasan et al. [110], which encode functions of the form $\mathbb{N}^n \to \mathbb{N}$ rather than $\mathbb{B}^n \to \mathbb{B}$ as in BDDs—and suitably partitioning the input net's places into *localities* (similar to the clusters of Semenov and Yakelov), Miner and Ciardo were able to generate the statespace with fewer iterations and a corresponding reduction in time. In later work, Ciardo et al. [111] further refined the use of event localities, to further improve the algorithm's performance. The benefit of using higher-level decision diagrams has been exhibited, for example by Strehl and Thiele [112] and Tovchigrechko [113].

While BDDs *can* encode functions with (co-)domains other than the Booleans, using a direct encoding can be both more intuitive and storage-efficient. While MDDs fully generalise BDDs to functions on the naturals, a finer generalisation is to encode functions with co-domains other than the Booleans. This natural generalisation of BDDs was originally observed by Clarke et al. [114] and Bahar et al. [115]. Such generalised BDDs, known as Multi Terminal BDDs (MTBDDs) have been used for a variety of purposes, including: efficiently representing matrices [116], model checking of probabilistic timed-automata [117] and for representing specifications of parallel programs [118].

A key problem related to the use of BDDs is that a particular ordering for the BDD's variables must be chosen a priori. Unfortunately, choosing an optimal variable ordering for a BDD is an NP-complete problem [119]. However, heuristics have been developed, which attempt to minimise the size of a particular BDD. Static approaches attempt to determine a fixed variable order, during construction of the BDD; for example, the variable-interleaving technique of Fujii et al. [120], which aims to place semantically-related (Fujii et al. consider the topology of the logic circuit) variables near each other in the BDD. On the other hand, the dynamic approach of Felt et al. [121] recognises that

statically-determined variable orders may not be optimal after operations have been applied to the BDD. Instead, they monitor the size of intermediate BDDs and rearrange small "windows" of consecutive variables to reduce the resulting BDD's size.

More recently, so-called *exact* algorithms have been further investigated, to improve the non-optimal results of heuristic algorithms. Indeed, Ebendt et al. [122] improve on the original algorithm of Friedman and Supowit [123] and improved algorithm of Drechsler et al. [124], by using the A* search algorithm to explore (subsets of) the set of all variable orderings, and using local heuristics to choose the best next variable at each step.

A recent, detailed overview of BDD optimisation techniques can be found in the book by Ebendt et al. [125]. Finally, while it is common that (variants of) BDDs are used symbolic model checking, as pointed out by Biere et al. [126], it is not the case that symbolic methods imply the use of BDDs, viable alternatives certainly do exist.

1.11 Linear programming & SAT solving

We briefly consider two related approaches, which solve Petri net model checking problems, by faithful translation into an alternate domain that possesses efficient algorithms. By translating to, and using efficient solving in the target domain, the statespace explosion can be avoided to a certain extent. The two approaches we cover are SAT and Linear/Integer Programming.

The Boolean Satisfiability (SAT) problem is concerned with determining the existence of a satisfying variable assignment V of a Boolean function, F. For example, the formula $F = x \vee (y \wedge z)$ is satisfied by $V = \{x \mapsto 1, y \mapsto 1, z \mapsto 0\}$, whereas the formula $(x \wedge y) \wedge (\neg x \wedge z)$ is *not* satisfiable. Using suitable encodings of the markings and transition relation of a net, it is possible to encode reachability problems as SAT problems. To restrict the size of input formulae, SAT solving is often used in a *bounded* context, where counter-examples are only considered up to some maximal size. Indeed, by *unrolling* the transition relation of a net k times, Petri net model checking can be bounded by searching for occurrence sequences reaching the target marking with length $\leq k$. For example, in the net of Fig. 1.12, the marking $\{p_2, p_3\}$ can be reached in bound 3.

A SAT-based approach for bounded checking of 1-safe Petri nets was introduced by Ogata et al. [127] that included checking of reachability and liveness properties, and illustrated the importance of compact encodings for efficient SAT solving. A related approach was used by Heljanko [128], who gave an encoding into *boolean circuits*, which generalise boolean formulae, highlighting the efficiency implications of using process [129], step or interleaving semantics.

Linear programming is the minimising of a linear equation, subject to linear constraints. Indeed, with suitable encodings of the markings and flow relation of an acyclic net, reachability problems can be approximated by determining the existence of a solution to a system of linear equations. For example, consider the net, N, shown in Fig. 1.12.

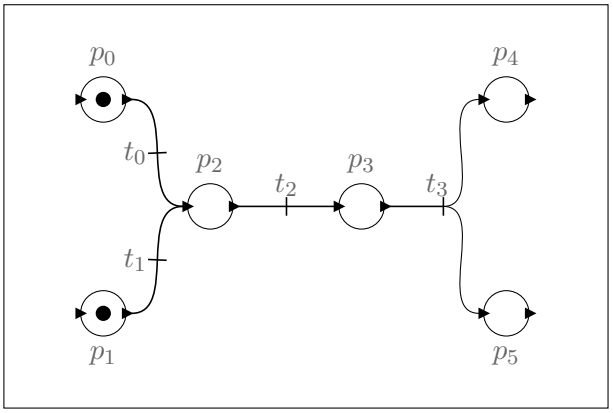


Figure 1.12: Example net, N

The *incidence matrix* of N is a $\left|\text{places}(N)\right| \times \left|\text{trans}(N)\right|$ matrix, where $I_{i,j}$ denotes the effect of firing transition t_j on the token count[2] of place p_i:

$$IM = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 1 & -1 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The initial/target markings are column vectors with $\left|\text{places}(N)\right|$ rows:

$$M_I = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad M_T = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

And finally, the *marking*, or *state equation* [5] is:

$$M_T = M_I + IM \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{1.1}$$

[2] 1 signifies that a token is added, -1 that it is taken away and 0 denotes no effect.

where the final column vector is referred to as the *Parikh vector*, with its x_i representing the number of occurrences of transition i in a firing sequence starting in M_I and reaches M_T. Marking equations are thus algebraic representations of the reachable markings of a net.

For the example net of Fig. 1.12, equation 1.1 has a solution of $x_0 \mapsto 1, x_1 \mapsto 1, x_2 \mapsto 2, x_3 \mapsto 1$, corresponding to the firing sequence: $\langle t_0, t_2, t_3, t_1, t_2 \rangle$.

Melzer and Römer [130] used linear programming to optimise the search strategy for certain instances of a deadlock-checking unfolding algorithm due to McMillan [81]; indeed, since unfoldings are acyclic, the linear programming approach is sound. In later work, Khomenko and Koutny [131, 132] improved Melzer and Römer's approach and extended it to check other properties such as reachability; furthermore, by encoding causality/conflict properties derived from the unfolding in the system of linear constraints, the solver's search space was reduced, exhibiting an impressive speed-up.

However, as Esparza [133] notes, since the linear programming solutions (over-)approximate reachable markings, the target marking may not in fact be reachable. However, the inverse implication does hold: if there are no solutions, the marking is certainly not reachable. Detailed overviews of the applications of linear programming techniques for analysing (P/T) nets can be found by Desel [134] and Silva et al. [135].

Chapter 2

Preliminaries

In this chapter, we introduce the required knowledge for our work; the following content is not original to this thesis, except the section on read arcs for Petri Nets With Boundaries (§2.6.9), which is an original contribution.

2.1 Notation

Here we introduce key notation used throughout this thesis.

$k, l, m, n \in \mathbb{N}$	Natural numbers: $\{0, 1, 2, \dots\}$
$\mathbb{B}$	Boolean values: $\{0, 1\}$
L^*	Free monoid, (all finite strings) on L
$\vec{l}$	Finite string: $\langle l_0, l_1, \dots, l_{k-1} \rangle$, of length $k \in \mathbb{N}$
ϵ	The unit element (empty string)
$z = xy$	If x and y are strings, z is a string formed by concatenating x and y; we refer to x as a *prefix* of z and y as a *suffix* of z
L^k	Set of all strings of a fixed length: $\left\{ l \;\middle\vert\; l \in L^* \wedge l = \langle l_0, l_1, \dots, l_{k-1} \rangle \right\}$
$X \uplus Y$	Disjoint union of sets: $\{\mathrm{inl}\, x \mid x \in X\} \cup \{\mathrm{inr}\, y \mid y \in Y\}$
2^X	Power set: $\{Y \mid Y \subseteq X\}$
$\underline{i}$	Ordinal: $\{0, 1, \dots, i-1\}$, $i \in \mathbb{N}$

2.2 Labelled Transition Systems

The notion of a labelled transition system (LTS) is a central conceptual tool used to capture semantic models in this thesis. An LTS is a collection of states, and (labelled) transitions relating pairs of states:

Definition 2.1 (LTS)**.**
An LTS is a 3-tuple: $(S, L, \rightarrow)$*, where* S *is the set of states,* L *the set of labels and* $\rightarrow \subseteq S \times L \times S$ *is the labelled transition relation. As standard, we write*

$$(x, l, y) \in \rightarrow \text{ as } x \xrightarrow{l} y.$$

Definition 2.2 (Finite LTS)**.**
An LTS, $(S, L, \rightarrow)$*, is finite if both* S *and* L *are finite.*

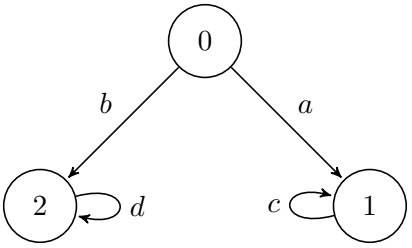


Figure 2.1: Example LTS

Example 2.1.
An example finite LTS is illustrated in Fig. 2.1, with $S = \{0, 1, 2\}$*,* $L = \{a, b, c, d\}$*, and* $\rightarrow = \big\{(0, a, 1), (0, b, 2), (1, c, 1), (2, d, 2)\big\}$*.*

The set of finite *runs* of an LTS consists of all finite strings of transitions where the states proceed from one transition to the next. The *trace* of a run is the sequence of labels of the underlying transitions:

Definition 2.3.
For an LTS, $(S, L, \rightarrow)$*, a run is a string over* $\rightarrow$*:*

$$\left\langle x_0 \xrightarrow{l_0} y_0,\ x_1 \xrightarrow{l_1} y_1,\ \ldots,\ x_{n-1} \xrightarrow{l_{n-1}} y_{n-1} \right\rangle$$

such that $x_i = y_{i-1}$*, for all* $1 \leq i < n$*. The trace of a run simply forgets the states, giving a corresponding sequence of labels:*

$$\langle l_0, l_1, \ldots, l_{n-1} \rangle$$

Example 2.2.
The set of traces of the LTS in Fig. 2.1 is the (infinite) set:

$$\big\{ \langle\rangle, \langle a \rangle, \langle b \rangle, \langle b, d, d, \ldots, d \rangle, \langle a, c, c, \ldots, c \rangle \big\}$$

In this thesis we frequently use LTSs that have pairs of fixed-length binary strings as labels, which we refer to as 2-LTSs, or $(k,\ l)$-LTSs if k and l are the binary string lengths.

Definition 2.4.
An $(k,\, l)$-LTS is an LTS, $(S, L, \rightarrow)$, where $L \subseteq \mathbb{B}^k \times \mathbb{B}^l$, where we write

$$x/y \text{ as syntactic sugar for } (x, y) \in L$$

Example 2.3.
Two example 2-LTSs are illustrated in Fig. 2.2.

(a) $(1,\, 2)$-LTS, X (b) $(2,\, 1)$-LTS, Y

Figure 2.2: Example 2-LTSs

As a notational shorthand, we often write 2-LTS labels containing one or more '$*$' characters, to represent the *set* of transitions where each '$*$' is expanded into a pair of transitions, with a 1 and 0 at that position in the label. For example, the label $*1/*$ corresponds to the set of labels $\{01/0, 01/1, 11/0, 11/1\}$ for a $(2,\, 1)$-LTS. When drawing LTSs, we often collapse multiple transitions between the same source/target states, drawing a single transition with a *set* of labels, which should be understood as a set of singly-labelled transitions. These labelling shorthand are illustrated in Fig. 2.3, where we give two presentations of the same LTS.

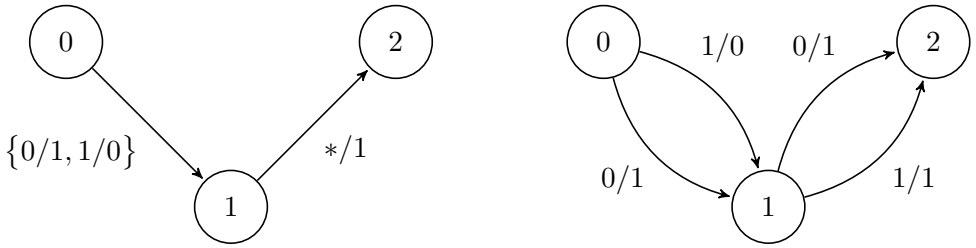


Figure 2.3: Compact vs Expanded label notation

When considering LTSs we often want to ignore the particular label and state names and instead concentrate on the transition connections, allowing us to identify those LTSs with the same underlying *structure*. We say that LTSs that can be identified in this way are *isomorphic*.

2.2.1 Isomorphism of LTSs

First, we describe how to map one LTS's transition structure onto another, known as a *homomorphism* between LTSs[1]:

Definition 2.5.
Consider two LTSs, $(S_1, L, \rightarrow_1)$ *and* $(S_2, L, \rightarrow_2)$*. A mapping,* f*, between the LTSs is a mapping on their states:* $f : S_1 \rightarrow S_2$*. A mapping is said to be an LTS homomorphism if for every* $x \xrightarrow{l}_1 y$ *we have* $f(x) \xrightarrow{l}_2 f(y)$*.*

An LTS homomorphism, f, is an isomorphism iff f is a bijection and we have $f(x) \xrightarrow{l}_2 f(y)$ iff we have $x \xrightarrow{l}_1 y$, i.e. there is a bijection between the transitions that respects the action of f on states. We write $X \cong Y$ when there exists an LTS isomorphism between X and Y.

2.2.2 2-LTS Compositions

We define two composition operations on 2-LTSs, synchronous and tensor, both of which are modifications of the cartesian product construction on LTSs. Later in the thesis, we will use these compositions to give *compositional* semantics of Petri Nets With Boundaries components.

First, we define the cartesian product construction on LTSs, which takes two LTSs and generates a new LTS, with the set of states (transitions) being the cartesian product of the underlying sets of states (transitions).

Definition 2.6 (Cartesian Product of LTSs)**.**
Given two LTS, X_1*:* $(S_1, L_1, \rightarrow_1)$*, and* X_2*:* $(S_2, L_2, \rightarrow_2)$ *their cartesian product is:* $(S_1 \times S_2, L_1 \times L_2, \rightarrow)$*, where*

$$(x, y) \xrightarrow{(\sigma_1, \sigma_2)} (x', y') \text{ iff } x \xrightarrow{\sigma_1}_1 x' \wedge y \xrightarrow{\sigma_2}_2 y'$$

Synchronous composition is defined for pairs of 2-LTSs that share a common "internal" label size; it restricts transitions in the product to those that agree on the shared label:

Definition 2.7 (Synchronous Composition)**.**
Given a (k, l)*-LTS* X *and a* (l, m)*-LTS* Y*, their synchronous composition, written* $X ; Y$*, is the* (k, m)*-LTS* $(S, L, \rightarrow_{sync})$*, where* $(S, L, \rightarrow)$ *is the cartesian product of* X *and* Y*, and* $\rightarrow_{sync}$ *is defined:*

$$(x, y) \xrightarrow[sync]{l_1/l_4} (x', y') \iff \exists\, l_2, l_3 \,.\, (x, y) \xrightarrow{(l_1/l_2,\, l_3/l_4)} (x', y') \wedge l_2 = l_3$$

[1] A more general definition of LTS homomorphism allows relabelling of transitions. However, we do not (and indeed, can not, since we must preserve labels to preserve compositions: Defn. 2.7) use such a generalisation in this thesis.

That is, we remove transitions whose labels do not agree on the l-sized internal boundary; furthermore, we hide the shared label component in the resulting transition. We sometimes write states of a synchronous composition as $(x\ ;\ y)$ rather than (x, y).

Remark 2.8. For a $(k,\ k)$-LTS X, we write X^m for the (left-associated) m-fold synchronous composition $X\ ;\ X\ ;\ \ldots\ ;\ X$.

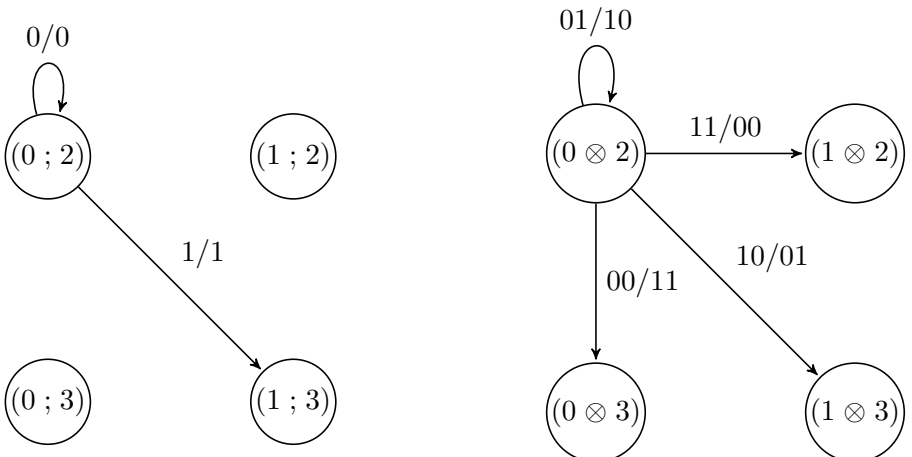
As we have seen, synchronous composition is only defined when the internal boundary sizes match up. Tensor composition, however, is defined for any pair of 2-LTSs, regardless of their boundary sizes:

Definition 2.9 (Tensor Composition)**.**
Given a $(k,\ l)$-LTS X and a $(m,\ n)$-LTS Y their tensor composition, written $X \otimes Y$, is a $(k + m,\ l + n)$-LTS, defined as $(S, L, \rightarrow_{tens})$, where $(S, L, \rightarrow)$ is their cartesian product and $\rightarrow_{tens}$ is defined:

$$(x, y) \xrightarrow{l_1 l_3 / l_2 l_4}_{tens} (x', y') \iff (x, y) \xrightarrow{(l_1/l_2,\ l_3/l_4)} (x', y')$$

In other words, pairwise concatenation of the labels of the cartesian product. Again, we sometimes write states of the tensor composition as $(x \otimes y)$ rather than (x, y).

Example 2.4.
An example of synchronous and tensor composition is given in Fig. 2.4, where we compose the LTSs X and Y from Fig. 2.2. Note that while we have drawn $(0\ ;\ 3)$ and $(1\ ;\ 2)$ in Fig. 2.4a, in general we will save space and omit such unconnected states.



(a) $(1,\ 1)$-LTS, $X\ ;\ Y$ (b) $(2,\ 2)$-LTS, $X \otimes Y$

Figure 2.4: Example 2-LTS compositions

Given a finite LTS, we can consider only its runs that start and finish in specified states. To do so, we identify a certain state as being *initial* and a collection of states as being *accepting* and then only consider runs of the LTS that start with the initial state and finish with an accepting state. This, of course, is the familiar concept of a non-deterministic finite automaton (NFA).

2.3 Non-deterministic Finite Automata

In this thesis, we will use NFAs to give the semantics of *marked* Petri Nets, those nets with certain initial and target markings. The initial marking will be the initial state of the NFA and the target marking will be the one accepting state of the NFA.

Definition 2.10 (NFA)**.**
A non-deterministic finite automaton is 5-tuple: $(Q, \Sigma, \rightarrow, i, A)$ *where* $(Q, \Sigma, \rightarrow)$ *is an LTS,* $i \in Q$ *and* $A \subseteq Q$*.*

Graphically, initial states have a small, unlabelled arrow pointing into them, and accepting states are drawn as double circles.

The *language* of an NFA is the traces of its runs that begin with the initial state, and end with an accepting state. We refer to individual traces in the language as *words*:

Definition 2.11.
Given an NFA, N*, its language, written* $\mathcal{L}(N)$*, is the set of all traces (i.e.* $\langle \sigma_0, \sigma_1, \ldots, \sigma_{n-1} \rangle$*, written as* $\vec{\sigma}$*) corresponding to runs of the underlying LTS:*

$$\left\langle x_0 \xrightarrow{\sigma_0} y_0,\ x_1 \xrightarrow{\sigma_1} y_1,\ \ldots,\ x_{n-1} \xrightarrow{\sigma_{n-1}} y_{n-1} \right\rangle$$

such that $x_0 = i$ *and* $y_{n-1} \in A$*. By Defn. 2.3, we must have* $x_i = y_{i-1}$*, for all* $1 \leq i < n$*.*

Example 2.5.
An example NFA is illustrated in Fig. 2.5. Its language is the (infinite) set:

$$\{tmw \mid w \in \{s\}^*\} \cup \{cmw \mid w \in \{s\}^*\}$$

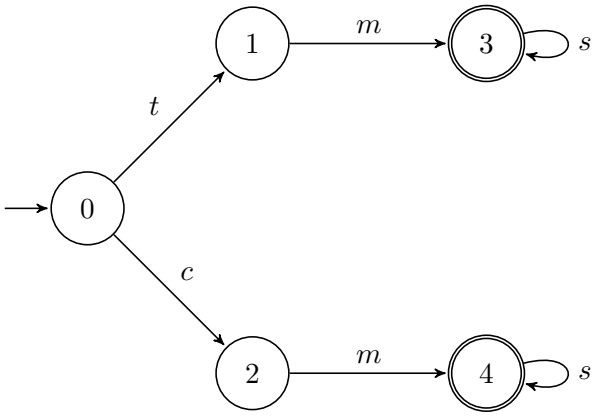


Figure 2.5: An example NFA

NFAs may have different structure, yet recognise the same language; we say that such NFAs are *language-equivalent*:

Definition 2.12.
N *and* M *are language-equivalent, written* $N \sim_{\mathcal{L}} M$*, if* $\mathcal{L}(N) = \mathcal{L}(M)$*.*

Example 2.6.
We have that $N \sim_{\mathcal{L}} M$, where N is illustrated in Fig. 2.5, and M in Fig. 2.6.

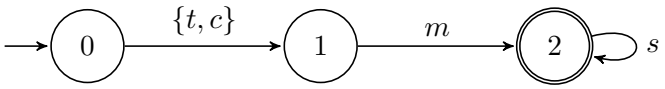


Figure 2.6: A NFA that is language equivalent to that of Fig. 2.5

2.3.1 Isomorphism of NFAs

A homomorphism between two NFAs, N_1 and N_2, is a homomorphism on the underlying LTSs that additionally preserves the initial and accepting states.

Definition 2.13.
Given two NFAs, N_1: $(Q_1, \Sigma_1, \rightarrow_1, i_1, A_1)$ and N_2: $(Q_2, \Sigma_2, \rightarrow_2, i_2, A_2)$, a homomorphism, f (comprised of components $f_S : Q_1 \rightarrow Q_2$ and $f_L : \Sigma_1 \rightarrow \Sigma_2$) on the underlying LTSs $(Q_1, \Sigma_1, \rightarrow_1)$ and $(Q_2, \Sigma_2, \rightarrow_2)$, is a NFA homomorphism if the following additional conditions are satisfied:

(i) $f_S(i_1) = i_2$

(ii) $\{f_S(x) \mid x \in A_1\} = A_2$

Clearly, if an NFA-homomorphism is an LTS-isomorphism, it is also a NFA-isomorphism.

2.3.2 2-NFAs

If the underlying LTS of an NFA N is a (k, l)-LTS we say that N is a (k, l)-NFA, or, if the particular k, l are unimportant, a 2-NFA.

We define composition operations on 2-NFAs in a similar manner to how we defined them on 2-LTSs: the underlying operation is performed on the 2-LTS, and the initial/accepting states are suitably modified.

Definition 2.14 (Synchronous composition of 2-NFAs)**.**
Given a (k, l)-NFA, N: $(Q_1, \Sigma_1, \rightarrow_1, i_1, A_1)$ and a (l, m)-NFA, M: $(Q_2, \Sigma_2, \rightarrow_2, i_2, A_2)$ their synchronous composition is a (k, m)-NFA:

$$(Q, \Sigma, \rightarrow, (i_1, i_2), A_1 \times A_2)$$

where $(Q, \Sigma, \rightarrow)$ is the synchronous composition of the underlying 2-LTSs (Defn. 2.7).

Definition 2.15 (Tensor composition of 2-NFAs)**.**
Given a $(k,\, l)$*-NFA,* $N\colon (Q_1, \Sigma_1, \to_1, i_1, A_1)$ *and a* $(m,\, n)$*-NFA,* $M\colon (Q_2, \Sigma_2, \to_2, i_2, A_2)$ *their tensor composition is a* $(k+m,\, l+n)$*-NFA:*

$$(Q, \Sigma, \to, (i_1,\, i_2)\,, A_1 \times A_2)$$

where $(Q, \Sigma, \to)$ *is the tensor composition of the underlying 2-LTSs (Defn. 2.9).*

Now, we turn to Petri nets, the semantics of which we give using LTSs/NFAs.

2.4 Petri nets

Petri nets are a mathematical model with a vivid graphical presentation, used to model concurrent and distributed systems. A Petri net consists of a set of places, and a set of transitions that connect sets of places to sets of places.

Definition 2.16 (Petri net)**.**
A Petri net is a 4-tuple: $(P, T, {}^\circ{-}, {-}^\circ)$*, where:*

- P *is the set of places,*
- T *is the set of transitions,*
- ${}^\circ{-},\ {-}^\circ : T \to 2^P$ *give, respectively, the pre- and post-sets of each transition.*

For $t \in T$, its preset, ${}^\circ t$ is the set of places that connect to t, and its postset, t°, is the set of places to which t connects. The *neighbourhood* of t is ${}^\circ t^\circ \stackrel{\text{def}}{=} {}^\circ t \cup t^\circ$. We write $\mathrm{places}(N)$ and $\mathrm{trans}(N)$ respectively, for the place and transition sets of N.

In this thesis, we consider the class of Petri nets where each place can only contain zero or one *token*. Other classes of Petri nets (e.g. P/T nets [136]) lift this restriction, but we do not use this generalisation, instead focussing on *1-bounded*, or *safe* nets. The definitions in this chapter essentially follow those of [137], albeit with a slightly different presentation. For example, we use a functional, rather than relational presentation of transition connections; this alternative presentation makes the extension of Petri nets to Petri nets with boundaries clearer (indeed, every Petri net is a degenerate Petri net with boundaries).

Definition 2.17 (Marking)**.**
A marking of a net N *is a function* $m : \mathrm{places}(N) \to \{0, 1\}$*.*

Graphically, a marking is illustrated by drawing black circles inside each place that is assigned a token by the marking.

Example 2.7.
An example Petri net, N, is illustrated in Fig. 2.7, with 7 places, 5 transitions, and with the marking function: $\mathrm{m}(p_i) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } i \leq 1 \\ 0 & \text{otherwise} \end{cases}$

In N, for example, ${}^\circ t_4 = \{p_2, p_4\}$, ${}^\circ t_1 = \{p_1\}$ *and* $t_1^\circ = \{p_2, p_5\}$.

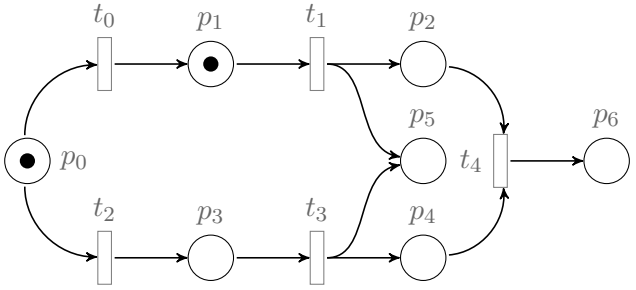


Figure 2.7: Example Petri net

The marking of a Petri net is transformed by *firing* transitions; we say one marking is *reachable* from another if there is a sequence of transition firings that transforms one marking to the other. Intuitively, when fired, a transition, t, consumes tokens from all $p \in {}^\circ t$ and produces a token at all $q \in t^\circ$, giving a new marking.

However, it is not the case that arbitrary subsets of net transitions can necessarily be fired at the same time. Two transitions cannot fire simultaneously if they attempt to produce a token at the same place (e.g. t_1 and t_3 in Fig. 2.7), or both consume the token from a place, (e.g. t_0 and t_2 in Fig. 2.7). To specify the transitions that may fire concurrently, we define the notion of *contention* between transitions.

Definition 2.18 (Transition contention)**.**
For a Petri net N, we say that $t, u \in \mathrm{trans}(N)$ are in contention (written $t \bowtie u$) when ${}^\circ t \cap {}^\circ u \neq \varnothing$ or $t^\circ \cap u^\circ \neq \varnothing$.

Note that contention is a property of the net's topology and does *not* depend on any particular marking. We say a set of transitions is *contention-free* if it contains no pair of *distinct* transitions in contention:

Definition 2.19 (Contention-free transition sets)**.**
For a Petri net N, we say that $S \subseteq \mathrm{trans}(N)$ is contention-free if $\forall t, u \in S$, we have:

$$t \bowtie u \implies t = u$$

For example, in the net shown in Fig. 2.7, $\{t_0, t_3\}$ is contention-free, but $\{t_0, t_2\}$ is not.

A transition may only be fired if it is *enabled*: when each place of its pre-set is marked, and no places in its post-set are marked:

Definition 2.20 (EN enabled transitions)**.**
For a net N*,* $t \in \mathrm{trans}(N)$ *is enabled for a marking* m*, written enabled*$_m(t)$*, if for all* $p \in {}^\circ t$*,* $\mathrm{m}(p) = 1$ *and for all* $q \in t^\circ$*,* $\mathrm{m}(q) = 0$*.*

Remark 2.21. A trivial observation is that a particular transition will be enabled at some, but not all, markings of the same net, whereas a given pair of transitions will either always be in contention, or never. Put differently, contention is a static property of the net, while a particular transition being enabled is a *dynamic* property of the net. Determining if we are able to fire a particular set of transitions thus relies on a mixture of static and dynamic information: we must have no transitions in contention in the set, and the current marking should enable all of the transitions. In some net models, such as P/T nets, the enabled condition of Defn. 2.20 is relaxed, allowing transitions to fire even if a token is already present on the place. However, in this thesis we do not consider such nets: their semantics are more complex, and their statespaces are often infinite.

We may lift several of the previous definitions from individual transitions to sets of transitions, in a straightforward manner. In the following, $S \subseteq \mathrm{trans}(N)$, for some net, N:

$$
\begin{aligned}
{}^\circ S &\stackrel{\text{def}}{=} \bigcup_{t \in S} {}^\circ t \\
S^\circ &\stackrel{\text{def}}{=} \bigcup_{t \in S} t^\circ \\
\textit{enabled}_m(S) &\stackrel{\text{def}}{=} \bigwedge_{t \in S} \textit{enabled}_m(t)
\end{aligned}
$$

Remark 2.22. Before continuing, we briefly mention that the class of all Petri nets can be sub-divided based upon either structure, token numbers and transition firing rules. Common net classes relating to structure are *pure*, *simple* or *finite* nets, to tokens and structure are *bounded* or *safe* nets and to transition firing rules are *elementary* nets. In turn, we have that a Petri net, N is:

- *pure* if, for any $t \in \mathrm{trans}(N)$, ${}^\circ t \cap t^\circ = \varnothing$
- *simple* if, for any $t, u \in \mathrm{trans}(N)$, ${}^\circ t = {}^\circ u \implies t = u$,
- k-*bounded* if, every reachable marking, m, has that: $\forall p \in \mathrm{places}(N), \mathrm{m}(p) \leq k$,
- *safe* if it is 1-bounded,
- *finite* if both $\mathrm{places}(N)$ and $\mathrm{trans}(N)$ are finite,

We may now define the *step firing* semantics of a Petri net, in which (sets of) transitions are fired: a set of transitions may fire if it is both contention-free (a static property) and enabled in the current marking of the net (a dynamic property):

Definition 2.23 (EN Firing Semantics)**.**
Given a net N and a marking m, a set $S \subseteq \mathrm{trans}(N)$ can fire iff S is contention-free and enabled$_m(S)$ (by Defn. 2.20). Firing S creates a new marking m', defined thus:

$$\mathrm{m'}(p_i) \stackrel{\mathrm{def}}{=} \begin{cases} 0 & \text{if } p_i \in {}^\circ S \\ 1 & \text{if } p_i \in S^\circ \\ \mathrm{m}(p_i) & \text{otherwise} \end{cases}$$

that is, tokens are consumed from all places in the pre-set of S, produced at all places in the post-set of S, and otherwise unchanged from m. If S can be fired to transform N from m to m', we write $N_m \rightarrow_S N_{m'}$.

For the remainder of this thesis, we only consider finite, pure and simple (Remark 2.22) Elementary Nets [138], which are defined in Defn. 2.24.

Definition 2.24 (Elementary Net)**.**
An elementary net (EN) is a Petri net, $(P, T, {}^\circ-, -^\circ)$, that uses the firing rule in Defn. 2.23.

Since ENs do not allow transitions to be enabled when there are tokens in any of their post-set places, safeness (Remark 2.22) is guaranteed. As a consequence, the state space—set of reachable markings—of an EN N is *finite* since the markings of N put at most one token in each place. Furthermore, markings can be identified with the sets of places $M \subseteq \mathrm{places}(N)$ they characterise.

In standard presentations, an EN N must have no unconnected places (i.e $\forall p \in \mathrm{places}(k)$, $\exists t \in \mathrm{trans}(N)$ s.t. $p \in {}^\circ t^\circ$) or transitions (i.e $\forall t \in \mathrm{trans}(k), {}^\circ t^\circ \neq \varnothing$). However in this thesis we lift this restriction, allowing transitions to not connect to places, as will be discussed when introducing Petri Nets With Boundaries.

2.4.1 Labelled transition system semantics of Petri Nets

The firing semantics of a Petri net can be used to construct an LTS—known as its *reachability graph* [5]—representing the relationships between all markings of the net.

The LTS semantics for a net N has states the markings of N, with transitions between markings m and m' being labelled by sets of (enabled, contention-free) transitions:

Definition 2.25 (LTS semantics of a Petri net)**.**
The LTS semantics of a net N is defined to be the LTS $(2^{\mathrm{places}(N)}, 2^{\mathrm{trans}(N)}, \mathit{fired})$, where $(m, S, m') \in \mathit{fired}$ iff S is contention-free and enabled by m, and $N_m \rightarrow_S N_{m'}$.

Example 2.8.
The LTS semantics of the Petri net of Fig. 2.7 is shown in Fig. 2.8. Recall that net markings are identified with the set of net states to which they assign a token, and

that LTS states correspond to markings of the underlying net, therefore, we draw LTS states containing sets of net states. The LTS transitions are labelled with sets of net transitions, fired to transform between the two markings. Note that each state of the LTS has a self-loop: the empty set of transitions can always be fired. For clarity, the 122 (out of 128 total) states that are not reachable from the illustrated marking $\{p_0, p_1\}$ *have not been drawn.*

For example, we have that $\{p_0, p_1\} \xrightarrow{\{t_1,t_2\}} \{p_2, p_3, p_5\}$ *and* $\{p_1, p_3\} \xrightarrow{\varnothing} \{p_1, p_3\}$*. Observe that the net transition* t_4 *doesn't appear in the label of any LTS transition - it can never be fired since no reachable marking places a token in both* p_2 *and* p_4*.*

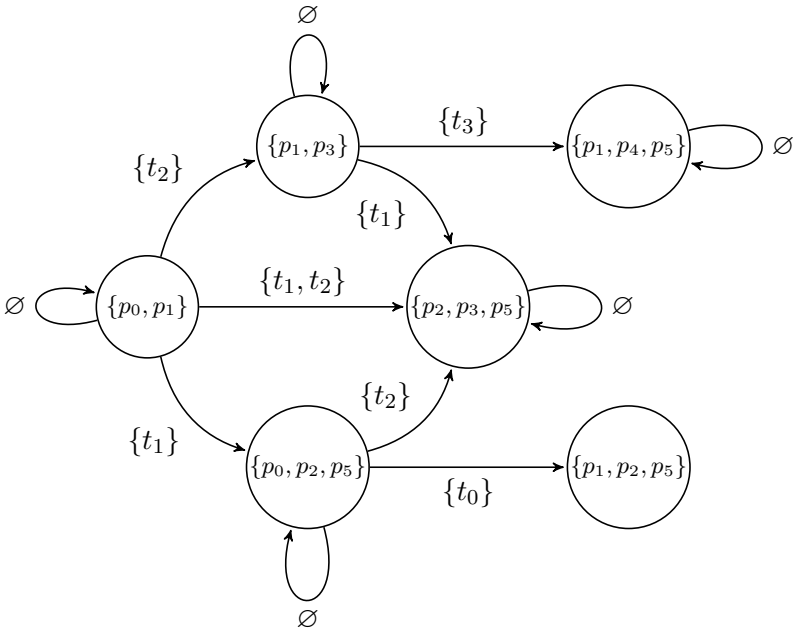


Figure 2.8: LTS semantics of the Petri net in Fig. 2.7.

Finally, as a consequence of the definition of Petri net LTS semantics, we observe that trace membership of an LTS semantics implies a valid corresponding sequence of firings in the underlying net:

Remark 2.26. Consider a net N and its LTS semantics X. Any trace, $t = \langle l_0, l_1, \ldots, l_n \rangle$, of X is a valid sequence of firings of N, between the markings corresponding to the states of *any* run with trace t.

2.5 Reachability and Coverability in Petri Nets

A natural question to ask about a Petri net is whether it is possible to reach a particular target marking from some initial marking. That is, does there exist a firing sequence that transforms the net from the initial marking to the target marking. This question is known as the *reachability problem* [139], and is PSPACE-complete [11].

Due to the correspondence between Petri net firing sequences and traces of their LTS semantics (Remark 2.26), reachability checking coincides with checking the existence of a trace between the states corresponding to the initial and target markings. For example, there is a trace between $\{p_0, p_1\}$ and $\{p_1, p_4, p_5\}$ in Fig. 2.8.

We often want to consider a particular net and pair of its markings, and refer to such a triple as a *marked* net:

Definition 2.27 (Marked Net)**.**
A marked net, (N, i, t), is comprised of a Petri net N and the initial/target markings: $i, t \subseteq \text{places}(N)$.

A related concept from the literature is a *net system* [137], which is simply a net along with a chosen initial marking.

Graphically, we represent a marked net as a Petri net with tokens on each place in the initial marking, and shading on each place in the target marking. For example, consider the net illustrated in Fig. 2.9, representing a fork-join system containing 2 tasks; the initial marking is $\{p_0\}$ (representing the initial, ready state of the system) and the target marking is $\{p_3, p_4\}$, where both tasks have finished, but have not been joined.

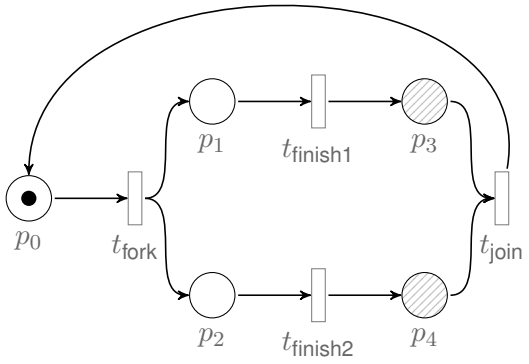


Figure 2.9: Marked Petri net representing a fork-join system

Given a marked Petri net and the LTS semantics of the net, we can construct a NFA, recognising the set of traces between the pair of states:

Definition 2.28 (NFA Semantics of a marked Petri Net)**.**
Given a marked net, (N, i, t), the NFA semantics extends the LTS semantics: the initial state is i and the set of accepting states is the singleton set $\{t\}$.

Example 2.9.
Consider the LTS shown in Fig. 2.8, representing the semantics of the net in Fig. 2.7. Take the initial state to be the original marking shown in Fig. 2.7, i.e. $\{p_0, p_1\}$, and the accepting states to be $\{\{p_2, p_3, p_5\}\}$. Now, the language of the NFA, N, obtained from

the LTS and chosen initial/accepting states is simple:

$$\mathcal{L}(N) = \left\{ \langle \{t_1, t_2\} \rangle, \langle \{t_2\}, \{t_1\} \rangle, \langle \{t_1\}, \{t_2\} \rangle \right\}$$

that is, we can either fire t_2 and t_1 in order, or concurrently, to reach the desired marking.

A simple observation following from Remark 2.26 is that a trace being a member of the language of an NFA for a particular reachability problem implies that the reachability problem is satisfiable:

Remark 2.29. If $w \in \mathcal{L}(N)$, where N is the NFA representing the reachability problem for a given marked net, then w is a *witness* of the final marking being reachable from the initial marking in the net. Generally, if $\mathcal{L}(N) \neq \varnothing$, then the reachability problem *is* satisfiable, whereas if $\mathcal{L}(N) = \varnothing$, then the reachability problem is *not* satisfiable.

Finally, we close this section by giving the definition of a slight modification of the reachability problem, that of *coverability*. Coverability of a marking, m, checks if any marking that *contains* m as a *sub-marking* is reachable:

Definition 2.30 (Sub-marking)**.**
Given a Petri net, N, and two of its markings, m and m', we say that m is a sub-marking of m', written $m \leq m'$, iff $\forall p \in m, q \in m'$.

Definition 2.31 (Coverability)**.**
Given a marked Petri net, (N, i, t), we say that t is coverable if $\exists m \,.\, t \leq m$ such that the reachability problem (N, i, m) is satisfiable.

2.6 Petri nets with boundaries

Petri nets with boundaries (PNBs) extend the definition of the EN nets introduced in the previous section by adding left and right boundaries, to which transitions of the net can also connect. PNBs are composed using two operations: a synchronising composition (an intuitive example is illustrated in Fig. 2.10; the graphical notation will be explained shortly), and a non-interacting, parallel composition. Indeed, by allowing transitions to connect to boundary ports, their behaviour can be *partially* specified; synchronously composing PNBs *completes* the specification of transitions by supplementing the places that the transition connects to.

Definition 2.32 (Petri net with Boundaries)**.**

A PNB is a 9-tuple: $(P, T, {}^\circ{-}, {-}^\circ, l, r, {}^\bullet{-}, {-}^\bullet, \bowtie)$, where:

- $(P, T, {}^\circ{-}, {-}^\circ)$ *is an EN,*
- $l, r \in \mathbb{N}$ *are respectively, the left and the right boundaries,*

- ${}^\bullet- : T \to 2^{\underline{l}}$ *and* $-^\bullet : T \to 2^{\underline{r}}$ *connect a transition to the left and right boundaries,*

- $\bowtie$ *is a contention relation (see Definition 2.33 below).*

We will refer to PNBs simply as nets herein; when we need to be precise we will explicitly name the type of nets being considered. As the definitions of ${}^\circ-, -^\circ$ were lifted to sets of transitions, so are ${}^\bullet-, -^\bullet$: for a PNB, N, and $U \subseteq \mathrm{trans}(N)$:

$$ {}^\bullet U \stackrel{\text{def}}{=} \bigcup_{t \in U} {}^\bullet t $$

and similarly for $U^\bullet$. We frequently need to state the boundaries of a given PNB, writing $N : (k, l)$ to indicate that N has left and right boundaries k and l, respectively.

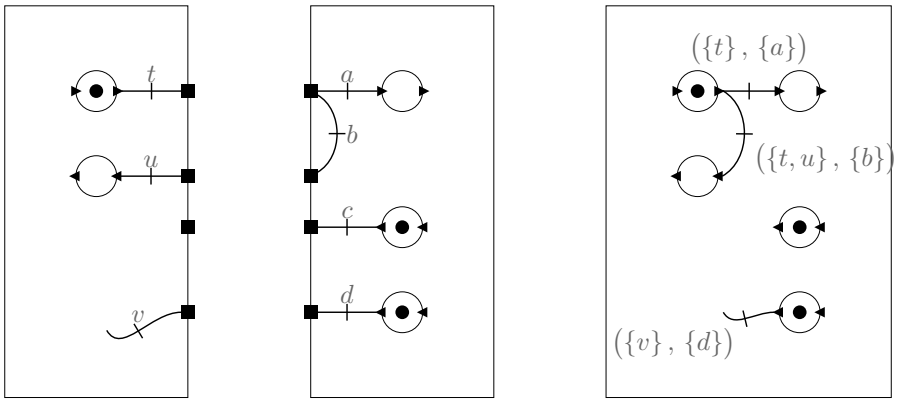


Figure 2.10: Example Synchronous Composition

2.6.1 Graphical Notation

PNBs are represented using an alternative to the classic Petri net graphical notation. The alternative notation is lighter weight than the classic notation, and is more intuitive when reasoning about and presenting PNB compositions.

In the alternative graphical representation, each place is drawn as *directed*, having an *in* and *out* port, drawn as a triangle pointing into and out of the place, respectively. Transitions are undirected links that connect an arbitrary set of boundary and place ports. The standard graphical presentation of Petri nets has directed arcs, where the direction is either “place to transition” or “transition to place”, as determined by the pre- and post-sets of each transition.

With our alternative notation, then, the pre-set of a transition is just the set of places to which the transition is connected via the *out* port, symmetrically, its post-set is the set of places to which the transition is connected via the *in* port.

In order to distinguish individual transitions and increase legibility, individual transitions are drawn with a small perpendicular mark.

We demonstrate the graphical notation in Fig. 2.11, which represents the same Petri net as that of Fig. 2.7.

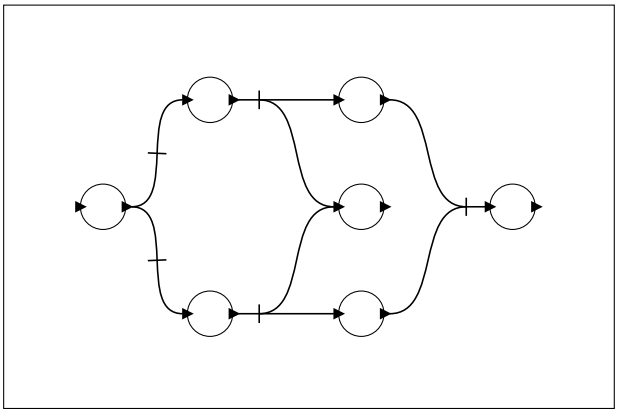


Figure 2.11: PNB representation of the Petri net in Fig. 2.7.

To emphasise the "internal" structure of a PNB, we often draw transitions (or parts thereof) that connect to boundary ports in a lighter colour; of course, formally there is no distinction, and therefore lighter sections can always be ignored. Additionally, where transitions cross, we sometimes use different patterns to make the structure of the transitions clearer.

Though we are yet to define composition of PNBs, we can give a graphical intuition, which we will make precise later in this chapter: synchronous composition corresponds to fusing the *inner* boundary ports of two PNBs, suitably synchronising the transitions connecting to those boundary ports. Parallel composition, on the other hand, corresponds to stacking two PNBs upon one another without fusing transitions.

2.6.2 PNB Firing Semantics

As for explained in Remark 2.21, transitions in contention cannot be fired concurrently; in Petri nets, two transitions are in contention precisely when they compete for a resource, i.e. they consume a token from, or produce a token at the same place.

For PNBs, there are *two* additional sources of contention between transitions; essentially, if the two transitions connect to the same boundary port, or were created by *fusing* multiple transitions that connected to the same boundary port, then they should be in contention. For now, we outline the definition of a contention relation, but defer discussion of its role in composition until §2.6.3. Note that a contention relation *must* contain all pairs of transitions that have structural contention (i.e. connecting to a common boundary/place port), but *may* also contain additional pairs, as in Exm. 2.11.

Definition 2.33 (Contention Relation)**.**
For a net, N, a symmetric relation, $\bowtie$, on $\mathrm{trans}(N)$ is said to be a contention relation, if for all $(t, u) \in \mathrm{trans}(N) \times \mathrm{trans}(N)$, $t \neq u$, when any of the following hold:

(i) ${}^\circ t \cap {}^\circ u \neq \varnothing$,

(ii) $t^\circ \cap u^\circ \neq \varnothing$,

(iii) ${}^\bullet t \cap {}^\bullet u \neq \varnothing$,

(iv) $t^\bullet \cap u^\bullet \neq \varnothing$.

then $t \bowtie u$.

We can lift contention from individual transitions to sets of transitions; indeed, abusing notation, we write $U \bowtie V$ for $U, V \subseteq \mathrm{trans}(N)$, if $t \bowtie u$ for some $t \in U$, $u \in V$.

We define contention-free (Defn. 2.19) and enabled (Defn. 2.20) *sets* of transitions in the same was as for Petri nets. Firing is also defined in the same way, though we repeat the definition here, for clarity:

Definition 2.34 (Firing Semantics for PNBs)**.**
Given a PNB N and a marking m, a set $U \subseteq \mathrm{trans}(N)$ can fire iff U is contention-free and is enabled by marking m. Firing U creates a new marking n, defined as follows:

$$\mathrm{n}(p_i) \stackrel{\mathrm{def}}{=} \begin{cases} 0 & \text{if } p_i \in {}^\circ U \\ 1 & \text{if } p_i \in U^\circ \\ \mathrm{m}(p_i) & \text{otherwise} \end{cases}$$

that is, tokens are consumed from all places in the pre-set of U, produced at all places in the post-set of U, and otherwise unchanged from m. If $U \subseteq \mathrm{trans}(N)$ is enabled and contention-free at m, and transforms the marking to n when fired, we write $N_m \rightarrow_U N_n$.

To illustrate the use of boundary ports, we make precise the synchronous composition of PNBs, which connects the boundary ports of suitable pairs of PNBs.

2.6.3 Synchronous composition of PNBs

In order to synchronously compose two PNBs, they must share a boundary with the same number of ports. Intuitively, composition forms a new net which has the places of both nets, and transitions that are formed of sets of transitions, one from each of the underlying nets. We often call nets that are to be composed *component nets*, or simply *components*.

To define composition, we require the notion of a *synchronisation* between two nets, which is a particular choice of transitions from the two components:

Definition 2.35 (Synchronisations)**.**
A synchronisation between two PNBs, $N : (k, l)$ *and* $M : (l, m)$ *is a pair*

$$(U, V) \in 2^{\mathrm{trans}(N)} \times 2^{\mathrm{trans}(M)}$$

of contention-free sets of transitions, such that $U^\bullet = {}^\bullet V$*.*

Synchronisations inherit an ordering from the subset ordering, pointwise:

$$(U, V) \subseteq (U', V') \stackrel{\mathrm{def}}{=} U \subseteq U' \wedge V \subseteq V'$$

We call the empty synchronisation, $(\varnothing, \varnothing)$, *trivial*. A synchronisation (U, V) is *minimal* when it is not trivial, and for all synchronisations $(U', V') \subseteq (U, V)$, then (U', V') is trivial or equal to (U, V). Intuitively, minimal synchronisations are the smallest collections of transitions from both nets that share a common set of boundary ports; put differently, we only retain those transitions that *must* fire together, to synchronise the two nets.

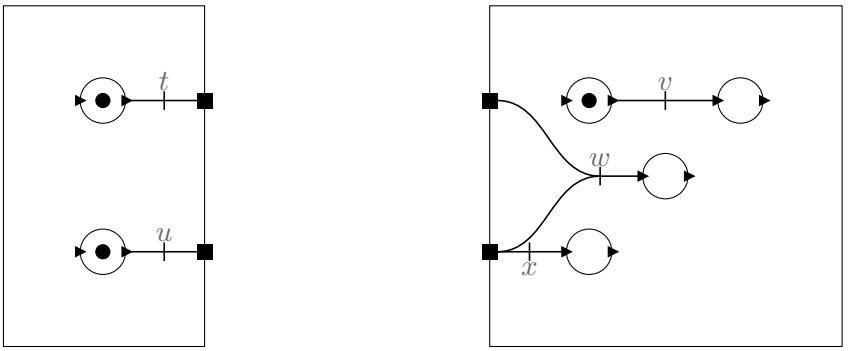


Figure 2.12: Components to be composed

Example 2.10.
Consider the two nets in Fig. 2.12; we have the following:

- $\left(\{t\}, \varnothing\right)$ *is not a synchronisation since* $\{t\}^\bullet = \{0\} \neq \varnothing = {}^\bullet\varnothing$*,*
- $\left(\{u\}, \{v, x\}\right)$ *and* $\left(\{t, u\}, \{v, w\}\right)$ *are synchronisations, but are not minimal,*
- $\left(\{u\}, \{x\}\right)$*,* $\left(\{t, u\}, \{w\}\right)$ *and* $\left(\varnothing, \{v\}\right)$ *are all minimal synchronisations.*

Indeed, since we wish to use minimal synchronisations as the transitions of composed PNBs, we require the notion of contention between synchronisations:

Definition 2.36 (Contention between synchronisations)**.**
Contention is lifted to synchronisations between N *and* M*:*

$$(U, V) \bowtie (U', V') \stackrel{\mathrm{def}}{=} U \bowtie_N U' \vee V \bowtie_M V'$$

where $\bowtie_X$ *is the contention relation of net* X*.*

For nets $N : (k,\, l)$ and $M : (l,\, m)$, let $\mathrm{minsync}(N,\, M)$ be the set of minimal synchronisations between them.

Remark 2.37. For any $(U, V) \in \mathrm{minsync}(N,\, M)$, we have that $|U| > 1$, or $|V| > 1$ only if there exists a transition in N or M that is connected to more than one common boundary port. If this were not the case, (U, V) would not be a *minimal* synchronisation, since we could remove a transition while preserving the synchronisation property. Properties concerning composition of nets with transitions that are only connected to one boundary port are explored in Chapter 3, where we elucidate the categorical structure of PNBs. For each transition $t \in \mathrm{trans}(N)$ with $t^\bullet = \varnothing$, we have $(\{t\}, \varnothing) \in \mathrm{minsync}(N,\, M)$, and similarly for $\mathrm{trans}(M)$. In other words, transitions in the components are free to fire without affecting the other component as long as they do not connect to the common boundary. If they do, the transitions must synchronise with transitions in the other component respecting their interaction on the shared boundary.

We can now give the definition of synchronous composition of nets with boundaries:

Definition 2.38 (Synchronous composition)**.**
The composition of PNBs $N : (k,\, l)$ and $M : (l,\, m)$, is written $N \,;\, M : (k,\, m)$, and has the following structure:

- $\mathrm{places}(N \,;\, M)$ *is* $\mathrm{places}(N) \uplus \mathrm{places}(M)$*,*
- $\mathrm{trans}(N \,;\, M)$ *is* $\mathrm{minsync}(N,\, M)$*, the set of minimal synchronisations,*
- ${}^\circ(U, V) \stackrel{\text{def}}{=} {}^\circ U \uplus {}^\circ V$ *and* $(U, V)^\circ \stackrel{\text{def}}{=} U^\circ \uplus V^\circ$*,*
- ${}^\bullet(U, V) \stackrel{\text{def}}{=} {}^\bullet U$ *and* $(U, V)^\bullet \stackrel{\text{def}}{=} V^\bullet$*,*
- *Contention is lifted to minimal synchronisations, as described in Defn. 2.36, but is subtle, as we discuss in Remark 2.39.*

In order for the composition to be well-defined, lifting the contention relations from PNBs N and M as per Defn. 2.36 must yield a well-defined contention relation for $N \,;\, M$, which is confirmed in [61, Definition 3.4].

Remark 2.39. We require that contention is *explicitly* preserved by synchronous composition: if certain transitions are in contention in the component nets, transitions containing them in the nets' composition should also be in contention. Indeed, this is the reason for requiring an explicit contention relation, rather than relying on the connectivity of the transitions alone to determine contention. As we will demonstrate shortly (Exm. 2.11), certain PNBs, when composed, form transitions in the composite that should be in contention, but are not, structurally. The intuition being that since transitions in a composition are formed of sets of transitions of the components, we should not be able to fire transitions of the composition that are conflicting as transitions of the components. Further examples, and the mathematical foundations of contention are given in [140].

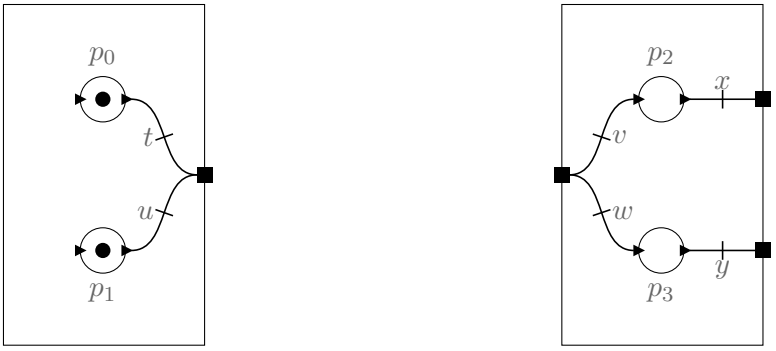
An example of synchronous composition is illustrated in Fig. 2.10. Note that there is no synchronisation containing c, it has no transition in N with which to synchronise, and that t appears in two synchronisations in $N \,;\, M$: synchronising both with a *and* b.

The two additional sources of contention relative to Petri nets, mentioned earlier in this section can now be stated precisely:

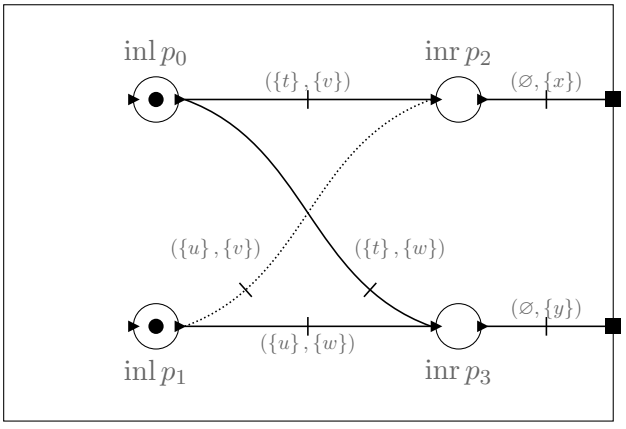
1. Connecting to the same boundary port leads to contention: Defn. 2.33(iii) and (iv)
2. Contention is preserved in compositions, by Defn. 2.36

We now demonstrate synchronous composition of two simple nets, showing why explicit contention relations are required, rather than inferring contention from PNB structure:

Example 2.11.
Consider the two component nets L in Fig. 2.13a and R in Fig. 2.13b and their composition in Fig. 2.13c. We have that $t \bowtie_L u$ and $v \bowtie_R w$.



(a) Component L. (b) Component R.



(c) $L \,;\, R$.

Figure 2.13: Example component nets and their synchronous composition

Consider the pair of transitions $(\{t\},\{v\})$ and $(\{u\},\{w\})$ in the composition. If contention was not remembered, these two transitions would not be in contention in the composite net (they share no common place/boundary ports), and could therefore be fired concurrently. However, the underlying sets of transitions in the components are *in*

contention (i.e. $\{t, u\}$ and $\{v, w\}$), since the transitions connect to the same boundary port and thus they cannot fire concurrently. Fortunately, in the examples we consider in this thesis, complications due to contention rarely play a prominent role.

Graphically, synchronous composition can be intuitively understood as fusing the transitions that are connected to the shared boundary ports in the components, duplicating transitions when there is a choice in the other component. For example, in Fig. 2.13a, t is connected to the first shared boundary port; in Fig. 2.13b there are two transitions connected to the first boundary port (v and w), thus in the composition, shown in Fig. 2.13c, there are two transitions formed from t.

2.6.4 Parallel composition of PNBs

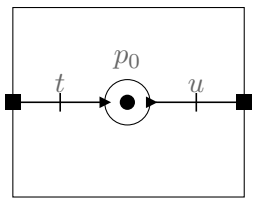
Whereas synchronous composition requires compatible component boundaries, intuitively to allow the right boundary of one component to be connected to the left boundary of the other, parallel composition has no such restriction, simply stacking the components on top of each other.

Definition 2.40 (Parallel composition)**.**
The parallel, or tensor, composition of PNBs $N : (k,\, l)$ and $M : (m,\, n)$, is written $N \otimes M : (k + m,\, l + n)$, and has the following structure:

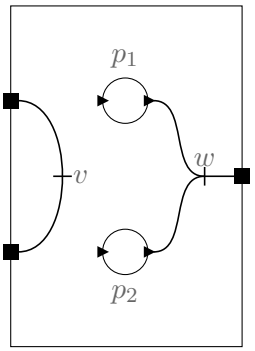
- $\mathrm{places}(N \otimes M) \stackrel{\text{def}}{=} \mathrm{places}(N) \uplus \mathrm{places}(M)$
- $\mathrm{trans}(N \otimes M) \stackrel{\text{def}}{=} \mathrm{trans}(N) \uplus \mathrm{trans}(M)$
- ${}^\circ(\mathrm{inl}\, t) \stackrel{\text{def}}{=} \{\mathrm{inl}\, p \mid p \in {}^\circ t\}$ *and* ${}^\circ(\mathrm{inr}\, t) \stackrel{\text{def}}{=} \{\mathrm{inr}\, p \mid p \in {}^\circ t\}$
- $(\mathrm{inl}\, t)^\circ \stackrel{\text{def}}{=} \{\mathrm{inl}\, p \mid p \in t^\circ\}$ *and* $(\mathrm{inr}\, t)^\circ \stackrel{\text{def}}{=} \{\mathrm{inr}\, p \mid p \in t^\circ\}$
- ${}^\bullet(\mathrm{inl}\, t) \stackrel{\text{def}}{=} {}^\bullet t$ *and* ${}^\bullet(\mathrm{inr}\, t) \stackrel{\text{def}}{=} \{b + k \mid b \in {}^\bullet t\}$
- $(\mathrm{inl}\, t)^\bullet \stackrel{\text{def}}{=} t^\bullet$ *and* $(\mathrm{inr}\, t)^\bullet \stackrel{\text{def}}{=} \{b + l \mid b \in t^\bullet\}$
- $\bowtie \stackrel{\text{def}}{=} \big\{(\mathrm{inl}\, t,\, \mathrm{inl}\, u) \;\big|\; (t,\, u) \in \bowtie_N\big\} \cup \big\{(\mathrm{inr}\, t,\, \mathrm{inr}\, u) \;\big|\; (t,\, u) \in \bowtie_M\big\}$

Unlike synchronous composition, transitions in a tensor composition are only in contention if they are in contention in one of the components. An example tensor composition is illustrated in Fig. 2.14.

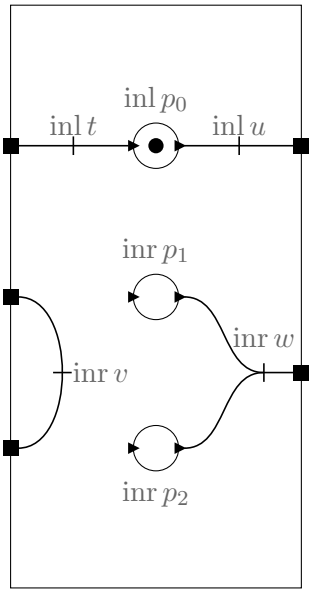
For convenience, and when there can be no confusion (e.g. when the places of the components are disjoint), we label places of a composite PNB as p instead of $\mathrm{inl}\, p$ or $\mathrm{inr}\, p$ and similarly for transitions.



(a) Component T.



(b) Component B.



(c) $T \otimes B$.

Figure 2.14: Example component nets and their parallel composition.

2.6.5 Connectedness, Purity and Simplicity

We briefly explore several standard Petri net properties that PNBs *do not* satisfy; specifically, connectedness, purity and simplicity (described in [141]). These properties, assuming they have been stated in the naive way for PNBs, are *not* preserved by synchronous composition and thus they are in-fact ill-defined for arbitrary PNBs.

When introducing ENs, we mentioned that in the literature, an EN should have no isolated places or transitions—the net is *connected*. However, we lift this restriction for the ENs underlying PNBs: we may have places that are unconnected to transitions and transitions unconnected to any places.

To see that connectedness is not preserved by composition, consider the component nets in Fig. 2.15. Assuming the naive extension of the definition from Petri nets, both L and R are connected (R vacuously so), yet their synchronous composition is not

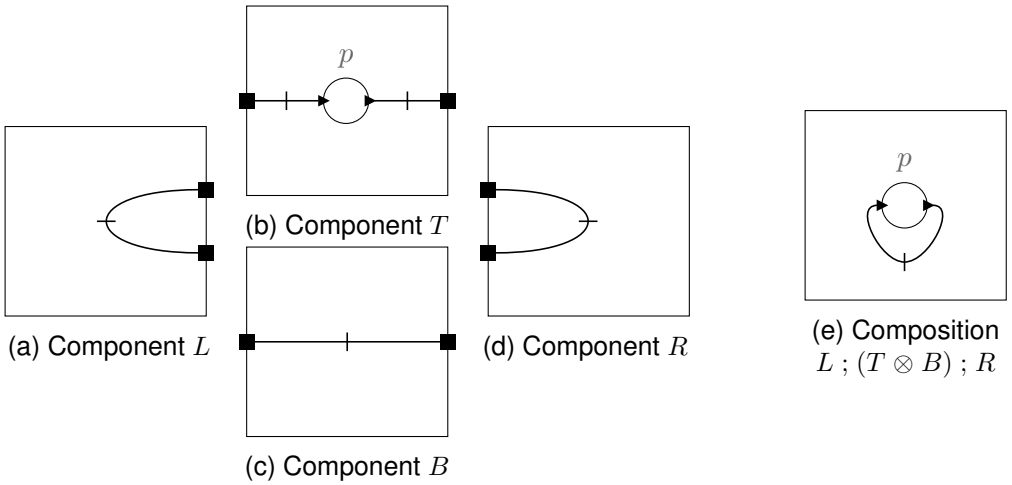
connected: there is no transition in R for t to synchronise with, therefore t is not present in any synchronisation, leaving p unconnected in $L \,;\, R$.

(a) Component L (b) Component R (c) Composition $L \,;\, R$

Figure 2.15: Connectedness is not preserved by composition

Whereas the EN definition of connectedness only concerns places and transitions, since PNB transitions can also connect to boundary ports, we might additionally require that boundary ports are connected (i.e. each boundary port has at least one transition connected to it). However, this is an unreasonable restriction: recall that each boundary port allows the *possibility* of partially specifying those transitions that connect to it; it is therefore entirely reasonable to allow no transition to be partially specified through a particular boundary port.

Purity is the property of Petri nets that there are no self loops: a single transition cannot be connected to both the out-port and the in-port of a single place. Consider the components shown in Fig. 2.16; each is pure (again by the naive extension of the definition to PNBs), yet their composition is impure, since there is a single transition connecting the out-port of p to its in-port.



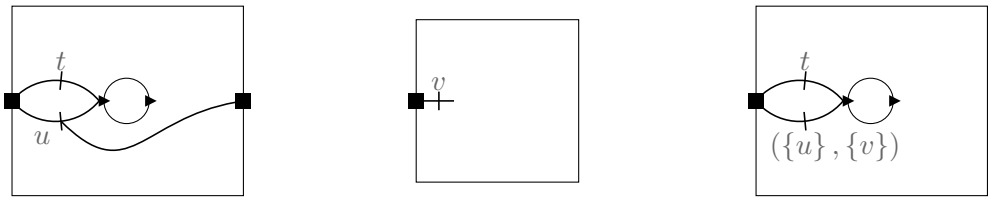
(a) Component L (b) Component T (d) Component R (e) Composition $L \,;\, (T \otimes B) \,;\, R$

(c) Component B

Figure 2.16: Purity is not preserved by composition

Finally, simplicity is the property that the set of places that a transition connects to uniquely identifies it. Again, assuming the naive extension to PNBs (i.e. also considering the boundary ports in the unique *footprint*), consider the components shown in Fig. 2.17. Each component is simple, yet their composition is not: both transitions

connect the out-port of p_1 to the in-port of p_2, yet they are different transitions, one is $(\{t\}, \varnothing)$ and the other is $(\{u\}, \{v\})$.



(a) Component L (b) Component R (c) Composition $L ; R$

Figure 2.17: Simplicity is not preserved by composition

Thus, despite the fact that any PNB, $N : (0, 0)$, is *isomorphic* to a Petri net, there can be no guarantee that the Petri net will necessarily be pure, connected or simple.

We now proceed to defining an LTS-based semantics for PNBs, in a similar fashion to that of Petri nets.

2.6.6 2-LTS Semantics of PNBs

Recall that the LTS semantics for Petri nets defined in §2.4.1 labels the LTS transition between two states (markings) being the set of Petri net transitions that were fired to transform between the markings.

For PNBs, we could use the same definition. However, as we will show later in this thesis—with a category theoretic proof §3.5.3 and an LTS-based proof §5.3—it is sufficient to forget the identities of the fired transitions, and instead only record their *effect* of their interactions on the boundaries. Furthermore, since PNBs have two boundaries, we use a 2-LTS semantics, with one label per boundary side.

As a first step towards defining the 2-LTS semantics of PNBs, we show how to map sets of boundary ports into binary strings: indeed, we use pairs of binary strings to record the interactions of fired PNB transitions on the boundaries:

$$\lceil - \rceil : 2^{\underline{k}} \to \mathbb{B}^k$$

$$\lceil O \rceil_l \stackrel{\text{def}}{=} \begin{cases} 1 \text{ if } l \in O \\ 0 \text{ otherwise,} \end{cases} \qquad \text{for } (0 \leq l < k)$$

that is, the l^{th} character of the string is 1 if the l^{th} boundary port is in the set of boundary ports, and 0 otherwise. Now, we can define the 2-LTS semantics of PNBs:

Definition 2.41 (2-LTS semantics of a PNB)**.**
For a PNB $N : (k, l)$, its 2-LTS semantics is written as $[\![N]\!]$ and is a (k, l)-LTS:

$$(2^{\text{places}(N)}, \mathbb{B}^k \times \mathbb{B}^l, \rightarrow)$$

with $m \xrightarrow{\alpha/\beta} n$ iff there is a set of PNB transitions, U, such that

$$N_m \rightarrow_U N_n \text{ with } \lceil {}^\bullet U \rceil = \alpha \text{ and } \lceil U^\bullet \rceil = \beta$$

Example 2.12.
The 2-LTS semantics for the simple PNB in Fig. 2.14b is given in Fig. 2.18.

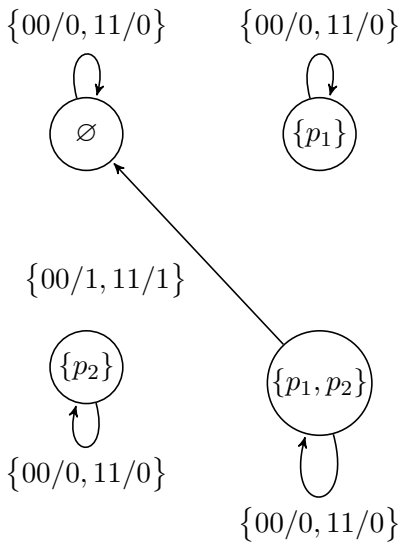


Figure 2.18: $(2, 1)$-LTS semantics of the PNB in Fig. 2.14b.

Note that the 2-LTS semantics of any net $N : (k, l)$ is always *reflexive*, that is, there exists a transition: $s \xrightarrow{0^k/0^l} s$ for every state, s, since the empty set of transitions is enabled, vacuously and is contention-free at every marking.

2.6.7 Marked PNBs

For a given PNB and pair of initial and final markings, we can consider its 2-LTS semantics as an (specially-labelled) NFA, as we did for Petri nets (Defn. 2.28), by taking the states corresponding to the appropriate markings as initial/accepting states of the NFA. To do so, we first make precise the definition of a PNB with initial/target markings:

Definition 2.42 (Marked PNB)**.**
A marked PNB is a triple, (N, m, n) consisting of a PNB, N, together with a particular initial (m) and target (n) marking.

We can now define the 2-NFA semantics of a *marked* PNB as follows:

Definition 2.43 (2-NFA semantics of a marked PNB)**.**
Given a marked PNB, (N, m, n), the 2-NFA semantics extends the 2-LTS semantics: the initial state is the state corresponding to m and the accepting states are the singleton set containing the state corresponding to n. We write $\langle\!\langle (N, m, n) \rangle\!\rangle$ for the 2-NFA semantics of a marked PNB, N.

Example 2.13.
Consider the marked PNB $(B, \{p_1, p_2\}, \varnothing)$, where B is taken from Fig. 2.14b. We construct $\langle\!\langle B \rangle\!\rangle$ by taking the 2-LTS shown in Fig. 2.18, and making the state corresponding to the empty marking as accepting, and that corresponding to the state marking both places as initial.

2.6.8 Isomorphism of PNBs

Suppose that $N, M : (k, l)$ are PNBs. We say that a mapping $f : N \to M$, comprised of two components: $f_P : \mathrm{places}(N) \to \mathrm{places}(M)$ and $f_T : \mathrm{trans}(N) \to \mathrm{trans}(M)$, is a homomorphism if the following hold, for all $t, u \in \mathrm{trans}(N)$:

1. ${}^\circ f_T(t) = \{ f_P(p) \mid p \in {}^\circ t \}$,
2. $f_T(t)^\circ = \{ f_P(p) \mid p \in t^\circ \}$,
3. ${}^\bullet f_T(t) = {}^\bullet t$,
4. $f_T(t)^\bullet = t^\bullet$,
5. $f_T(t) \bowtie_N f_T(u) \iff t \bowtie_M u$.

We say that a homomorphism f is an isomorphism iff both components are bijections, writing $N \cong M$ when an isomorphism exists between N and M.

Isomorphic nets clearly have isomorphic markings (and thus 2-LTS semantics):

Lemma 2.44.
If N and M are nets, and $N \cong M$, then $[\![N]\!] \cong [\![M]\!]$.

Proof. Immediate: the state component of the LTS-isomorphism is the place component of the PNB-isomorphism; the PNB transition structure is preserved, the label component of the LTS-isomorphism is the identity function. □

2.6.9 Read Arcs

In this thesis we introduce a slight modification of PNBs, adding a new arc type (connection between transitions and places), called a read arc, as found in contextual nets [142].

Read arcs allow for (concurrent) *non-destructive* reads of a token at a particular place. In standard PNBs, it is possible for a transition to check for the presence of a token at a particular place: the transition can remove the token from the place under consideration, placing it at a temporary place, before another transition fires, placing the token back in the original place. However, only one such remove/replace loop can be concurrently fired for any single target place (each removal transition is in contention with every other). We say that remove/replace loops are *destructive reads* of a token. An example of a net using remove/replace loops is illustrated in Fig. 2.19; its intuitive behaviour is that p_1 is filled by u, emptied by v and the presence of its token can be checked by t and w. Indeed, t and w both attempt to remove p_1's token, which cannot happen concurrently, hence the presence of a token at p_1 cannot be signalled on the left and right at once. Indeed, using remove/replace loops (and the extra required places) in this manner leads to (undesired) additional behaviour, as can be witnessed in the 2-LTS semantics shown in Fig. 2.20.

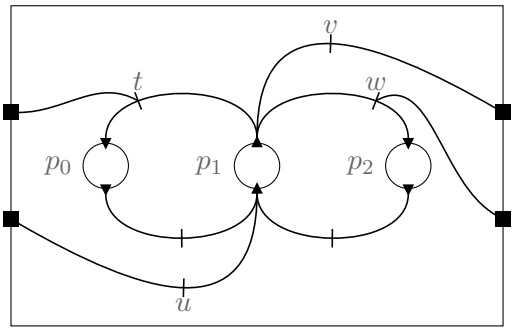


Figure 2.19: PNB with remove/replace loops

Adding read arcs to the definition of PNBs requires making simple modifications to the definitions of contending/enabled transitions:

Definition 2.45 (PNB with read arcs)**.**
A PNB with read arcs is a 10-tuple: $(P, T, {}^\circ-, -^\circ, l, r, {}^\bullet-, -^\bullet, {}^\star-, \bowtie)$*, where:*

- $(P, T, {}^\circ-, -^\circ, l, r, {}^\bullet-, -^\bullet, \bowtie)$*, is a PNB,*
- ${}^\star- : T \to 2^P$ *connects a transition to places via read arcs,*
- $\bowtie$ *is a read-arc aware contention relation: a symmetric relation such that for* $(t, u) \in T \times T$*,* $t \neq u$*, for which any of the conditions of Defn. 2.33 or the following condition holds:*

$$ {}^\star t \cap {}^\circ u^\circ \neq \varnothing $$

then $t \bowtie u$*.*

Remark 2.46. Observe that two transitions, t, u may have ${}^\star t \cap {}^\star u \neq \varnothing$, but $t \not\bowtie u$. Indeed, this is what allows concurrent non-destructive reads.

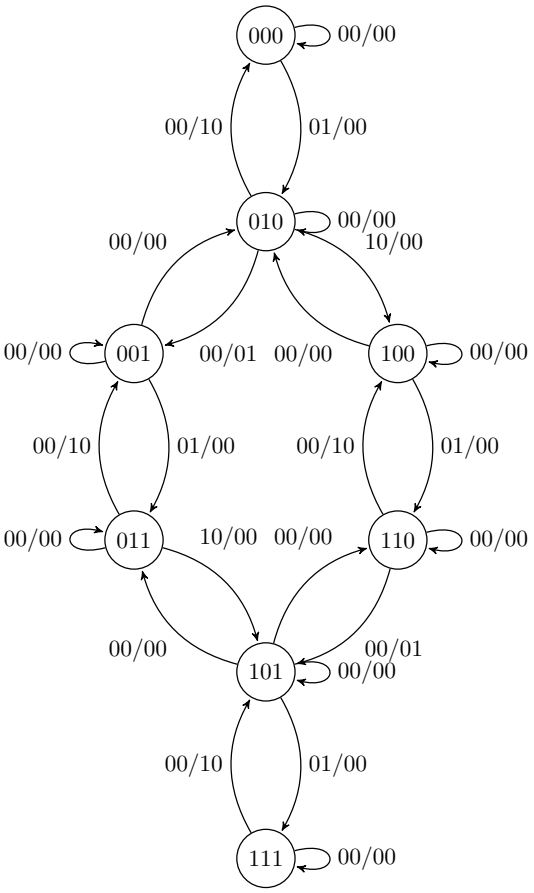


Figure 2.20: 2-LTS semantics of the PNB with remove/replace loops shown in Fig. 2.19. For compact representation, markings are identified with binary strings: there is a 1 at position i iff place p_i is marked.

The graphical representation of a read arc is an undirected edge connecting to the side of a place. The example shown in Fig. 2.19 can be encoded using read arcs as shown in Fig. 2.21. Now, t and u are not in contention, and are both enabled by the current marking. Thus, the set $\{t, u\}$ *can* be fired (preserving the net's marking).

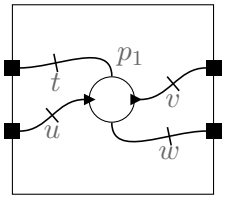


Figure 2.21: PNB with read arcs

We tweak the definition of enabled transitions from that of Defn. 2.20, to require that each place being read has a token, and that a transition does not produce/consume a token at the same place as it reads:

Definition 2.47 (Enabled Transitions for PNBs with Read Arcs)**.**
For a PNB with read arcs, N, a transition $t \in \mathrm{trans}(N)$ is enabled for a marking, m, written enabled$_m(t)$, if:

1. $\forall p \in {}^\circ t, \mathrm{m}(p) = 1$
2. $\forall q \in {}^\star t, \mathrm{m}(q) = 1$
3. $\forall r \in t^\circ, \mathrm{m}(r) = 0$
4. ${}^\circ t^\circ \cap {}^\star t = \varnothing$

Remark 2.48. To see why these modifications are necessary, observe that PNBs *without* read arcs may contain *self-conflicting* transitions: a transition can share a place between its pre and post sets. However, sets containing a self-conflicting transition *are* contention-free by Defn. 2.19, but, they cannot be fired, since they are not enabled. Indeed, to be enabled, a transition's pre-set must be marked and the post set unmarked; a self-conflicting PNB transition would require a place to be both marked and unmarked and therefore is never enabled.

On the other hand, PNBs with read arcs may contain a transition, t, which attempts to consume the token from a place and also read the token, i.e. there is some $p \in {}^\circ t$ such that $p \in {}^\star t$. Such a transition is similarly *never* enabled, by Defn. 2.47. Intuitively, it does not make sense to non-destructively read a token *and* destructively remove a token from the same place.

We are now able to construct LTS semantics of PNBs with read arcs, by amending the firing semantics of Defn. 2.34 to use the modified definition of enabled transition sets (Defn. 2.47). The 2-LTS semantics of the PNB with read arcs shown in Fig. 2.21 is shown in Fig. 2.22; it is clear that the undesirable *additional* behaviour due to using remove/replace loops is no longer present.

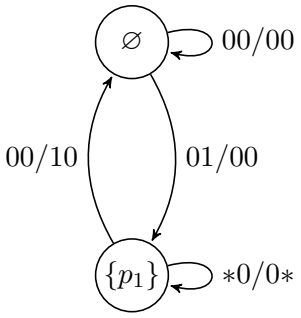


Figure 2.22: 2-LTS semantics of the PNB with read arcs shown in Fig. 2.21.

Chapter 3

Categorical Structure

In this chapter we show that both PNBs and 2-LTSs have a natural categorical structure. We consider the categories of PNBs, **PNB**, and 2-LTSs, **2-LTS**. Our aim is to show that these categories are in fact Product and Permutation Categories, known as PROPs, and that there is a semantic functor $\langle\!\langle - \rangle\!\rangle : \mathbf{PNB} \to \mathbf{2\text{-}LTS}$ taking a PNB to its 2-LTS statespace, preserving the PROP structure. By elucidating the PROP structure we show that the two types of composition for PNBs/2-LTSs enjoy desirable properties (associativity, identities), while functoriality ensures that the compositions are preserved when taking the semantics of components, in a precise sense. Indeed, we demonstrate that the property of *compositionality*—that the semantics of a composite PNB is determined by the semantics of the components—is simply an instance of functoriality.

3.1 Preliminaries

We briefly recall the requisite definitions of strict symmetric monoidal categories and functors. For an in-depth introduction to symmetric monoidal categories, and their intuitive graphical presentation, see [143].

Definition 3.1 (Category)**.**
A (locally small) category, $\mathcal{C}$, is comprised of:

- *A collection of objects,* $\mathrm{obj}_{\mathcal{C}}$*,*
- *A collection of morphisms; for each pair of objects, X, Y we have a set of morphisms between those objects, called their hom-set, written:* $\mathrm{Hom}_{\mathcal{C}}(X, Y)$*,*
- *For every object, X, we have the identity morphism:* $\mathrm{id}_X \in \mathrm{Hom}_{\mathcal{C}}(X, X)$*,*
- *For each $f \in \mathrm{Hom}_{\mathcal{C}}(X, Y)$ and $g \in \mathrm{Hom}_{\mathcal{C}}(Y, Z)$, there is a composite morphism:* $f \,;\, g \in \mathrm{Hom}_{\mathcal{C}}(X, Z)$*.*

Subject to the following:

$$f \,;\, \mathrm{id}_Y = f = \mathrm{id}_X \,;\, f$$
$$f \,;\, (g \,;\, h) = (f \,;\, g) \,;\, h$$

for any $X, Y, Z, W \in \mathrm{obj}_{\mathcal{C}}$*, and* $f \in \mathrm{Hom}_{\mathcal{C}}(X, Y)$*,* $g \in \mathrm{Hom}_{\mathcal{C}}(Y, Z)$ *and* $h \in \mathrm{Hom}_{\mathcal{C}}(Z, W)$*.*

Definition 3.2 (Isomorphisms)**.**
A morphism $f \in \mathrm{Hom}_{\mathcal{C}}(X, Y)$ *is an isomorphism if there exists* $f^{-1} \in \mathrm{Hom}_{\mathcal{C}}(Y, X)$ *subject to the conditions:*

$$f \,;\, f^{-1} = \mathrm{id}_X$$
$$f^{-1} \,;\, f = \mathrm{id}_Y$$

Definition 3.3 (Product Category)**.**
For categories $\mathcal{C}$ *and* $\mathcal{D}$*, we can lift their operations pointwise to form the product category,* $\mathcal{C} \times \mathcal{D}$*, which has the following structure:*

$$\mathrm{obj}_{\mathcal{C}\times\mathcal{D}} = \mathrm{obj}_{\mathcal{C}} \times \mathrm{obj}_{\mathcal{D}}$$
$$\mathrm{Hom}_{\mathcal{C}\times\mathcal{D}}((X,\, Y)\,, (Z,\, W)) = \mathrm{Hom}_{\mathcal{C}}(X, Z) \times \mathrm{Hom}_{\mathcal{D}}(Y, W)$$
$$\mathrm{id}_{(X,\, Y)} = (\mathrm{id}_X,\, \mathrm{id}_Y)$$
$$(f,\, g) \,;\, (h,\, i) = (f \,;\, h,\, g \,;\, i)$$

Definition 3.4 (Functor)**.**
A functor is a mapping between categories, $\mathcal{F} : \mathcal{C} \to \mathcal{D}$*, subject to the conditions:*

$$\mathcal{F}\,(\mathrm{id}_X) = \mathrm{id}_{\mathcal{F}(X)}$$
$$\mathcal{F}(f \,;\, g) = \mathcal{F}\,(f) \,;\, \mathcal{F}\,(g)$$

Definition 3.5 (Bifunctor)**.**
A bifunctor is a functor whose domain category is a product category.

Definition 3.6 (Natural Transformation)**.**
A natural transformation is a mapping between functors, $\eta : \mathcal{F} \to \mathcal{G}$*, comprised of a morphism, called a component,* $\eta\,(X) : \mathcal{F}\,(X) \to \mathcal{G}\,(X)$*, for each* $X \in \mathrm{obj}_{\mathcal{C}}$*, such that for any* $f \in \mathrm{Hom}_{\mathcal{C}}(X, Y)$*, the diagram in Fig. 3.1 commutes.*

Definition 3.7 (Natural Isomorphism)**.**
A natural transformation, η*, is a natural isomorphism if every component is an isomorphism.*

Definition 3.8 (Monoidal Category)**.**
A monoidal category is a 6-tuple, $(\mathcal{C}, \otimes, I, \alpha, \lambda, \rho)$*, where:*

- $\mathcal{C}$ *is a category,*

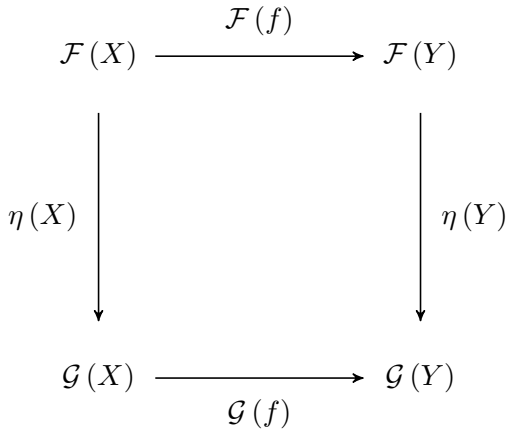


Figure 3.1: Natural transformation commuting diagram

- $\otimes$: $\mathcal{C} \times \mathcal{C} \to \mathcal{C}$ *is a bifunctor, called tensor,*
- $I \in \mathrm{obj}_{\mathcal{C}}$ *is the tensor unit,*
- $\alpha_{X,Y,Z} : (X \otimes Y) \otimes Z \to X \otimes (Y \otimes Z)$ *is the associativity natural isomorphism,*
- $\lambda_X : I \otimes X \to X$ *(*$\rho_X : X \otimes I \to X$*) are the left (right) identity natural isomorphisms.*

Subject to the pentagon and triangle conditions, illustrated in Fig. 3.2 and Fig. 3.3.

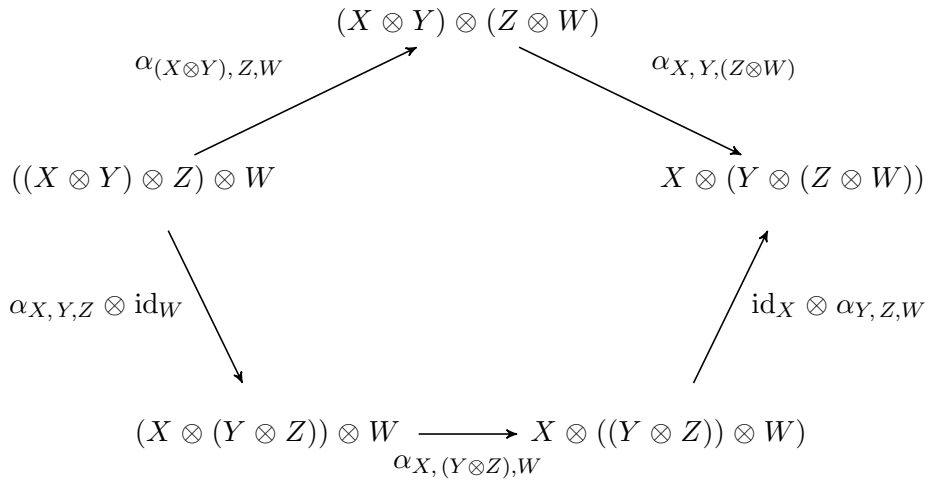


Figure 3.2: Pentagon coherence condition

Definition 3.9 (Strict Monoidal Category)**.**
A strict monoidal category, $(\mathcal{C}, \otimes, I, \alpha, \lambda, \rho)$*, is a monoidal category where* α, λ *and* ρ *are all identity mappings.*

Definition 3.10 (Symmetric Monoidal Category)**.**
A symmetric monoidal category is a monoidal category with a braiding, γ*, which is a*

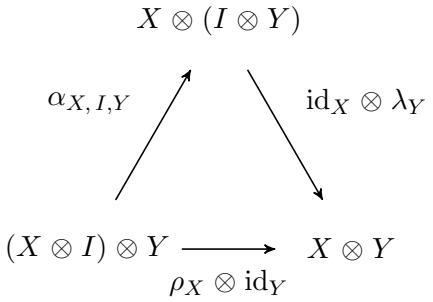


Figure 3.3: Triangle coherence condition

natural isomorphism witnessing the commutativity of the tensor product:

$$\gamma_{(X,Y)} : X \otimes Y \to Y \otimes X$$

such that the symmetry is self-inverse (i.e. Fig. 3.4 commutes) and the hexagon diagram commutes (Fig. 3.5).

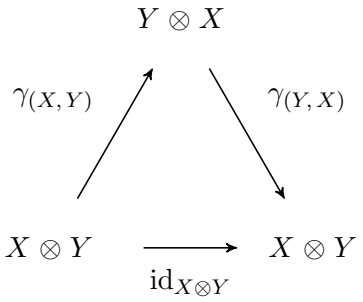


Figure 3.4: Symmetry is self-inverse

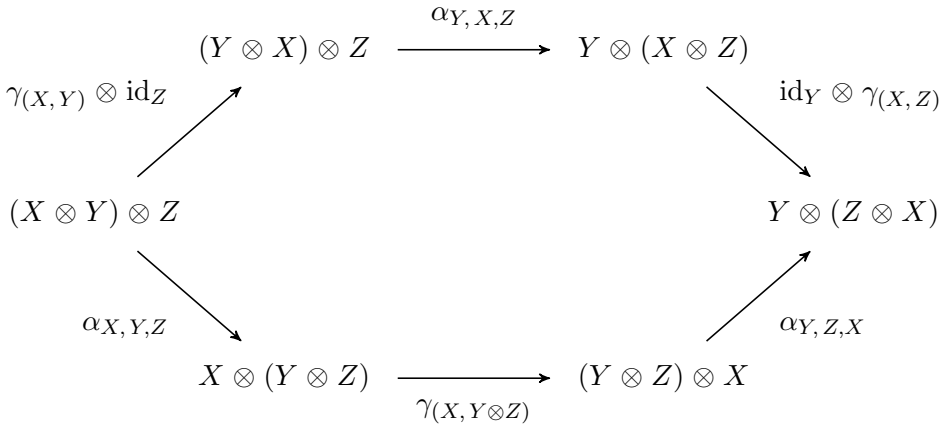


Figure 3.5: Hexagon coherence condition

Symmetric monoidal categories with objects the natural numbers and tensor being addition are known as PROPs [144]:

Definition 3.11 (PROP)**.**
A PROP (Products and Permutations) is a strict monoidal category that has $\mathbb{N}$ *as its objects, and* $\otimes$ *being addition on objects.*

Monoidal functors map between monoidal categories, whilst preserving the monoidal structure:

Definition 3.12 (Monoidal Functor)**.**
A monoidal functor is a functor between monoidal categories, $\mathcal{C}$ *and* $\mathcal{D}$*, formed of three components,* $\left(\mathcal{F}, \theta^1, \theta^2\right)$ *where:*

- $\mathcal{F} : \mathcal{C} \to \mathcal{D}$ *is a functor,*
- $\theta^1 : I \to \mathcal{F}(I)$ *is a morphism in* $\mathcal{D}$*,*
- $\theta^2_{(X,Y)} : \mathcal{F}(X) \otimes \mathcal{F}(Y) \to \mathcal{F}(X \otimes Y)$ *is a natural transformation.*

subject to the conditions illustrated in Fig. 3.6 and Fig. 3.7.

Definition 3.13 (Strict Monoidal Functor)**.**
A monoidal functor, $\left(\mathcal{F}, \theta^1, \theta^2\right)$*, is strict iff* θ^1 *and* θ^2 *are identity mappings.*

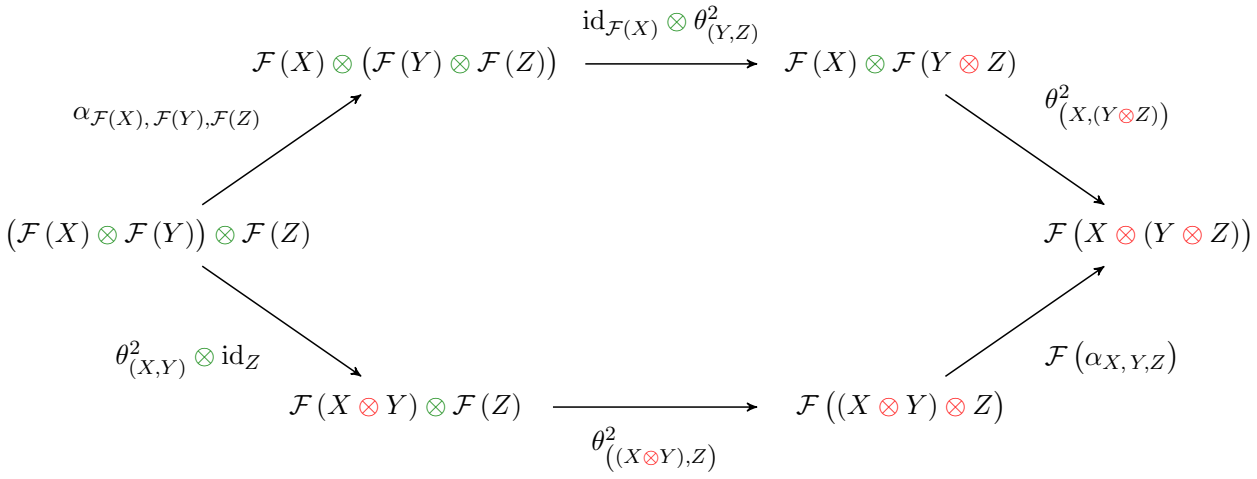


Figure 3.6: Monoidal functor coherence 1

A symmetric monoidal functor is then a monoidal functor that preserves the braided structure:

Definition 3.14 (Symmetric Monoidal Functor)**.**
A monoidal functor, $\mathcal{F}$ *between symmetric monoidal categories* $\mathcal{C}$ *and* $\mathcal{D}$ *is symmetric if the diagram in Fig. 3.8 commutes.*

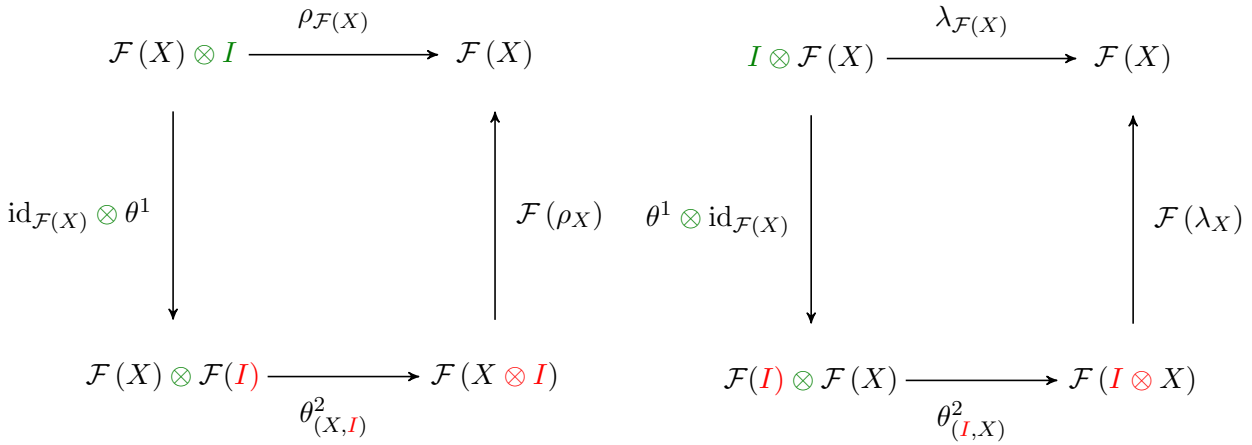


Figure 3.7: Monoidal functor coherence 2

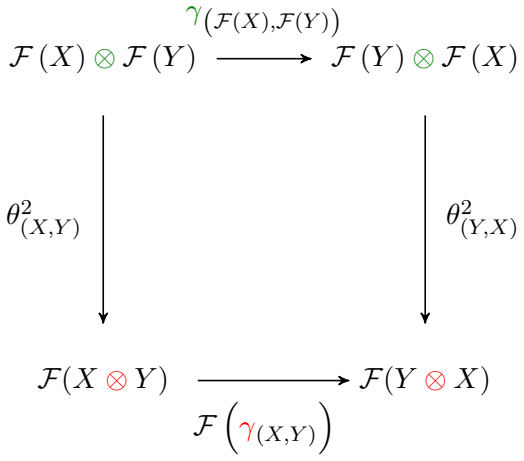


Figure 3.8: Symmetric monoidal functor coherence

3.2 The category of PNBs

As shown by Bruni et al. [61, Proposition 5.1], PNBs form a category. Before we give the structure of this category, we must describe isomorphism classes of PNBs:

Definition 3.15 (PNB Isomorphism Class)**.**
For a PNB, $N : (k, l)$*, its PNB isomorphism class, written* $[N] : (k, l)$ *is the set* $\{M \mid N \cong M\}$.

Now, **PNB**, the category of PNBs has the following structure:

- Objects are the natural numbers, $\mathbb{N}$,
- Arrows from k to l are the PNB isomorphism classes: $[N] : (k, l)$,

- The identity morphism for $k \in \mathbb{N}$, is the net $P_{id_k} : (k,\, k)$, illustrated in Fig. 3.9. P_{id_k} has no places, k boundaries on the left and right, and transitions connecting the i^{th} left boundary to the i^{th} right boundary,
- The composition of morphisms $N : (k,\, l)$ and $M : (l,\, m)$, is $N \;;\; M : (k,\, m)$, obtained using PNB composition, defined in Defn. 2.38.

To show that such a structure is well-defined, Bruni et al. [61, Proposition 5.1] proved:

1. PNB composition is compatible with isomorphism classes: $N \;;\; M' \cong N' \;;\; M$ with $N \cong N'$ and $M \cong M'$,
2. PNB composition is associative up-to isomorphism: $N \;;\; (M \;;\; O) \cong (N \;;\; M) \;;\; O$,
3. PNB composition has a unit up-to isomorphism: $N \;;\; P_{id} \cong N \cong P_{id} \;;\; N$.

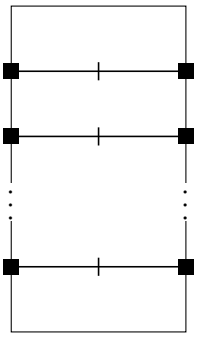


Figure 3.9: Net $P_{id_k} : (k,\, k)$

Furthermore, there is a (strict) monoidal structure on **PNB**:

- The tensor product is addition on objects and $\otimes$-composition (Defn. 2.40) on morphisms,
- The unit is 0,
- α, λ and ρ are identities.

Bruni et al. [61][Proposition 5.2] assures us that tensor is well-defined for equivalence classes of PNBs, and is a bifunctor. Indeed, addition is strictly associative with 0 being its strict identity, thus it follows that **PNB** is strict monoidal:

Proposition 3.16.
PNB *is a strict monoidal category.*

We now show that there is a *symmetric* monoidal structure on **PNB**. For any $k, l \in \mathbb{N}$, there is a PNB, $P_{\mathsf{sw}(k,\,l)} : (k+l,\, l+k)$, illustrated in Fig. 3.10. $P_{\mathsf{sw}(k,\,l)}$ has no places, and

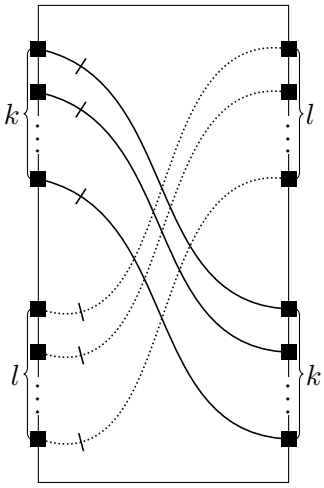


Figure 3.10: Net $P_{\mathsf{sw}(k,\,l)} : (k + l,\, l + k)$

$k + l$ of each of transitions, left boundaries and right boundaries. The $k + l$ transitions are connected to boundary ports as follows: for $0 \leq x < k$, ${}^\bullet t_x = x$ and $t_x^\bullet = l + x$ and for $0 \leq y < l$, ${}^\bullet t_{k+y} = k + y$ and $t_{k+y}^\bullet = y$.

$P_{\mathsf{sw}(k,\,l)}$ is an example of a PNB whose transitions are 1-1 with its boundary ports; we refer to such PNBs as *permutation* PNBs:

Definition 3.17 (Permutation PNB)**.**
A PNB, $P_\sigma : (k,\, k)$, is a Permutation PNB iff σ is a permutation on $\underline{k}$, $\mathrm{places}(P_\sigma) = \varnothing$, and the following hold, with t, u ranging over $\mathrm{trans}(P_\sigma)$:

- $|{}^\bullet t| = 1$,
- $|t^\bullet| = 1$,
- *if $t \neq u$ then* ${}^\bullet t \cap {}^\bullet u = \varnothing$,
- *if $t \neq u$ then* $t^\bullet \cap u^\bullet = \varnothing$,
- $\forall b \in \underline{k}$, $\exists v \in \mathrm{trans}(N)$ ***s.t.*** ${}^\bullet v = \{b\}$.
- $\forall b \in \underline{l}$, $\exists v \in \mathrm{trans}(N)$ ***s.t.*** $v^\bullet = \{b\}$.

That is, each transition is connected to exactly one left (right) boundary port, distinct transitions do not share left (right) boundary ports and for every left (right) boundary port, there is a transition that connects to that boundary port. The boundary connections of the transitions are determined by σ: given $t \in \mathrm{trans}(P_\sigma)$, $t^\bullet = \big\{\sigma(b) \;\big|\; {}^\bullet t = \{b\}\big\}$.

Permutation PNBs embed permutations on $\underline{k}$; we refer to such permutations as k-permutations. Indeed, we can compose permutations, both sequentially and in parallel:

Definition 3.18 (Synchronous composition of permutations)**.**
Given two k-permutations, σ_1 and σ_2, their synchronous composition is a k-permutation, written $\sigma_1 \,;\, \sigma_2$, and simply composes the underlying bijections: $(\sigma_1 \,;\, \sigma_2)(x) \stackrel{\mathrm{def}}{=} \sigma_2(\sigma_1(x))$.

Definition 3.19 (Parallel composition of permutations)**.**
Given a k-permutation, σ_1, and l-permutation, σ_2, their tensor composition is a $(k+l)$-permutation, written $\sigma_1 \otimes \sigma_2$, and defined:

$$(\sigma_1 \otimes \sigma_2)(x) \stackrel{\text{def}}{=} \begin{cases} \sigma_1(x) \text{ if } 0 \leq x < k \\ \sigma_2(x-k)+k \text{ if } k \leq x < k+l \end{cases}$$

that is, we directly apply σ_1 if x is in its domain, else we apply a shift to x, such that it is in the domain of σ_2, before applying σ_2 and the inverse shift.

We later make use of two particular permutations, $\mathsf{id}(k)$ and $\mathsf{sw}(l, m)$ (identity and "swap", or *braid* permutations, respectively):

Example 3.1.
$\mathsf{id}(k)$ is a k-permutation, with: $\mathsf{id}(k)(x) \stackrel{\text{def}}{=} x$.

Example 3.2.
$\mathsf{sw}(l, m)$ is a $l+m$-permutation, with:

$$\mathsf{sw}(l, m)(x) \stackrel{\text{def}}{=} \begin{cases} x+m \text{ if } 0 \leq x < l \\ x-l \text{ otherwise.} \end{cases}$$

A simple lemma confirms that permutation PNBs are closed (up-to isomorphism) under synchronous and tensor composition.

Lemma 3.20.
If $P_{\sigma_1} : (k, k)$, $P_{\sigma_2} : (k, k)$ and $P_{\sigma_3} : (l, l)$ are permutation PNBs, then:

1. *We have $P_{\sigma_1} \,;\, P_{\sigma_2} : (k, k)$, with $P_{\sigma_1} \,;\, P_{\sigma_2} \cong P_{\sigma_1;\sigma_2}$,*
2. *We have $P_{\sigma_1} \otimes P_{\sigma_3} : (k+l, k+l)$, with $P_{\sigma_1} \otimes P_{\sigma_3} \cong P_{\sigma_1 \otimes \sigma_3}$.*

Proof. For part 1, we have that each $b \in \underline{l}$ determines unique $t \in \mathrm{trans}(N)$ and $u \in \mathrm{trans}(M)$. Thus $(\{t\}, \{u\})$ is a minimal synchronisation, with ${}^\bullet(\{t\}, \{u\})$ and $(\{t\}, \{u\})^\bullet$ being unique, in particular, since $\{t\}^\bullet = {}^\bullet\{u\}$, we have: $\{u\}^\bullet = \{\sigma_2(\sigma_1(b)) \mid \{b\} = {}^\bullet\{t\}\}$. For part 2, observe that no boundary ports or transitions are created, and existing transitions' connections are preserved with those in P_{σ_3} being appropriately shifted. □

We prove 3 technical lemmas relating to PNB compositions with a permutation PNB, $N \,;\, P_\sigma$: the first says that each transition of N synchronises with a uniquely defined set of transitions in P_σ. The second says that the transitions of $N \,;\, P_\sigma$ (that is, minimal synchronisations) consist of single transitions from N. Finally, the third says that transitions of $N \,;\, P_\sigma$ are in bijection with those of N.

Lemma 3.21.
For $P_\sigma : (k, k)$, each $T \in 2^{\underline{k}}$ (i.e. T is a set of left boundary ports of P_σ) determines a unique $V \subseteq \mathrm{trans}(P_\sigma)$ such that $T = {}^\bullet V$.

Proof. By induction on T: in the base case of $T = \varnothing$, $\varnothing$ suffices. In the case of $T = T' \cup \{b\}$, we apply the I.H. to T', obtaining the unique $V' \subseteq \mathrm{trans}(P_\sigma)$ such that $T' = {}^\bullet V'$. Then, by our assumptions on $\mathrm{trans}(P_\sigma)$, there is a unique $u \in \mathrm{trans}(P_\sigma)$ such that ${}^\bullet u = b$; therefore, $V' \cup \{u\}$ satisfies the requirements. □

Lemma 3.22.
For $N : (k, l)$ and $P_\sigma : (l, l)$, each $(U, V) \in \mathrm{trans}(N \,;\, P_\sigma)$ has $|U| = 1$.

Proof. For a contradiction, assume that we have a $(U, V) \in \mathrm{trans}(N \,;\, P_\sigma)$ with $|U| > 1$. Take any $t \in U$, and using Lem. 3.21 identify the (possibly empty) unique set $W \subseteq \mathrm{trans}(P_\sigma)$ such that $t^\bullet = {}^\bullet W$; since W is unique in $\mathrm{trans}(P_\sigma)$, we must have that $W \subseteq V$.

Then, since U and V are contention-free, we have that $(U \setminus \{t\})^\bullet = U^\bullet \setminus t^\bullet = {}^\bullet V \setminus {}^\bullet W = {}^\bullet(V \setminus W)$, and clearly, $(U \setminus \{t\}, V \setminus W) \subseteq (U, V)$. In other words, (U, V) is not a *minimal* synchronisation, contradicting $(U, V) \in \mathrm{trans}(N \,;\, P_\sigma)$. □

Lemma 3.23.
For $N : (k, l)$ and $P_\sigma : (l, l)$, there is a bijection between $\mathrm{trans}(N)$ and $\mathrm{trans}(N \,;\, P_\sigma)$. In particular, each $t \in \mathrm{trans}(N)$ determines a unique $(U, V) \in \mathrm{trans}(N \,;\, P_\sigma)$ such that $U = \{t\}$.

Proof. We have that $\mathrm{trans}(N \,;\, P_\sigma)$ is the set of all minimal synchronisations between N and P_σ. By Lem. 3.22 every $(U, V) \in \mathrm{trans}(N \,;\, P_\sigma)$ has $|U| = 1$, i.e. $U = \{t\}$ for some t, such that, by Lem. 3.21, V is uniquely determined by $t^\bullet$. In the other direction, suppose that there is no $(U, V) \in \mathrm{trans}(N \,;\, P_\sigma)$ such that $U = \{t\}$; by Lem. 3.21 we have the unique set of transitions $U \subseteq \mathrm{trans}(P_\sigma)$ such that $t^\bullet = {}^\bullet U$. Clearly $(\{t\}, U)$ is a minimal synchronisation and thus must appear in $\mathrm{trans}(N \,;\, P_\sigma)$, a contradiction. □

Indeed, we can easily obtain the symmetric versions of the previous lemmas, for composition with a permutation PNB on the left; the proofs of which follow the same arguments:

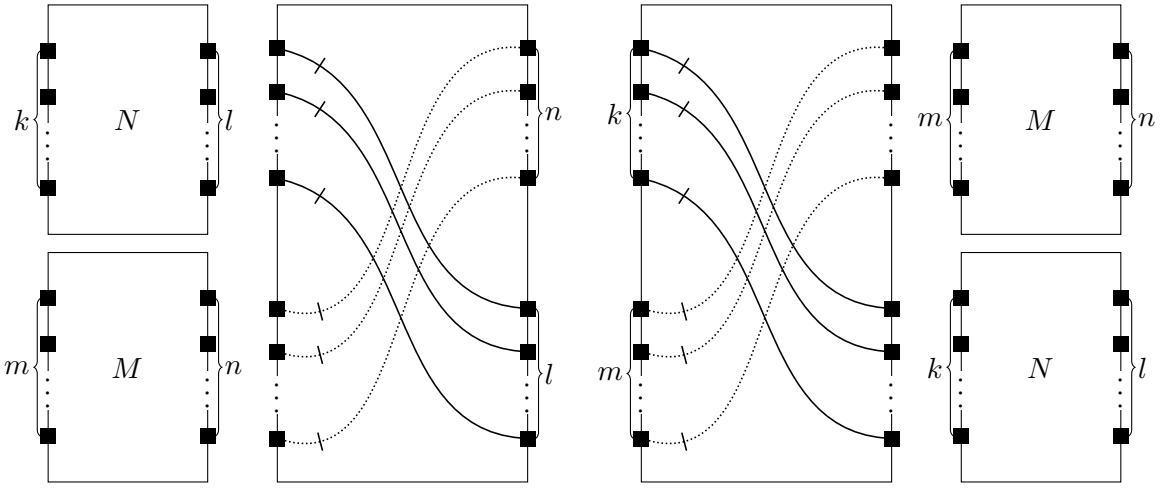
Lemma 3.24.
For $P_\sigma : (k, k)$, each $T \in 2^{\underline{k}}$ determines a unique $U \subseteq \mathrm{trans}(P_\sigma)$ such that $T = U^\bullet$.

For $P_\sigma : (k, k)$ and $M : (k, l)$, each $t \in \mathrm{trans}(M)$, determines a unique $U \subseteq \mathrm{trans}(P_\sigma)$ such that $U^\bullet = {}^\bullet t$. □

Lemma 3.25.
For $P_\sigma : (k, k)$ and $M : (k, l)$, each $(U, V) \in \mathrm{trans}(P_\sigma \,;\, M)$ has $|V| = 1$. □

Lemma 3.26.
For $P_\sigma : (k,\, k)$ and $M : (k,\, l)$, there is a bijection between $\mathrm{trans}(M)$ and $\mathrm{trans}(P_\sigma \,;\, M)$. In particular, each $t \in \mathrm{trans}(M)$ determines a unique $(U, V) \in \mathrm{trans}(P_\sigma \,;\, M)$ such that $V = \{t\}$.

We may now prove a proposition relating tensor and the "swap" PNB; the intuition is provided graphically in Fig. 3.11, where both compositions should be isomorphic.



(a) Composition $(N \otimes M) \,;\, P_{\mathsf{sw}(l,\, n)}$ (b) Composition $P_{\mathsf{sw}(k,\, m)} \,;\, (M \otimes N)$

Figure 3.11: Prop. 3.27, graphically; (a) is isomorphic to (b).

Proposition 3.27.
For $N : (k,\, l)$ and $M : (m,\, n)$, the following holds:

$$(N \otimes M) \,;\, P_{\mathsf{sw}(l,\, n)} \cong P_{\mathsf{sw}(k,\, m)} \,;\, (M \otimes N)$$

Proof. For any $k, l \in \mathbb{N}$, $P_{\mathsf{sw}(k,\, l)}$ has no places, thus we can give an isomorphism on places that simply maps between the tagged places of N and M, from the top (bottom) left component, to the bottom (top) right component:

$$\begin{aligned} \mathrm{inl}\,(\mathrm{inl}\,p) &\mapsto \mathrm{inr}\,(\mathrm{inr}\,p) \quad \text{for } p \in \mathrm{places}(N) \\ \mathrm{inl}\,(\mathrm{inr}\,q) &\mapsto \mathrm{inr}\,(\mathrm{inl}\,q) \quad \text{for } q \in \mathrm{places}(M) \end{aligned}$$

To show the transitions are also isomorphic, consider any $(U, V) \in \mathrm{trans}((N \otimes M) \,;\, P_{\mathsf{sw}(l,\, n)})$; we can apply Lem. 3.22, obtaining that $U = \{\mathrm{inl}\,t\}$ or $U = \{\mathrm{inr}\,t\}$. Indeed, in the former case $t \in \mathrm{trans}(N)$ and $t \in \mathrm{trans}(M)$ in the latter.

Assume without loss of generality that $i = 0$. Then, ${}^\bullet\mathrm{inl}\,t = {}^\bullet t$, and $\mathrm{inl}\,t^\bullet = t^\bullet$; indeed, since (U, V) is a synchronisation, we have that $\{\mathrm{inl}\,t\}^\bullet = {}^\bullet V$, thus, by the definition of the permutation underlying $P_{\mathsf{sw}(l,\, n)}$, $V^\bullet = \{b + n \mid b \in t^\bullet\}$. This gives us ${}^\bullet(U, V) = {}^\bullet t$ and $(U, V)^\bullet = \{b + n \mid b \in t^\bullet\}$.

Furthermore, by the definition of $\otimes$, and Lem. 3.26, we also have a $(U', V') \in \mathrm{trans}(P_{\mathsf{sw}(k,m)} \; ; (M \otimes N)))$, with $V' = \{\mathrm{inr}\, t\}$. Now, ${}^\bullet\mathrm{inr}\, t = \{b + m \mid b \in {}^\bullet t\}$ and $\mathrm{inr}\, t^\bullet = \{b + n \mid b \in t^\bullet\}$ Again, since (U', V') is a synchronisation, we have that $U'^\bullet = {}^\bullet\{\mathrm{inr}\, t\}$. By the definition of $P_{\mathsf{sw}(k,m)}$ we have that ${}^\bullet U' = \left\{ b - m \;\middle|\; b \in {}^\bullet\{(t, 1)\} \right\}$, which, by the definition of ${}^\bullet\mathrm{inr}\, t$, gives ${}^\bullet U' = {}^\bullet t$. Therefore, ${}^\bullet(U', V') = {}^\bullet t$ and $(U', V')^\bullet = \{b + n \mid b \in t^\bullet\}$.

Indeed, we have shown that $(U, V) \in (N \otimes M) \; ; \; P_{\mathsf{sw}(l,n)}$ determines $(U', V') \in P_{\mathsf{sw}(k,m)} \; ; (M \otimes N)$ such that ${}^\bullet(U, V) = {}^\bullet(U', V')$ and $(U, V)^\bullet = (U', V')^\bullet$, as required. The opposite direction uses a similar argument. □

The following isomorphisms assure us that we can equivalently braid k past $l + m$ in one step or two individual steps, and similarly for $k + l$ past m:

Proposition 3.28.
We have the following isomorphisms:

(a) $P_{\mathsf{sw}(k,l+m)} \cong (P_{\mathsf{sw}(k,l)} \otimes P_{id_m}) \; ; (P_{id_l} \otimes P_{\mathsf{sw}(k,m)})$

(b) $P_{\mathsf{sw}(k+l,m)} \cong (P_{id_k} \otimes P_{\mathsf{sw}(l,m)}) \; ; (P_{\mathsf{sw}(k,m)} \otimes P_{id_l})$

Proof. By definition; observe that we may consider P_{id_k} as a trivial permutation PNB, $P_{id(k)}$. Then, it only remains to verify that the corresponding compositions on permutations are equal:

(a) $\mathsf{sw}(k,\, l + m) = (\mathsf{sw}(k,\, l) \otimes \mathsf{id}(m)) \; ; (\mathsf{id}(l) \otimes \mathsf{sw}(k,\, m))$

(b) $\mathsf{sw}(k + l,\, m) = (\mathsf{id}(k) \otimes \mathsf{sw}(l,\, m)) \; ; (\mathsf{sw}(k,\, m) \otimes \mathsf{id}(l))$

which is immediate by definition. □

Proposition 3.29.
We have the following isomorphism: $P_{\mathsf{sw}(k,l)} \; ; P_{\mathsf{sw}(l,k)} \cong P_{id_{k+l}}$

Proof. Immediate, relying on the fact that composing σ_1 and σ_1^{-1} is equal to the identity permutation. □

We can now show that **PNB**is a strict *symmetric* monoidal category.

Proposition 3.30.
PNB *is a symmetric monoidal category.*

Proof. Prop. 3.27 confirms that the braiding is a natural isomorphism, Prop. 3.28 assures us that the braiding satisfies the hexagon axioms of Defn. 3.10 (simplified since our associators are identities) and Prop. 3.29 says that the braiding is symmetric. □

Finally, it follows that **PNB** is a PROP:

Proposition 3.31.
PNB *is a PROP.*

Proof. By [61, Proposition 5.1] and propositions 3.16 and 3.30. □

While it is informative to expose the PROP structure of **PNB** (illustrating that PNBs are an instance of a very general structure), it is the underlying monoidal category structure that is most "useful" later in this thesis. Indeed, associativity of both composition types ensures that we are free to compose from left-to-right, or right-to-left, or any grouping in between, as we do for example in Fig. 6.18, while identity of ;-composition allows us to pass *signals* past a (composite) component, for example in Fig. 4.9.

3.3 The category of 2-LTSs

We can show that 2-LTSs form a category, in a similar fashion to PNBs. In the following, we define isomorphism classes of 2-LTSs, and then show that composition is compatible with such classes and furthermore, is associative and has identities up-to isomorphism.

Definition 3.32 (PNB Isomorphism Class)**.**
For a 2-LTS, $X : (k, l)$*, its 2-LTS isomorphism class, written* $[X] : (k, l)$ *is the set* $\{Y \mid X \cong Y\}$.

In the following, $L_{id_k} : (k, k)$, is a 2-LTS, illustrated in Fig. 3.12, with a single state, and transitions the self-loops labelled by $\{x \mid x \in \mathbb{B}^n\}$

Proposition 3.33.
The following isomorphisms hold:

(i) Given 2-LTSs $X, X' : (k, l), Y, Y' : (l, m)$*, with* $X \cong X'$ *and* $Y \cong Y'$*, we have that* $X ; Y \cong X' ; Y'$*,*

(ii) For 2-LTSs $X : (k, l), Y : (l, m), Z : (m, n)$*, we have* $(X ; Y) ; Z \cong X ; (Y ; Z)$*,*

(iii) For any 2-LTS $X : (k, l)$*, we have* $L_{id_k} ; X \cong X \cong X ; L_{id_l}$*.*

Proof. For (i) we have that $(x ; y) \xrightarrow{\alpha/\beta} (x' ; y')$ is a transition of $X ; Y$ iff $x \xrightarrow{\alpha/\gamma} x'$ and $y \xrightarrow{\gamma/\beta} y'$ are transitions in X and Y, respectively, for some $\gamma \in \mathbb{B}^l$. By the definition of 2-LTS isomorphism, we have corresponding transitions in X' and Y', which, by the definition of 2-LTS composition gives the required transition in $X' ; Y'$. The converse argument is similar.

For (ii) we have that $((x\,;\,y)\,;\,z) \xrightarrow{\alpha/\delta} \Big((x'\,;\,y')\,;\,z'\Big)$ is a transition of $(X\,;\,Y)\,;\,Z$ iff $(x\,;\,y) \xrightarrow{\alpha/\gamma} (x'\,;\,y')$ and $z \xrightarrow{\gamma/\delta} z'$ are transitions in $X\,;\,Y$ and Z, respectively, for some $\gamma \in \mathbb{B}^m$, and $x \xrightarrow{\alpha/\beta} x'$ and $y \xrightarrow{\beta/\gamma} y'$ are transitions in X and Y, respectively, for some $\beta \in \mathbb{B}^l$. These transitions give $(y\,;\,z) \xrightarrow{\beta/\delta} (y'\,;\,z')$ in $Y\,;\,Z$, and thus $\big(x\,;\,(y\,;\,z)\big) \xrightarrow{\alpha/\delta} \Big(x'\,;\,\big(y'\,;\,z'\big)\Big)$ in $X\,;\,(Y\,;\,Z)$, as required.

For (iii), we have $(s\,;\,x) \xrightarrow{\alpha/\beta} (s\,;\,x')$ in $L_{id_k}\,;\,X$ (where s is the single state of L_{id_k}) iff we have $s \xrightarrow{\alpha/\alpha} s$ in L_{id_k} and $x \xrightarrow{\alpha/\beta} x'$ in X, giving $(x\,;\,s') \xrightarrow{\alpha/\beta} (x'\,;\,s')$ in $X\,;\,L_{id_l}$ (assuming s' is the single state of L_{id_l}), as required. Again, the converse follows a similar argument. □

Now, **2-LTS**, the category of 2-LTSs has the following structure:

- Objects are the natural numbers, $\mathbb{N}$,
- Arrows from k to l are the $(k,\,l)$-LTS isomorphism classes,
- The identity morphism for $k \in \mathbb{N}$, is L_{id_k},
- The composition of morphisms $X : (k,\,l)$ and $Y : (l,\,m)$, is $X\,;\,Y : (k,\,m)$, obtained using the variant of the product construction on LTSs described in Defn. 2.7.

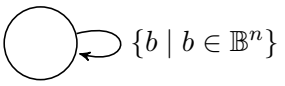


Figure 3.12: 2-LTS $L_{id_k} : (k,\,k)$

Proposition 3.34.
2-LTS *is a category.*

Proof. Prop. 3.33(i) ensures that 2-LTS composition is well-defined on equivalence classes of 2-LTSs, whilst (ii) and (iii) ensure that composition is associative and has L_{id_k} as identity. □

We have the following (strict) monoidal structure on **2-LTS**:

- Tensor is addition on objects and the modification of the product construction (Defn. 2.9) on morphisms,
- The identity object is 0,
- The associator and left/right unitors are identity natural isomorphisms.

We must assure ourselves that $\otimes$ is well-defined on equivalence classes of 2-LTSs, and is a bifunctor:

Proposition 3.35.
The following isomorphisms hold:

(i) Given 2-LTSs $X, X' : (k, l)$, $Y, Y' : (m, n)$, with $X \cong X'$ and $Y \cong Y'$, we have that $X \otimes Y \cong X' \otimes Y'$,

(ii) For any objects k, l, we have that $L_{id_{(k+l)}} \cong L_{id_k} \otimes L_{id_l}$

(iii) For 2-LTSs $X : (k, l)$, $Y : (l, m)$, $Z : (n, o)$, $W : (o, p)$, we have $(X \,;\, Y) \otimes (Z \,;\, W) \cong (X \otimes Z) \,;\, (Y \otimes W)$

Proof. For (i), the argument follows that of Prop. 3.33(i). For (ii) we use that $\mathbb{B}^{(k+l)} = \left\{ xy \;\middle|\; x \in \mathbb{B}^k, y \in \mathbb{B}^l \right\}$ and the obvious isomorphism between x and (x, x). Finally, for (iii), we have a transition $\left((x_1 \,;\, y_1) \otimes (x_2 \,;\, y_2)\right) \xrightarrow{\alpha\beta/\gamma\delta} \left(\left(x_1' \,;\, y_1'\right) \otimes \left(x_2' \,;\, y_2'\right)\right)$ in $(X \;;\; Y \otimes Z \;;\; W)$ iff we have transitions $(x_1 \,;\, y_1) \xrightarrow{\alpha/\gamma} \left(x_1' \,;\, y_1'\right)$ in $X \;;\; Y$ and $(x_2 \,;\, y_2) \xrightarrow{\beta/\delta} \left(x_2' \,;\, y_2'\right)$ in $(Z \;;\; W)$, iff there exists $\epsilon \in \mathbb{B}^l$ and $\zeta \in \mathbb{B}^o$ such that we have $x_1 \xrightarrow{\alpha/\epsilon} x_1'$ in X, $y_1 \xrightarrow{\epsilon/\gamma} y_1'$ in Y, $x_2 \xrightarrow{\beta/\zeta} x_2'$ in Z, $y_2 \xrightarrow{\zeta/\delta} y_2'$ in W. Then, we have $(x_1 \otimes x_2) \xrightarrow{\alpha\beta/\epsilon\zeta} \left(x_1' \otimes x_2'\right)$ in $X \otimes Z$, and $(y_1 \otimes y_2) \xrightarrow{\epsilon\zeta/\gamma\delta} \left(y_1' \otimes y_2'\right)$ in $Y \otimes W$, giving $\left((x_1 \otimes x_2) \,;\, \left(x_1' \otimes x_2'\right)\right) \xrightarrow{\alpha\beta/\gamma\delta} \left((y_1 \otimes y_2) \,;\, \left(y_1' \otimes y_2'\right)\right)$ in $(X \otimes Z) \;;\; (Y \otimes W)$, as required. □

This is enough to prove that **2-LTS** is (strict) monoidal:

Proposition 3.36.
2-LTS *is a strict monoidal category.*

Proof. The tensor product is addition on objects, which is strictly associative, and has 0 as identity. The required coherence conditions of Defn. 3.8 are then identities. Finally, by Prop. 3.35, we have the required proof of bifunctoriality. □

As we did for PNBs, we may also embed k-permutations into a 2-LTS:

Definition 3.37 (Permutation 2-LTS)**.**
A 2-LTS, $L_\sigma : (k, k)$, is a Permutation 2-LTS iff σ is a k-permutation, it has a single state, x, and transitions:

$$\left\{ x \xrightarrow{\alpha} x \;\middle|\; \alpha \in \left\{ \lceil o \rceil / \lceil \sigma(o) \rceil \;\middle|\; o \in 2^{\underline{k}} \right\} \right\}$$

L_σ is illustrated in Fig. 3.13.

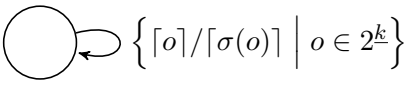


Figure 3.13: 2-LTS $L_\sigma : (k,\, k)$

A specific permutation 2-LTS is that which embeds the $\mathsf{sw}(k,\, l)$ permutation: for $k, l \in \mathbb{N}$, there is a 2-LTS, $L_{\mathsf{sw}(k,\,l)} : (k+l,\, l+k)$, illustrated in Fig. 3.14. $L_{\mathsf{sw}(k,\,l)}$ has a single state, and has self-loops labelled with $\left\{\alpha\beta/\beta\alpha \;\middle|\; \alpha \in \mathbb{B}^k, \beta \in \mathbb{B}^l\right\}$.

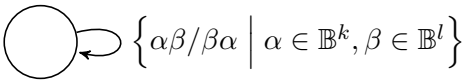


Figure 3.14: 2-LTS $L_{\mathsf{sw}(k,\,l)} : (k+l,\, l+k)$

Proposition 3.38.
For $X : (k,\, l)$ and $Y : (m,\, n)$, the following holds:

$$(X \otimes Y)\ ;\ L_{\mathsf{sw}(l,\,n)} \cong L_{\mathsf{sw}(k,\,m)}\ ;\ (Y \otimes X)$$

Proof. Immediate from the definitions of 2-LTS composition and $L_{\mathsf{sw}(-,\,-)}$. Letting z be the single state of $L_{\mathsf{sw}(l,\,n)}$ and w be that of $L_{\mathsf{sw}(k,\,m)}$, we have:

$$\begin{aligned}
& & \left((x \otimes y)\ ;\ z\right) \xrightarrow{\alpha\gamma/\delta\beta} \left(\left(x' \otimes y'\right)\ ;\ z\right) \in X \otimes Y\ ;\ L_{\mathsf{sw}(l,\,n)} \\
\iff & & (x \otimes y) \xrightarrow{\alpha\gamma/\beta\delta} \left(x' \otimes y'\right) \in X \otimes Y \\
\iff & & x \xrightarrow{\alpha/\beta} x' \in X, \text{ and} \\
& & y \xrightarrow{\gamma/\delta} y' \in Y \\
\iff & & (y \otimes x) \xrightarrow{\gamma\alpha/\delta\beta} \left(y' \otimes x'\right) \in Y \otimes X \\
\iff & & \left((w \otimes y)\ ;\ x\right) \xrightarrow{\alpha\gamma/\delta\beta} \left(\left(w \otimes y'\right)\ ;\ x'\right) \in L_{\mathsf{sw}(k,\,m)}\ ;\ Y \otimes X
\end{aligned}$$

□

Proposition 3.39.
The following are isomorphisms:

(a) $L_{\mathsf{sw}(k,\,l+m)} \cong (L_{\mathsf{sw}(k,\,l)} \otimes L_{id_m})\ ;\ (L_{id_l} \otimes L_{\mathsf{sw}(k,\,m)})$

(b) $L_{\mathsf{sw}(k+l,\,m)} \cong (L_{id_k} \otimes L_{\mathsf{sw}(l,\,m)})\ ;\ (L_{\mathsf{sw}(k,\,m)} \otimes L_{id_l})$

Proof. We give a proof for (b), the proof of (a) is similar:

Observe that L_{id_k}, $L_{\mathsf{sw}(l,\,m)}$, $L_{\mathsf{sw}(k,\,m)}$ and L_{id_l} all have a single state, thus their composition also has a single state.

The transitions of $L_{\mathsf{sw}(k+l,\,m)}$ are labelled by:

$$\left\{\alpha\beta/\beta\alpha \;\middle|\; \alpha \in \mathbb{B}^{k+l}, \beta \in \mathbb{B}^{m}\right\} = \left\{\gamma\delta\beta/\beta\gamma\delta \;\middle|\; \gamma \in \mathbb{B}^{k}, \delta \in \mathbb{B}^{l}, \beta \in \mathbb{B}^{m}\right\}$$

On the RHS, we have that $(L_{id_k} \otimes L_{\mathsf{sw}(l,\,m)})$ has transitions labelled with:

$$\left\{\gamma\delta\beta/\gamma\beta\delta \;\middle|\; \gamma \in \mathbb{B}^{k}, \delta \in \mathbb{B}^{l}, \beta \in \mathbb{B}^{m}\right\}$$

and $(L_{\mathsf{sw}(k,\,m)} \otimes L_{id_l})$ has transitions labelled with:

$$\left\{\gamma\beta\delta/\beta\gamma\delta \;\middle|\; \gamma \in \mathbb{B}^{k}, \delta \in \mathbb{B}^{l}, \beta \in \mathbb{B}^{m}\right\}$$

thus $(L_{id_k} \otimes L_{\mathsf{sw}(l,\,m)})$; $(L_{\mathsf{sw}(k,\,m)} \otimes L_{id_l})$ has transitions labelled by

$$\left\{\gamma\delta\beta/\beta\gamma\delta \;\middle|\; \gamma \in \mathbb{B}^{k}, \delta \in \mathbb{B}^{l}, \beta \in \mathbb{B}^{m}\right\}$$

as required. □

Proposition 3.40.
We have the following isomorphism:

$$L_{\mathsf{sw}(k,\,l)} \,;\, L_{\mathsf{sw}(l,\,k)} \cong L_{id_{k+l}}$$

Proof. $L_{\mathsf{sw}(k,\,l)}$ has transitions labelled by

$$\left\{\alpha\beta/\beta\alpha \;\middle|\; \alpha \in \mathbb{B}^{k}, \beta \in \mathbb{B}^{l}\right\}$$

and $L_{\mathsf{sw}(l,\,k)}$ has transitions labelled by

$$\left\{\beta\alpha/\alpha\beta \;\middle|\; \beta \in \mathbb{B}^{l}, \alpha \in \mathbb{B}^{k}\right\}$$

by the definition of composition, $L_{\mathsf{sw}(k,\,l)}$; $L_{\mathsf{sw}(l,\,k)}$ has transitions labelled by

$$\left\{\alpha\beta/\alpha\beta \;\middle|\; \alpha \in \mathbb{B}^{k}, \beta \in \mathbb{B}^{l}\right\} = \left\{\gamma/\gamma \;\middle|\; \gamma \in \mathbb{B}^{k+l}\right\}$$

as required. □

We can now prove that **2-LTS** is a strict symmetric monoidal category, and indeed, a PROP:

Proposition 3.41.
2-LTS *is a strict symmetric monoidal category.*

Proof. By Propositions 3.38, 3.39 and 3.40. □

Proposition 3.42.
2-LTS *is a PROP.*

Proof. By propositions 3.34, 3.36 and 3.41. □

3.4 Mapping between PNB and 2-LTS

Given $N : (k, l)$, we can generate its 2-LTS statespace, $\langle\!\langle N \rangle\!\rangle : (k, l)$, using the firing semantics described in §2.6.2.

This mapping respects the identity PNB:

Proposition 3.43.
For any $k \in \mathbb{N}$, $\left\langle\!\left\langle P_{id_k} \right\rangle\!\right\rangle \cong L_{id_k}$.

Proof. There are no places in P_{id_k}, giving only one possible state in $\left\langle\!\left\langle P_{id_k} \right\rangle\!\right\rangle$, corresponding to the empty marking. Each transition of P_{id_k} is always enabled, allowing arbitrary subsets of transitions to be fired, giving transitions in $\left\langle\!\left\langle P_{id_k} \right\rangle\!\right\rangle$ labelled by each element of $\left\{ x \;\middle|\; x \in \mathbb{B}^k \right\}$. Therefore, we have $\left\langle\!\left\langle P_{id_k} \right\rangle\!\right\rangle = L_{id_k}$, as required. □

and further, respects the compositions of PNBs:

We abuse notation when referring to states of the 2-LTS corresponding to a PNB composed of N and M. Such states are markings of the underlying PNB; that is, some $M \subseteq 2^{\text{places}(N) \uplus \text{places}(M)}$, indeed, we can partition M into x, a marking of N and y, a marking of M, justifying our notation (x, y).

Proposition 3.44.
For any pair of PNBs, $N : (k, l)$ *and* $M : (l, m)$*, we have that*

$$\langle\!\langle N \,;\, M \rangle\!\rangle \cong \langle\!\langle N \rangle\!\rangle \,;\, \langle\!\langle M \rangle\!\rangle$$

Proof. The states of $\langle\!\langle N \,;\, M \rangle\!\rangle$ are of the form $2^{\text{places}(N) \uplus \text{places}(M)}$, isomorphic to those of $\langle\!\langle N \rangle\!\rangle \,;\, \langle\!\langle M \rangle\!\rangle$, which are of the form $2^{\text{places}(N)} \times 2^{\text{places}(M)}$. By [61, Theorem 3.8], transitions, $(x, y) \xrightarrow{\alpha/\beta} (x', y')$, exist in $\langle\!\langle N \,;\, M \rangle\!\rangle$ iff, for $\gamma \in \mathbb{B}^l$, there are transitions $x \xrightarrow{\alpha/\gamma} x'$ in $\langle\!\langle N \rangle\!\rangle$ and $y \xrightarrow{\gamma/\beta} y'$ in $\langle\!\langle M \rangle\!\rangle$, corresponding to transitions in $\langle\!\langle N \rangle\!\rangle \,;\, \langle\!\langle M \rangle\!\rangle$. □

It follows that $\langle\!\langle - \rangle\!\rangle$: **PNB** $\to$ **2-LTS** is a functor:

Proposition 3.45.
$\langle\!\langle - \rangle\!\rangle$ *is a functor: identity on objects and firing semantics (Defn. 2.34) on morphisms.*

Proof. By propositions 3.43 and 3.44. □

Lemma 3.46.
For any pair of PNBs, $N : (k,\, l)$ *and* $M : (m,\, n)$*, we have:*

$(x, y) \xrightarrow{\alpha\gamma/\beta\delta} (x', y') \in \langle\!\langle N \otimes M \rangle\!\rangle$ *iff there are* $x \xrightarrow{\alpha/\beta} x' \in \langle\!\langle N \rangle\!\rangle$ *and* $y \xrightarrow{\gamma/\delta} y' \in \langle\!\langle M \rangle\!\rangle$*.*

Proof. ($\Rightarrow$) Suppose we have $(x, y) \xrightarrow{\alpha\gamma/\beta\delta} (x', y')$ in $\langle\!\langle N \otimes M \rangle\!\rangle$. There is a contention-free set of enabled transitions, $U \in \mathrm{trans}(N \otimes M)$. Indeed, we can partition U into (contention-free, enabled) $V \subseteq \mathrm{trans}(N)$ and $W \subseteq \mathrm{trans}(M)$, with $\lceil {}^\bullet V \rceil = \alpha$, $\lceil V^\bullet \rceil = \beta$, $\lceil {}^\bullet W \rceil = \gamma$ and $\lceil W^\bullet \rceil = \delta$, which correspond to the required transitions in $\langle\!\langle N \rangle\!\rangle$ and $\langle\!\langle M \rangle\!\rangle$.

($\Leftarrow$) Suppose we have $x \xrightarrow{\alpha/\beta} x'$ in $\langle\!\langle N \rangle\!\rangle$ and $y \xrightarrow{\gamma/\delta} y'$ in $\langle\!\langle M \rangle\!\rangle$; then, there exist contention-free, enabled $V \subseteq \mathrm{trans}(N)$ and $W \subseteq \mathrm{trans}(M)$, with $\lceil {}^\bullet V \rceil = \alpha$, $\lceil V^\bullet \rceil = \beta$, $\lceil {}^\bullet W \rceil = \gamma$, and $\lceil W^\bullet \rceil = \delta$. By definition, $V \uplus W \subseteq \mathrm{trans}(N \otimes M)$, giving the required transition in $\langle\!\langle N \otimes M \rangle\!\rangle$. □

Proposition 3.47.
For any pair of PNBs, $N : (k,\, l)$ *and* $M : (m,\, n)$*, we have that*

$$\langle\!\langle N \rangle\!\rangle \otimes \langle\!\langle M \rangle\!\rangle \cong \langle\!\langle N \otimes M \rangle\!\rangle$$

Proof. Similar to the proof of Prop. 3.44, employing Lem. 3.46 to show equality of labelled transitions. □

Proposition 3.48.
$\langle\!\langle - \rangle\!\rangle$ *is a strict monoidal functor.*

Proof. The isomorphic 2-LTSs of Prop. 3.47 form[1] a natural transformation with each component being an identity map. Similarly, the unit morphism is an identity map, since $\langle\!\langle - \rangle\!\rangle$ is the identity mapping on objects. □

Proposition 3.49.
We have a 2-LTS isomorphism, $L_{sw(k,\,l)} \cong \left\langle\!\!\left\langle P_{sw(k,\,l)} \right\rangle\!\!\right\rangle$.

Proof. The PNB $P_{\mathsf{sw}(k,\,l)}$ (shown in Fig. 3.10) has no places, hence its 2-LTS statespace has a single place corresponding to the empty marking. Since the same permutation is embedded in the PNB and 2-LTS, by definition, the generated 2-LTS statespace will have transitions labelled the same as those in $L_{\mathsf{sw}(k,\,l)}$, as required. □

Proposition 3.50.
$\langle\!\langle - \rangle\!\rangle$ *is a strict symmetric monoidal functor.*

[1] Recall that morphisms of **2-LTS** are isomorphism classes of 2-LTSs.

Proof. Since α is formed of identities, the commutativity of Fig. 3.8 is reduced to requiring equality of the top and bottom morphisms; since morphisms of **2-LTS** are equivalence classes, this requirement is satisfied by Prop. 3.49. □

This statement is tantamount to saying that $\langle\!\langle - \rangle\!\rangle$ is a homomorphism of PROPs.

Proposition 3.51.
$\langle\!\langle - \rangle\!\rangle$ is a homomorphism of PROPs.

Proof. Immediate. □

Observe that Prop. 3.44 and Prop. 3.47 are precisely equivalent to the standard definition of *compositionality*, which states that the semantics of a composite PNB is determined by the semantics of the component PNBs. Put differently, there is no emergent behaviour when composing the semantics of the components — the semantics embody a precise account of all possible behaviours of the components. Indeed, we revisit compositionality, in its standard setting, in Prop. 5.4.

3.5 Encoding Reachability

We can annotate PNBs with a pair of markings, encoding a particular initial and target marking, as described in §2.6.7. Intuitively, such an annotated PNB represents a reachability problem: can the target marking be reached from the initial marking?

3.5.1 The category of mPNBs

mPNBs form a strict symmetric monoidal category, **mPNB**, by lifting the various constructions from PNBs to mPNBs. We do not offer proofs for the definitions in this section, since they follow simply from the corresponding definitions on PNBs:

Definition 3.52.
Synchronous composition of mPNBs, (N, m, n) and (M, m', n') is defined as: $(N, m, n) \,;\, (M, m', n') \stackrel{\text{def}}{=} (N \,;\, M, m \uplus m', n \uplus n')$

The category of mPNBs has structure:

- Objects are the natural numbers, $\mathbb{N}$,
- Arrows from k to l are the isomorphism classes of mPNBs: $[(N, m, n) : (k,\, l)]$,
- The identity morphism for $k \in \mathbb{N}$, is $(P_{id_k}, \varnothing, \varnothing)$

- The composition of morphisms is per Defn. 3.52.

Proposition 3.53.
mPNB *is a category.* □

Definition 3.54.
Tensor composition of mPNBs, (N, m, n) *and* (M, m', n') *is defined as:* $(N, m, n) \otimes (M, m', n') \stackrel{\text{def}}{=} (N \otimes M, m \uplus m', n \uplus n')$

mPNB has a monoidal structure:

- The tensor product is addition on objects and the operation described in Defn. 3.54 on morphisms,
- The unit is 0,
- The associator and left/right unitors are identity natural isomorphisms.

Proposition 3.55.
mPNB *is a strict monoidal category.* □

Furthermore, **mPNB** has a symmetric structure, given by the following morphism:

$$(P_{\mathsf{sw}(k,l)}, \varnothing, \varnothing) : (k + l,\, l + k)$$

Proposition 3.56.
mPNB *is a symmetric monoidal category.* □

3.5.2 The category of 2-NFAs

A 2-NFA is a 2-LTS, together with a chosen initial state and a set of accepting states, as defined in §2.3.2.

We then have that **2-NFA**, the category of 2-NFAs has structure:

- Objects are the natural numbers, $\mathbb{N}$,
- Arrows from k to l are the isomorphism classes of 2-NFAs: $[(X, i, A) : (k,\, l)]$,
- The identity morphism for $k \in \mathbb{N}$, is $(L_{id_k}, 0, \{0\})$, where 0 is the state of L_{id_k},
- The composition of morphisms is per Defn. 2.14.

Proposition 3.57.
2-NFA *is a category.* □

We have a monoidal structure on **2-NFA**:

- The tensor product is addition on objects and the operation described in Defn. 2.15 on morphisms,
- The unit object is 0,
- The associator and left/right unitors are identity natural isomorphisms.

Proposition 3.58.
2-NFA *is a strict monoidal category.* □

Furthermore, **2-NFA** has a symmetric structure, given by the following morphism:

$$(L_{\mathsf{sw}(k,l)}, 0, \{0\}) : (k+l,\, l+k)$$

where 0 is the single state of $L_{\mathsf{sw}(k,l)}$.

Proposition 3.59.
2-NFA *is a symmetric monoidal category.* □

3.5.3 Mapping between mPNB, 2-NFA, PNB and 2-LTS

mPNBs can be given a 2-NFA semantics:

Definition 3.60 (2-NFA semantics of mPNBs)**.**
For a mPNB, (N, m, n)*, its 2-NFA semantics,* ${}_m\langle\!\langle (N, m, n) \rangle\!\rangle$*, is* (X, i, A)*, where* X *is the 2-LTS semantics of* N*,* i *is the state corresponding to* m*, and* A *is the (singleton set containing the) state corresponding to* n*.*

Proposition 3.61.
${}_m\langle\!\langle - \rangle\!\rangle : \mathbf{mPNB} \to$ **2-NFA** *is a strict symmetric monoidal functor, identity on objects and 2-NFA semantics on morphisms.* □

Furthermore, we have two forgetful (strict symmetric monoidal) functors, mapping **mPNB** to **PNB**, and **2-NFA** to **2-LTS**. The forgetful functor on **mPNB**, $U_{\mathsf{PNB}} : \mathbf{mPNB} \to \mathbf{PNB}$ is identity on objects, and forgets the markings on morphisms:

$$U_{\mathsf{PNB}}((N, m, n)) \stackrel{\text{def}}{=} N$$

Similarly, we can forget the initial and accepting states of a 2-NFA, to obtain a 2-LTS. The forgetful functor $U_{\text{2-LTS}}$: **2-NFA** $\to$ **2-LTS** is identity on objects, and on arrows:

$$U_{\text{2-LTS}}((X, i, A)) \stackrel{\text{def}}{=} X$$

Indeed, it is equivalent to take the 2-NFA semantics of a mPNB, before forgetting the initial and accepting states, or to forget the initial/target markings of the mPNBbefore taking the 2-LTS semantics. This equivalence is embodied in the commuting diagram of Fig. 3.15.

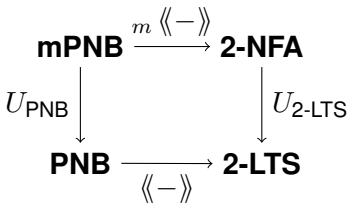


Figure 3.15: Commuting diagram illustrating the categories' relationships

3.5.4 Summary

In this chapter we have illustrated the categorical structure of PNBs and 2-LTSs and the relationship between these categories. We highlighted that the functorial relationship between the categories **PNB** and **2-LTS** is equivalent to the property of compositionality and lifted the constructions to *marked* PNBs and corresponding 2-NFAs leading to the categories **mPNB** and **2-NFA** and illustrated their inter-relations.

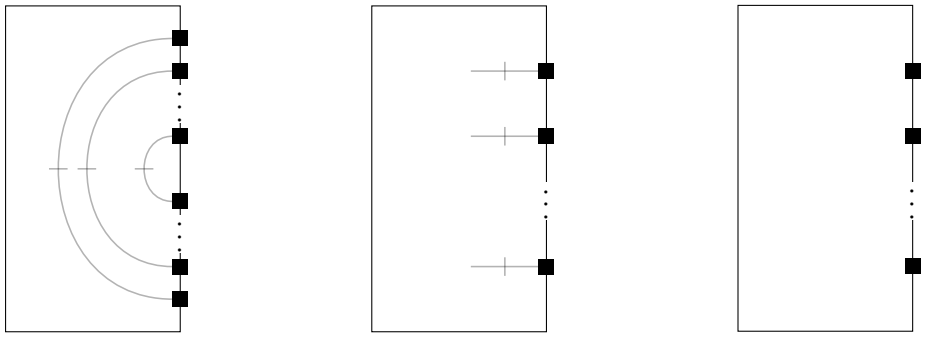
Chapter 4

Benchmarks and a Domain Specific Language for Net Compositions

In this chapter we introduce the example systems that we will use to demonstrate and benchmark our reachability checking techniques. Several of these examples are taken from Corbett's benchmark suite [145], commonly used as reference benchmarks in the Petri net literature. However, here we give alternative, *parameterised* specifications in a *component-wise* manner; we propose that such specifications are more natural and easier to construct and reason about, relative to the standard monolithic definitions. Indeed, for this purpose, we motivate and introduce a *Domain Specific Language* (DSL) for constructing such systems, proving that it ensures that invalid constructions cannot be formed. This chapter is formed from two of the author's papers; the core was presented at Petri Nets 2014 [2], while some of the example systems are taken from [146].

4.1 Component-wise Specification of Nets

Before introducing and specifying our example net systems, we describe a collection of "wiring" nets that we frequently use in specifications. Such wiring nets do not contain places, only transitions connecting the boundary ports in various ways. Consider the "left-end" components shown in Fig. 4.1 (such named since they are to be composed on the left of other components). Each illustrated component is a representation of a parametric family of nets over $k \in \mathbb{N} \setminus \{0\}$; for notational convenience we drop the subscript when $k = 1$. The ETA_k net, shown in Fig. 4.1a, has $2k$ right boundary ports, with transitions connecting the ports $0 \leq l < k$ to $2k - 1 - l$.

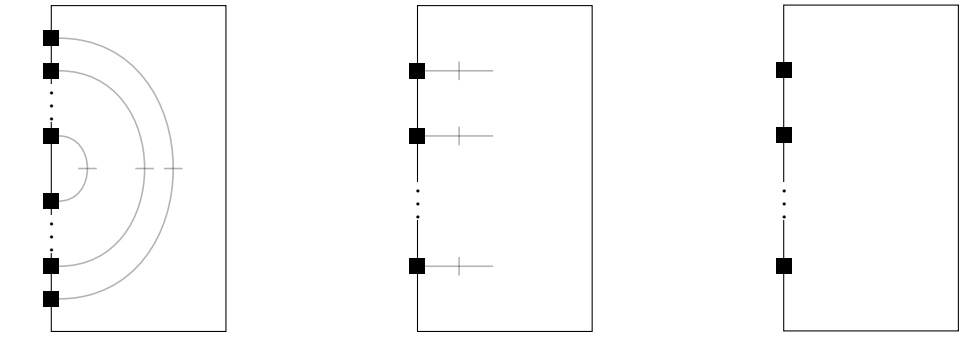


(a) ETA$_k$: $(0,\, 2k)$ (b) $\top_k$: $(0,\, k)$ (c) $\uparrow_k$: $(0,\, k)$

Figure 4.1: Left-end Component Families

The $\top_k$ net, shown in Fig. 4.1b, has k right boundary ports, each with a single corresponding transition; composing with such a net completes the partial specification of any transitions connected to the shared boundary port, but without adding further connections to places. Finally, the $\uparrow_k$ net, shown in Fig. 4.1c, has k right boundary ports, but no transitions; composing with this net terminates any transitions, t, connected to the shared boundary ports in the other component — there are no transitions in $\uparrow$ to synchronise with and thus t cannot appear as part of a transition in the composite net.

Similarly, we have right-end component nets, as shown in Fig. 4.2 which are reflections of their left-end counterparts.



(a) EPSILON$_k$: $(2k,\, 0)$ (b) $\bot_k$: $(k,\, 0)$ (c) $\downarrow_k$: $(k,\, 0)$

Figure 4.2: Right-end Component Families

Finally, we often use identity net components: such components have k left and right boundary ports, and k transitions, connecting each left boundary port to the corresponding right boundary port, as illustrated in Fig. 4.3. Such nets preserve the transitions of any components they are composed with (in a formal sense: PNBs form a category, with this family of nets as the identity morphisms — see §3.2).

We now introduce two basic net systems, the k-bit Buffer and Token Ring, before introducing a convenient programming language to specify such systems.

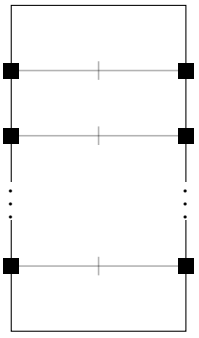


Figure 4.3: $\mathsf{ID}_k : (k, k)$

4.1.1 k-bit Buffer

A k-buffer net (taken from [82]) models a system of k buffer cells that can each either be "full" or "empty". Such a system is naturally described in a compositional manner: to form an k-buffer, simply compose k suitable 1-buffer components.

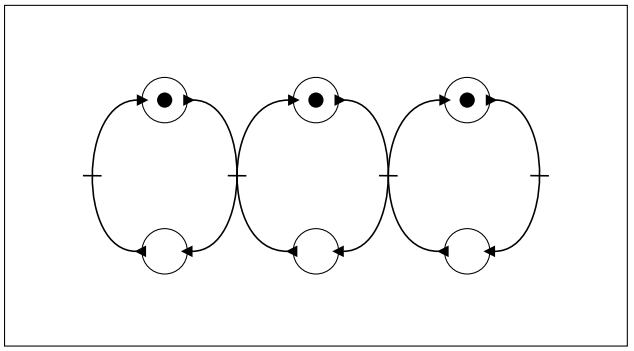
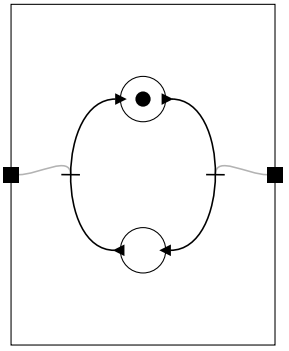


Figure 4.4: $\mathsf{BUFFER}(3)$

For example, $\mathsf{BUFFER}(3)$ is illustrated in Fig. 4.4. Taking a component-wise view, and using the single-buffer component illustrated in Fig. 4.5, the schematic for $\mathsf{BUFFER}(3)$ is illustrated in Fig. 4.6: with suitable terminators, we simply synchronously compose 3 buffers together.



(a) $\mathsf{BUFFER} : (1, 1)$

Figure 4.5: $\mathsf{BUFFER}(k)$ component net

Recall that to aid intuition, we often mark (parts of) partially-specified transitions using a lighter shade, emphasising the fully specified structure of a component net.

Figure 4.6: Schematic of BUFFER(3)

Arbitrarily-sized buffers are defined:

$$\text{BUFFER}(k) \stackrel{\text{def}}{=} \top \,;\, \text{BUFFER}^k \,;\, \bot$$

4.1.2 Token Ring

Consider a model of a simple token ring network, taken from [147]. Such a network consists of k worker processes sharing a single token; a worker may only work if it holds the token. As an example, TOKENRING(3) is illustrated in Fig. 4.7. All workers are initially "able to work"; a token is "injected" into the system — upon receiving the token, a worker may choose to do work, or pass the token onto the next worker.

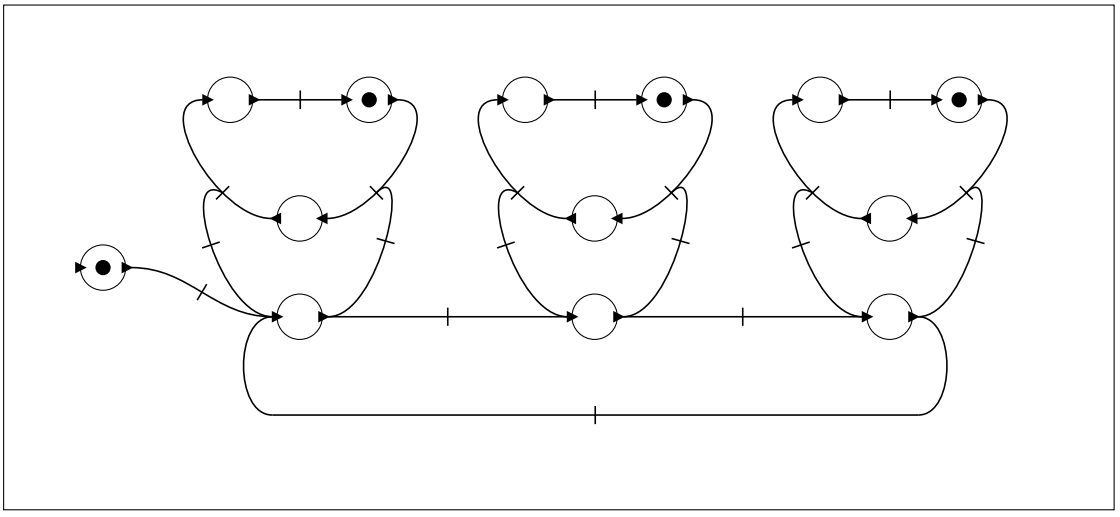
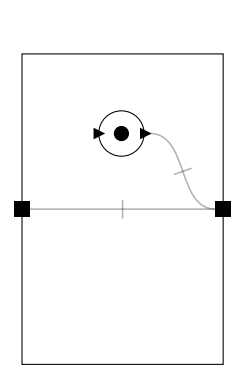
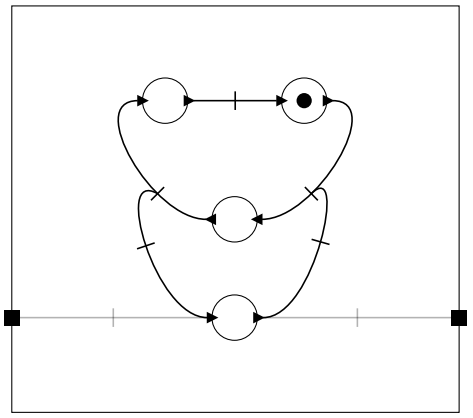


Figure 4.7: TOKENRING(3)

Again, a token ring system is naturally described in a compositional manner: each worker is a single component, as illustrated in Fig. 4.8b, that may interact (via transitions connected to its left/right boundary ports) with the workers on its left and right, in order to receive/send the token. The injector component, shown in Fig. 4.8a, simply holds a token that can be injected, before acting as ID. The last worker is connected to the first, completing the ring. The schematic of TOKENRING(3) is illustrated in Fig. 4.9.



(a) INJECTOR : $(1, 1)$



(b) WORKER : $(1, 1)$

Figure 4.8: TOKENRING(k) components

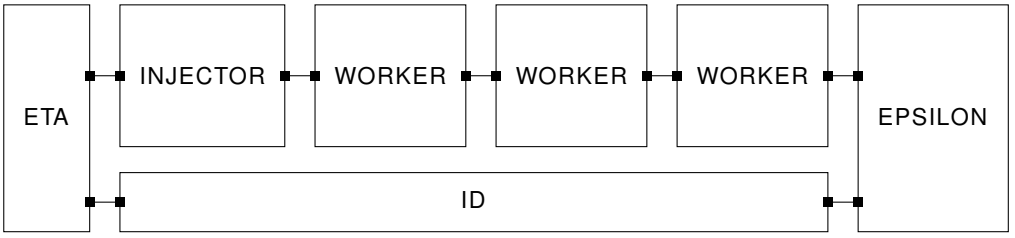


Figure 4.9: Schematic of TOKENRING(3)

TOKENRING(k) is defined:

$$\text{TOKENRING}(k) \stackrel{\text{def}}{=} \text{ETA} \;;\; \Big(\big(\text{INJECTOR} \;;\; \text{WORKER}^k\big) \otimes \text{ID}\Big) \;;\; \text{EPSILON}$$

4.1.3 A Language for Net Composition

In the previous examples we demonstrated the algebraic description of Petri net systems in terms of their component PNBs. We now motivate using a Domain Specific Language (DSL), PNBml, that evaluates to the algebra of PNBs, but adds expressive high-level functional programming language features.

Consider again the algebraic description of a token ring network; how might we generate the algebraic expression representing a similar network of, say, 10 worker components? The simple approach given is to explicitly write the term

$$\text{ETA} \;;\; ((\text{INJECTOR} \;;\; \text{WORKER} \;;\; \text{WORKER} \cdots \;;\; \text{WORKER}) \otimes \text{ID}) \;;\; \text{EPSILON}$$

where WORKER appears precisely 10 times. However, this is clearly not scalable: for large numbers of components, or more complex components (that may themselves be

formed of component compositions) it becomes a nuisance to construct such low-level expressions, and furthermore, ensure that they are correctly composed.

Consider the following expression that we might (accidentally) write when composing worker nets, WORKER : $(1,\ 1)$:

$$\text{WORKER}\ ;\ (\text{WORKER} \otimes \text{WORKER})$$

the resulting net is undefined, since the two nets being synchronously composed have different size boundaries — composition on such nets is not definable in a unique way[1]. Indeed, it is easy to observe that WORKER $\otimes$ WORKER : $(2,\ 2)$. Yet, for the synchronous composition to be well-defined, we must have that WORKER $\otimes$ WORKER has boundaries $(1, k)$ for any k, a contradiction, indicating an invalid expression. To ensure that we disallow such invalid expressions, we use an appropriate notion of *type*, which can be ascribed to expressions, to ensure that incompatible nets are never composed during evaluation. We will return to the description of the type system later in this section.

When describing complex components, we would like any repeated sub-components to be described only once, rather than each time they are used. Specifically, we would like to bind a name to a sub-expression and be able to use that name to refer to the sub-expression. For example, we might consider an extended token ring network model where each worker task, WORKER, is comprised of two sub-components: WORKER_1 and WORKER_2. Using name binding, we might describe a sequence of such tasks as:

$$\textbf{bind}\ w = (\text{WORKER}_1\ \textbf{;}\ \text{WORKER}_2)\ \textbf{in}\ w\ \textbf{;}\ w\ \textbf{;}\ \ldots\ \textbf{;}\ w$$

Another improvement that we can make is to introduce a compact form for expressions that feature sequences of a repeated component, with *parameterised* length.

$$\begin{aligned}&\textbf{bind}\ w = (\text{WORKER}_1\ \textbf{;}\ \text{WORKER}_2)\ \textbf{in}\\ &\textbf{n\_sequence}\ k\ w\end{aligned}$$

where **n_sequence** $k\ c$ represents $((c\ ;\ c)\ ;\ c) \cdots ;\ c$ with c appearing k times, similarly to the syntax sugar introduced in Remark 2.8.

Finally, consider the procedure that takes an arbitrary-length sequence of workers and forms a ring by connecting the first to the last. Such a procedure can be written, using a lambda notation common to functional programming languages, as:

$$\boldsymbol{\lambda} x : \mathrm{Net}\langle 1,\ 1\rangle\ \textbf{.}\ \text{ETA}\ \textbf{;}\ (x \otimes \text{ID})\ \textbf{;}\ \text{EPSILON}$$

[1] In general there will be many ways to align the boundaries, giving different nets.

that is, take a suitable sequence of workers (with left and right boundaries of size 1), represent it by the variable x and perform the appropriate compositions to form the ring.

Other examples, such as those in the remainder of this section, are naturally *parametric* and can thus be compactly represented for any particular parameter choice. As examples, we may represent the BUFFER(k) system as:

$$\top \textbf{ ; n\_sequence } k \text{ BUFFER } ; \bot$$

and the TOKENRING(3) system shown in Fig. 4.7, with the expression:

$$\begin{aligned}
&\textbf{bind } makeRing = \boldsymbol{\lambda} x : \mathrm{Net}\langle 1,\, 1\rangle \textbf{ . } \text{ETA } ; (x \otimes \text{ID}) ; \text{EPSILON } \textbf{in} \\
&\textbf{bind } procs = \textbf{n_sequence } 3 \text{ WORKER } \textbf{in} \\
&makeRing\ (\text{INJECTOR } ; procs)
\end{aligned}$$

We defer formally introducing PNBml to §4.3. Instead, we now use it to give encodings of several well-known example systems.

4.1.4 Complete Trees

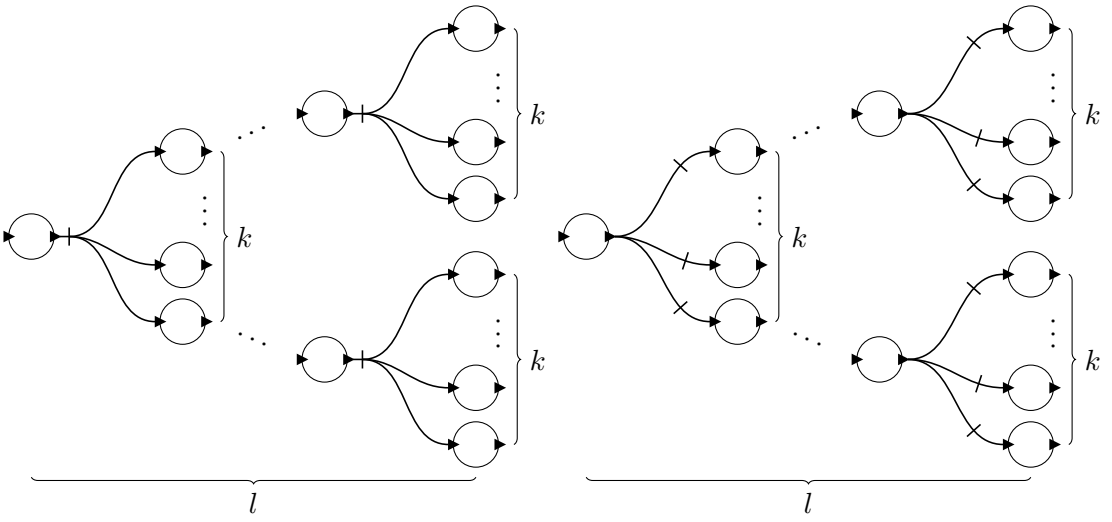
Another class of systems with regular structure that can exploit a component-wise specification are (full, complete) (k, l)-trees, with width k and depth l. In such trees, each non-leaf node has exactly k children (full) and every "level" of the tree other than the last is fully saturated (complete), i.e. there are k^m nodes at depth $m < l$. Together, these properties give the total number of nodes as:

$$\frac{1}{k-1}\left(k^{l+1} - 1\right)$$

for example, a $(3, 2)$-tree contains 13 nodes.

We identify 2 such classes of trees: *conjunction* trees, $T_\wedge(k, l)$, illustrated in Fig. 4.10a, and *disjunction* trees, $T_\vee(k, l)$, illustrated in Fig. 4.10b. A conjunction tree node is connected by a single transition to all of its (direct) children, whereas a disjunction tree's nodes have distinct transitions to each child.

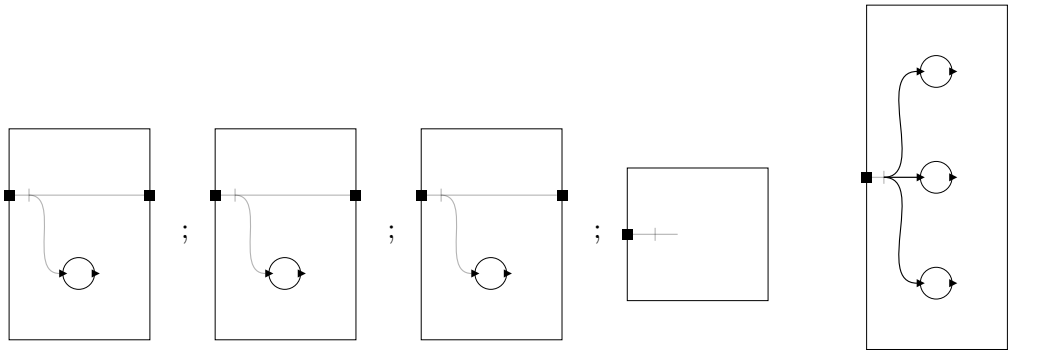
Such trees are naturally specified in a compositional manner. To see how, we first discuss how to construct a transition connecting the k leaf nodes in $T_\wedge(k, l)$. Let us consider this transition for $T_\wedge(3, 2)$; such a transition must connect to the first leaf place *and* the remaining 2 leaf places. We can represent this by a component with a single left and right boundary and a single transition connecting the left/right boundary ports and the in-port of its single leaf place. The intuition being that the transition connects to the in-port of the place and whatever else is connected to the right boundary port of the



(a) Conjunction tree, $T_\wedge(k,l)$ (b) Disjunction tree, $T_\vee(k,l)$

Figure 4.10: (k,l)-Tree Nets

component: if we sequentially compose 3 such components (illustrated in Fig. 4.12c) and a suitable "terminator" component (illustrated in Fig. 4.2b), as shown in Fig. 4.11a, we will obtain a component isomorphic to one with a single left boundary and transition connecting the 3 places' in-ports (illustrated in Fig. 4.11b). In a similar manner, we can give component-wise specifications for internal nodes, and thus the entire tree net.

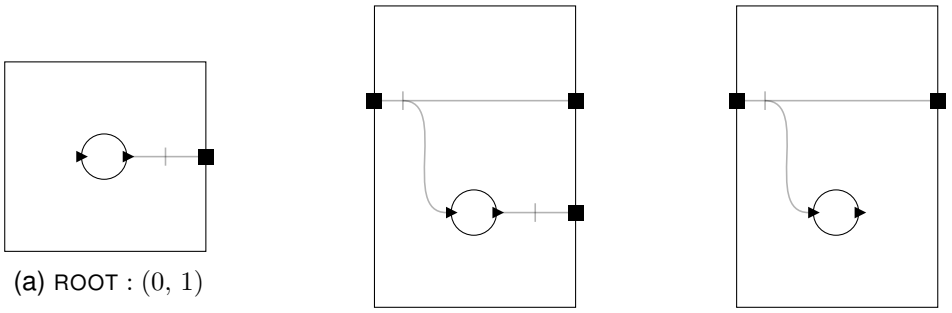


(a) 3 leaf components and a terminator (b) 3 leaf component

Figure 4.11: Isomorphic specification of 3 leaf nodes

We can now give the components and specification for $T_\wedge(k,l)$; the root, internal node and leaf components are shown in Fig. 4.12. Using PNBml syntax, $T_\wedge(k,S(l))$ is:

$T_\wedge(k,S(l)) \stackrel{\text{def}}{=}$ **bind** $leaves =$ **n_sequence** k LEAF$_\wedge$ **;** $\bot$ **in**
bind $addSubTree = \boldsymbol{\lambda} x : \mathrm{Net}\langle 1,\, 0\rangle$ **.** **n_sequence** k $\big($NODE$_\wedge$ **;** (ID $\otimes$ x)$\big)$ **;** $\bot$ **in**
bind $createSubTrees = \boldsymbol{\lambda} x : \mathbb{N}$ **.** **fold**$_\mathbb{N}$ x $leaves$ $addSubTree$ **in**
ROOT **;** $(createSubTrees\ l)$



(a) ROOT : $(0,\ 1)$ (b) NODE$_\wedge$: $(1,\ 2)$ (c) LEAF$_\wedge$: $(1,\ 1)$

Figure 4.12: $T_\wedge(k, l)$ component nets

Intuitively, $createSubTrees$ adds l "levels", to the bottom level of k leaf nodes, where each non-leaf level is k repetitions of the level before. Indeed, to get the correct depth, we must define the net system in terms of $S(l)$, not l.

Remark 4.1. A relevant design detail of our language to briefly discuss at this point is that of providing folds over $\mathbb{N}$ (which, as noted by Coquand [148], can obscure a program's meaning), instead of pattern matching on $\mathbb{N}$ and recursive definitions; indeed, had we allowed such features, the definition of $T_\wedge(k, l)$ can be expressed more directly:

$$
\begin{aligned}
T_\wedge(k,l) \stackrel{\text{def}}{=}\ & \textbf{bind}\ mkSubTree = \boldsymbol{\lambda} x : \mathbb{N}\ \textbf{.} \\
& \quad \textbf{match}\ x\ \textbf{with} \\
& \qquad 1 \Rightarrow \textbf{n\_sequence}\ k\ \text{LEAF}_\wedge\ \textbf{;}\ \bot \\
& \qquad S(m) \Rightarrow \textbf{n\_sequence}\ k\ \Big(\text{NODE}_\wedge\ \textbf{;}\ \big(\text{ID} \otimes (mkSubTree\ m)\big)\Big)\ \textbf{;}\ \bot \\
& \quad \textbf{in}\ \text{ROOT}\ \textbf{;}\ (mkSubTree\ l)
\end{aligned}
$$

Notice that the definitions of $leaves$ and $addSubTree$ have been inlined, and since we can pattern match on $\mathbb{N}$, we can directly define $T_\wedge(k, l)$, since the "base-case" pattern match[2] can be for 1, rather than 0. However, the choice to provide folds instead has the benefit that termination is guaranteed, since the structural recursion is "hidden" in the implementation of **fold**$_\mathbb{N}$. Indeed, in the presence of general recursion, non-terminating computations abound; static approximations are however possible, e.g. that of Abel and Altenkirch [149], which checks for decreasing-size arguments to recursive calls, but these, along with pattern matching, are non-trivial to implement.

Continuing after our slight digression, as an example, $T_\wedge(2, 2)$ is shown in Fig. 4.13, with the corresponding schematic shown in Fig. 4.14.

Having shown how to construct $T_\wedge(k, l)$, we now consider $T_\vee(k, l)$; we simply need to modify the node and leaf components from their definitions for $T_\wedge(k, l)$: rather than

[2]The pattern match is of course "incomplete", since there is no case for 0, but such a matter is not important for the purposes of our discussion.

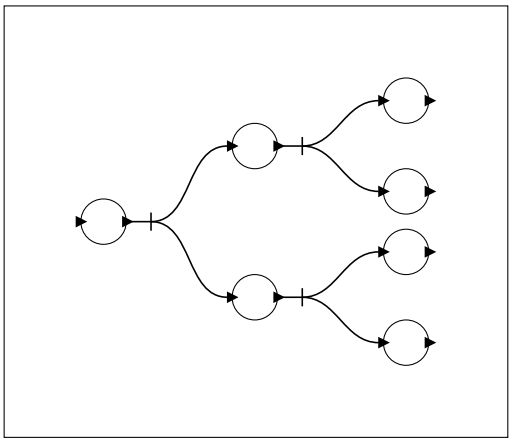


Figure 4.13: $T_\wedge(2,2)$

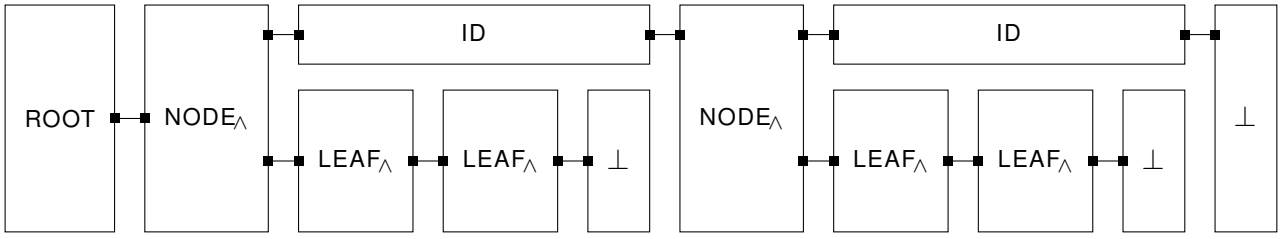


Figure 4.14: Schematic of $T_\wedge(2,2)$

single transitions into the node (leaf) and its sibling, we have separate transitions. The modified components are illustrated in Fig. 4.15.

(a) NODE$_\vee$: $(1,\,2)$ (b) LEAF$_\vee$: $(1,\,1)$

Figure 4.15: $T_\vee(k,l)$ component nets

Indeed, by replacing NODE$_\wedge$ with NODE$_\vee$ and LEAF$_\wedge$ with LEAF$_\vee$ in the specification, we obtain $T_\vee(k,l)$. As an example, $T_\vee(2,2)$ is illustrated in Fig. 4.16.

4.1.5 Cliques

A (directed) clique is a (directed) graph where there is an edge between every pair of nodes. Similarly, a clique net has transitions such that for every pair of distinct places, there are transitions to-and-from the places.

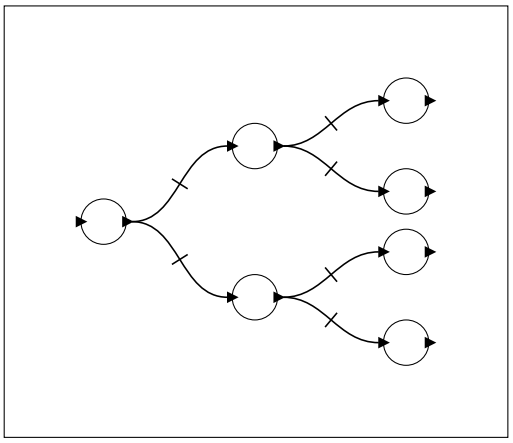


Figure 4.16: $T_V(2,2)$

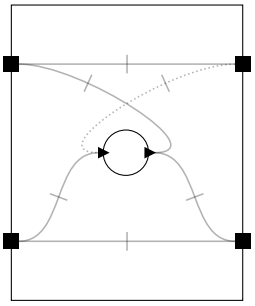


Figure 4.17: CLIQUE1 : $(2,\ 2)$

Perhaps surprisingly, a compact, component-wise specification can be given for cliques; if we specify a single "clique component", we can construct a (suitably terminated) sequence of such components to arrive at the specification of a clique. Indeed, a single clique component, CLIQUE1, is illustrated in Fig. 4.17. Such a component has transitions that allow it to produce tokens at the next and previous components, consume tokens from the next and previous components, and pass tokens through from left to right boundaries and vice-versa (it is these last transitions that allow, say, the 3^{rd} component to produce tokens at the 1^{st}, only via connections to the 2^{nd}. Using the previously-defined INJECTOR component, we can inject a single token into the clique system.

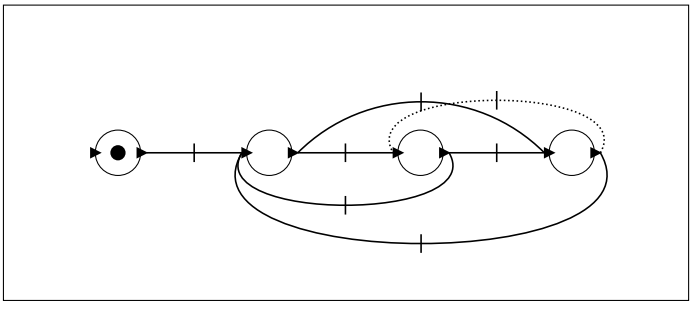


Figure 4.18: CLIQUE(3)

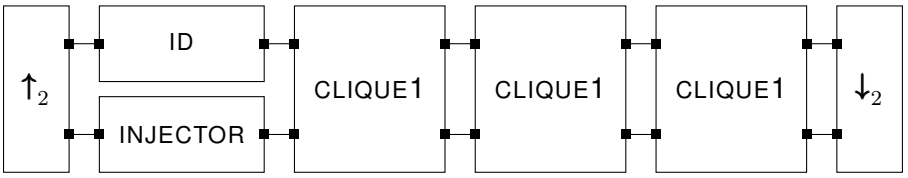


Figure 4.19: Schematic of CLIQUE(3)

As an example, CLIQUE(3) is illustrated in Fig. 4.18, with its schematic shown in Fig. 4.19. Cliques of arbitrary size are defined:

$$\text{CLIQUE}(k) \stackrel{\text{def}}{=} \uparrow_2 \text{ ; } (\text{ID} \otimes \text{INJECTOR}) \text{ ; } \textbf{n\_sequence } k \text{ CLIQUE1 ; } \downarrow_2$$

4.1.6 Powersets

A POWERSET(k) net has $k+1$ places. There is a chosen start place, S, with the remaining places being the elements of $\underline{k}$. There are 2^k transitions, all with a single source, S, and with targets being the elements of $2^{\underline{k}}$. For example, POWERSET(3) is illustrated in Fig. 4.20, where transition t_X connects S to all $x \in X$.

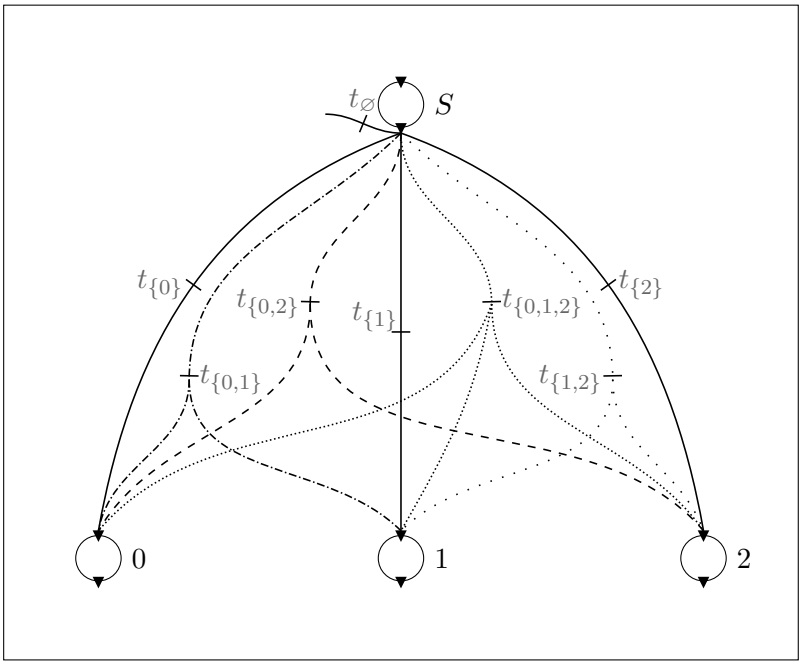
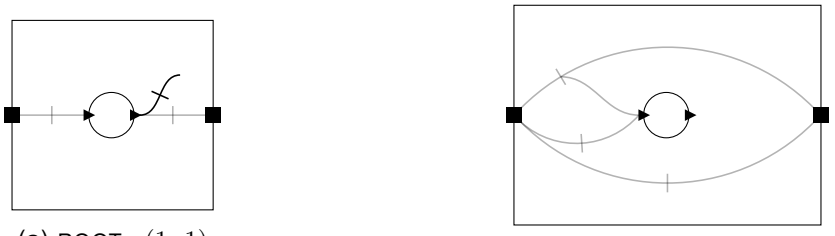


Figure 4.20: POWERSET(3)

To obtain a component-wise specification, consider partitioning the non-start places of the net into individual components. Then, the global transitions must be constructed from compositions of suitable local transitions. Locally, each incoming partially specified

transition should be connected to the place of the component (and other components) in three ways:

1. To the local place and any further specified places
2. To the local place alone,
3. Only to the further specified places.



(a) ROOT : $(1, 1)$ (b) POWERSET1 : $(1, 1)$

Figure 4.21: POWERSET(k) component nets

Such a component is illustrated in Fig. 4.21b. With the ROOT component illustrated in Fig. 4.21a (which either passes its token into a sequence of POWERSET1 components or not) and the INJECTOR component from TOKENRING(k), we have:

$$\text{POWERSET}(k) \stackrel{\text{def}}{=} \uparrow \text{ ; INJECTOR ; ROOT ; } \textbf{n\_sequence}\ k\ \text{POWERSET1 ; } \downarrow$$

The schematic for the POWERSET(3) net illustrated in Fig. 4.20, is shown in Fig. 4.22.

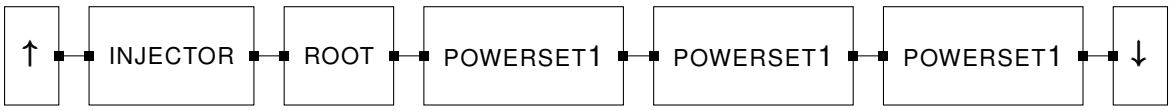


Figure 4.22: Schematic of POWERSET(3)

4.2 Benchmark Systems

In this section, we introduce several *more realistic* benchmark systems than those introduced thus far. Several of the systems are taken from Corbett's [145] benchmark suite, but have been reformulated in terms of compositional specifications. To arrive at the component-based specifications, we examined the original monolithic Petri net and Ada models, as they existed in the Petri net community, and decomposed them into components by hand. By doing so, the natural system descriptions in terms of component structure and component compositions can be explicitly presented.

4.2.1 Overtake Protocol

The OVERTAKE(k) net system simulates a queue of k automated cars that can overtake one another. To ensure that no collisions occur, a collection of locks are used: one between each pair of cars. To overtake, a car must hold both the lock in front of it, and the lock behind it; this prevents, for example, a car attempting to overtake a car that is itself overtaking.

To give a component-wise specification of such a system, we construct components that represent a single car, a single shared lock and then form a sequence of such components. Indeed, the component for a single car has simple local behaviour (where we name the states reached after each action):

1. Request the rear lock (ρ),
2. If request failed, reset (ι), if successful $(\dagger)$, request the front lock $(\star)$,
3. If request failed, release rear lock $(\sharp)$, if successful, $(\ddagger)$, perform overtake $(\odot)$,
4. Release the rear lock $(*)$,
5. Release the front lock (ι).

A component embodying such a local behaviour is illustrated in Fig. 4.23, where the "perform overtake" transition is highlighted in blue.

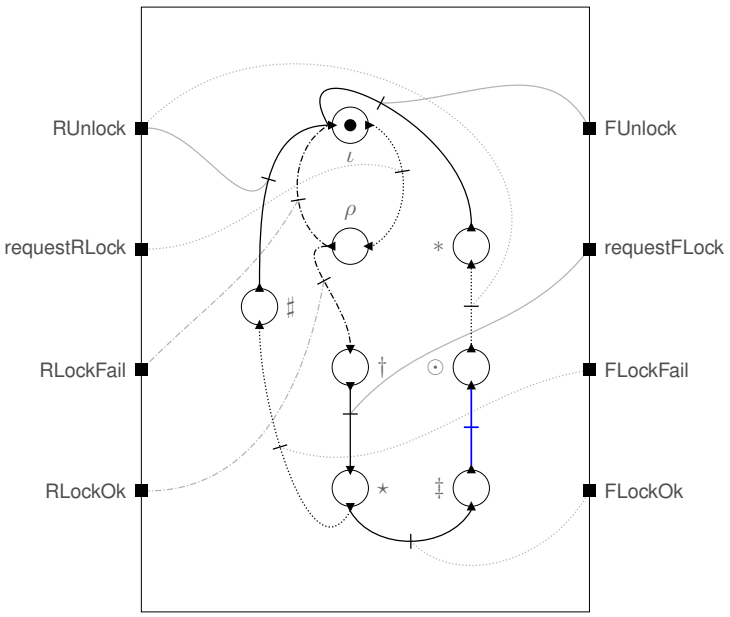


Figure 4.23: CAR : $(4, 4)$

The lock component has three states:

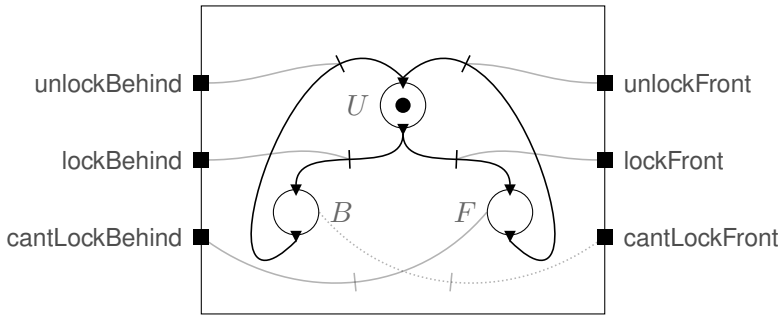
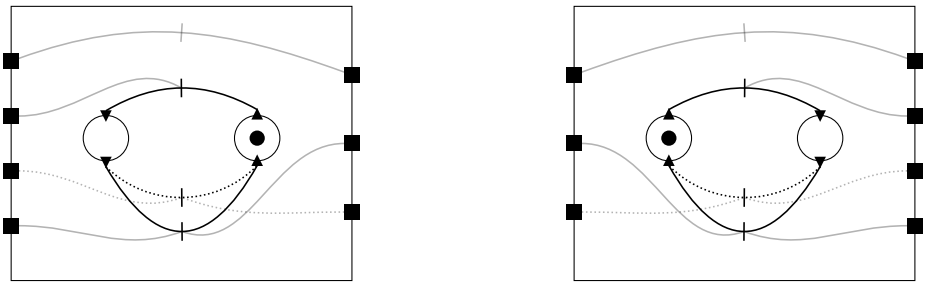


Figure 4.24: LOCK : $(3, 3)$



(a) RLOCKINTERFACE : $(4, 3)$ (b) FLOCKINTERFACE : $(3, 4)$

Figure 4.25: Lock interface components

1. Unlocked (U)
2. Locked by the car behind (B)
3. Locked by the car in front (F)

with partially-specified transitions that allow the cars in front/behind to lock/unlock and check if they can lock the lock. Such a component is illustrated in Fig. 4.24.

The desired system behaviour requires that cars can request the lock, which can then fail or succeed. In order to separate this behaviour into two steps, we require suitable "impedance matcher" interface components between each CAR and LOCKs. Such interfaces have local states to distinguish between request made/request response, passing signals to their boundaries as appropriate, as illustrated in Fig. 4.25.

Given these components, we can define the OVERTAKE(k) net system:

$$\begin{aligned}\text{OVERTAKE}(k) \stackrel{\text{def}}{=}\ & \textbf{bind } \mathit{interfacedLock} = \text{RLOCKINTERFACE} \textbf{ ; } \text{LOCK} \textbf{ ; } \text{FLOCKINTERFACE} \textbf{ in} \\ & \textbf{bind } \mathit{lockCar} = \mathit{interfacedLock} \textbf{ ; } \text{CAR} \textbf{ in} \\ & \uparrow_4 \textbf{ ; n\_sequence } k\ \mathit{lockCar} \textbf{ ; } \mathit{interfacedLock} \textbf{ ; } \downarrow_4\end{aligned}$$

Due to the complexity of the components, we do not draw the composite OVERTAKE(k) system. The schematic for OVERTAKE(2) is shown in Fig. 4.26, where, for simplicity, we have collapsed the three constituents of $\mathit{interfacedLock}$ into a single component, iLock.

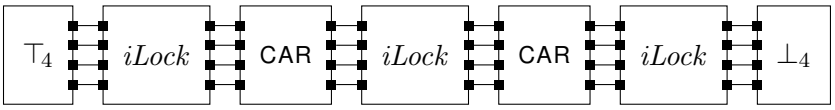
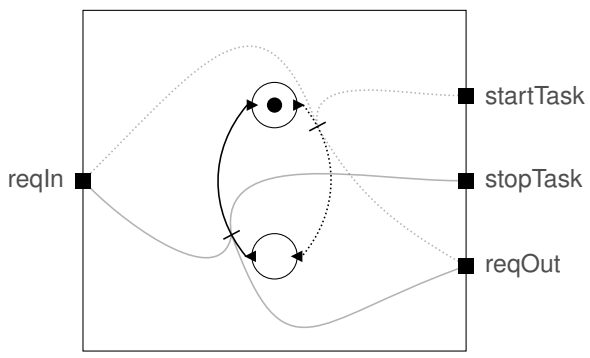


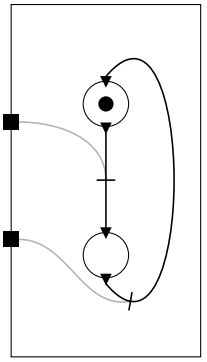
Figure 4.26: Schematic of OVERTAKE(2)

4.2.2 Hartstone

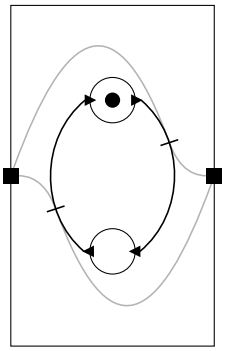
The HARTSTONE(k) system models a program that starts k tasks in some order, lets them compute, before instructing them to stop them in the same order. In the original description of the problem, a central controller is directly connected to the k tasks. Using PNBs, we can simplify this description: we construct k "single task" controllers, each responsible for a single task, with each controller passing signals to the next controller.



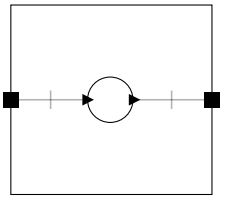
(a) CONTROLLER : $(1, 3)$



(b) TASK: $(2, 0)$



(c) MASTER : $(1, 1)$



(d) OPB : $(1, 1)$

Figure 4.27: HARTSTONE(k) component nets

The component in Fig. 4.27d is a *one-place buffer* (OPB), which allows a token to asynchronously flow through a series of connected components. Without the OPBs, the fully specified transitions would connect the "ready" place in each CONTROLLER component together, similarly for the "working" place; the tasks would therefore be simultaneously stopped, which is not what we wish to model.

The PNBml expression to represent a HARTSTONE(k) system, with k tasks is as follows:

$$\begin{aligned}\text{HARTSTONE}(k) \stackrel{\text{def}}{=}\ & \textbf{bind}\ asyncTask = \text{OPB}\ \textbf{;}\ \text{CONTROLLER}\ \textbf{;}\ (\text{TASK} \otimes \text{ID})\ \textbf{in} \\ & \textbf{bind}\ tasks = \textbf{n\_sequence}\ k\ asyncTask\ \textbf{in} \\ & \text{ETA}\ \textbf{;}\ (\text{MASTER}\ \textbf{;}\ tasks) \otimes \text{ID}\ \textbf{;}\ \text{EPSILON}\end{aligned}$$

This expression represents a sequence of controller/tasks, which are wired to a master controller (that models the protocol of repeatedly starting all processes and then stopping them). Signals are looped back around (via ETA/EPSILON), such that MASTER receives the signal when all controllers have already received it. The schematic of HARTSTONE(2) is shown in Fig. 4.28.

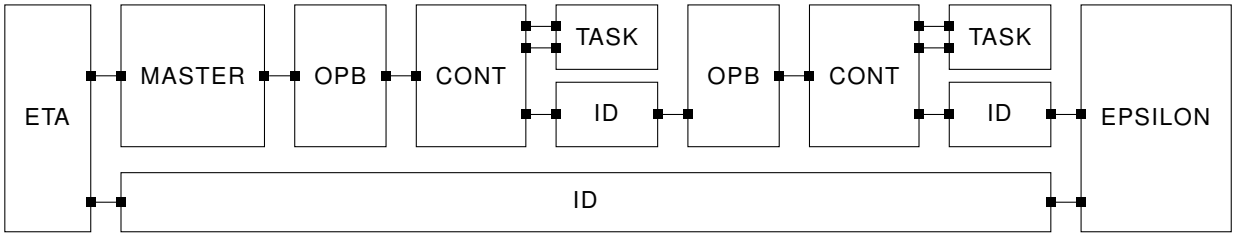
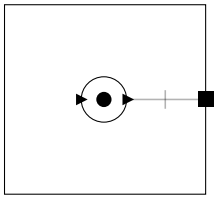


Figure 4.28: Schematic of HARTSTONE(2)

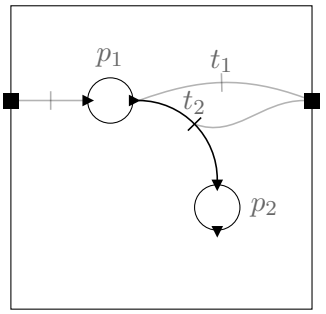
4.2.3 Iterated Choice

The ITER-CHOICE(k) system models a simple sequence of components that each have a choice between two transitions to fire; the choice they make is recorded in the resulting marking. Such a system has a simple component-wise specification as a sequence of suitable "choice" components.

A single choice component is illustrated in Fig. 4.29b; the choice of transition is between those consuming tokens from p_1: either t_1 or t_2. When t_2 is chosen, the choice is recorded by a token being present in p_2. A single token is injected into a sequence of single choice components by the ADDTOK component, as shown in Fig. 4.29a.



(a) ADDTOK: $(0,\ 1)$



(b) CHOICE: $(1,\ 1)$

The component-wise specification of a sequence of choice components is then:

$$\text{ITER-CHOICE}(k) \stackrel{\text{def}}{=} \text{ADDTOK} \;;\; \textbf{n\_sequence}\ k\ \text{CHOICE} \;;\; \bot$$

the schematic of such a system is almost trivial, e.g. the schematic of ITER-CHOICE(2) is illustrated in Fig. 4.30, with the corresponding composite net shown in Fig. 4.31.

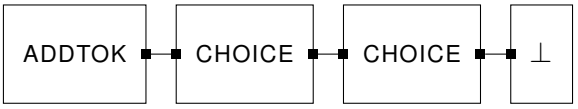


Figure 4.30: Schematic of ITER-CHOICE(2)

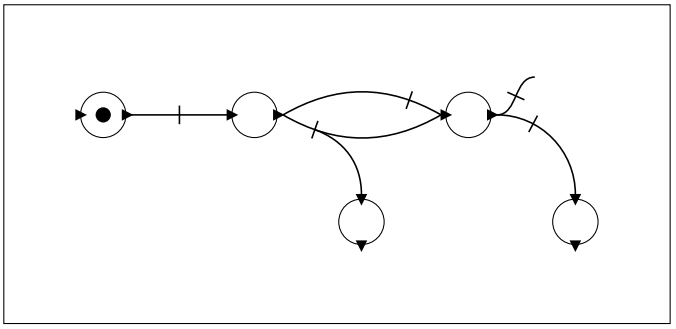
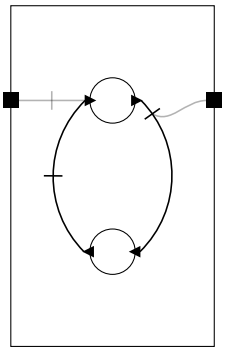


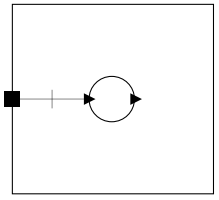
Figure 4.31: ITER-CHOICE(2)

4.2.4 Replicator

A replicator is a simple component that is able to fire the transition connected to its right boundary port an unbounded number of times, once the transition connected to its left boundary port has fired once. Sequences of such replicators can be formed into a chain, similarly to the ITER-CHOICE(k) system.



(a) REPLICATOR: $(1, 1)$



(b) TAKETOK: $(1, 0)$

The REPLICATORS(k) system is defined:

$$\text{REPLICATORS}(k) \stackrel{\text{def}}{=} \text{ADDTOK} \;;\; \textbf{n\_sequence}\ k\ \text{REPLICATOR} \;;\; \text{TAKETOK}$$

the schematic of such a system is again almost trivial, e.g. the schematic for REPLICATORS(3) is illustrated in Fig. 4.33, with the corresponding composite net shown in Fig. 4.34.

Figure 4.33: Schematic of REPLICATORS(3)

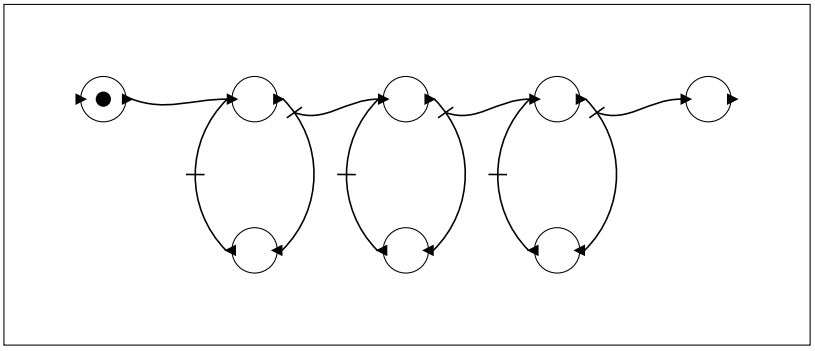


Figure 4.34: REPLICATORS(3)

4.2.5 DAC: Divide and Conquer

The DAC(k) system [145] models the recursion in divide and conquer approaches to problem solving. DAC(k) is comprised of k WORKER components; each worker can chose to invoke a computation in a child process, or perform all computation itself. If a worker invokes a child process, it must then wait for it and all of its descendants to finish. Each layer in the recursion is modelled by the addition of a worker net. Varying the number of worker nets allows one to treat recursion up to any depth. The worker chain is terminated by a net without synchronising transitions, forcing the last worker to do any remaining work itself.

The components of DAC(k) are illustrated in Fig. 4.35; the CONTROLLER component starts the chain of workers, before "joining" (waiting for) them.

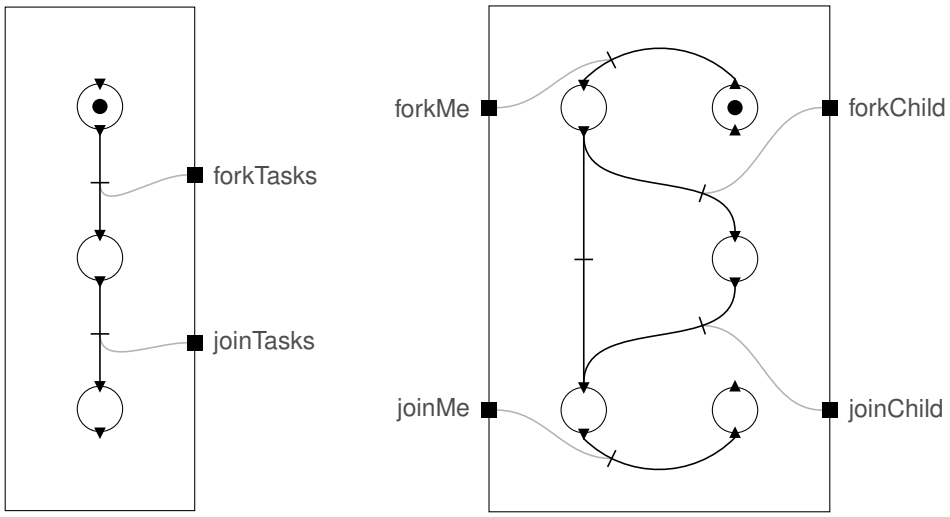
The specification of DAC(k) with k workers is:

$$\text{DAC}(k) \stackrel{\text{def}}{=} \text{CONTROLLER} \mathbin{;} \textbf{n\_sequence}\ k\ \text{WORKER} \mathbin{;} \downarrow_2$$

The schematic of DAC(3) is shown in Fig. 4.36.

4.2.6 Dining Philosophers

The DPH(k) system is the classic example of a concurrent system modelled by Petri nets. A natural compositional specification is to describe components for individual



(a) CONTROLLER : $(0, 2)$ (b) WORKER : $(2, 2)$

Figure 4.35: DAC(k) component nets

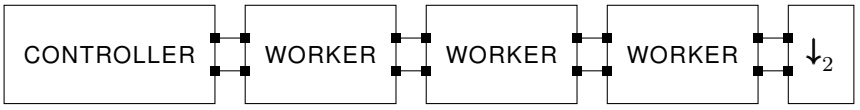
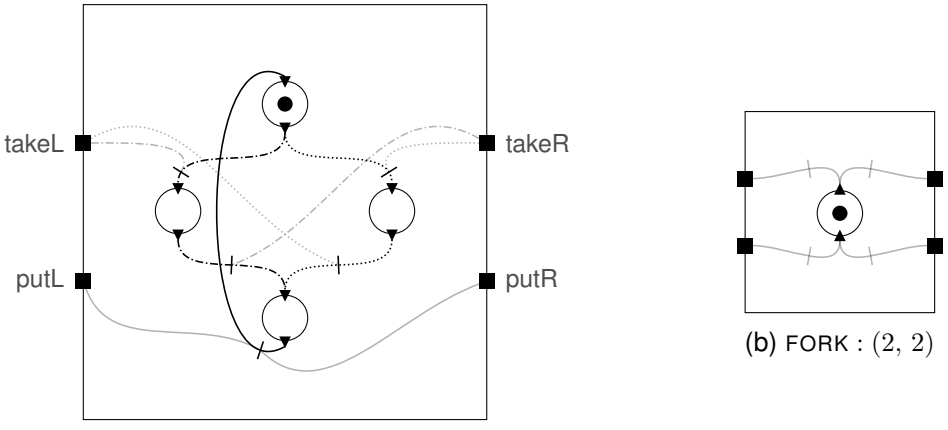


Figure 4.36: Schematic of DAC(3)

philosophers and forks, which are connected in a loop. The PHILO component, shown in Fig. 4.37a, attempts to take the forks on its left and right (in either order), before eating and replacing the forks. A FORK component, illustrated in Fig. 4.37b, can be taken and replaced from either its left or right.



(a) PHILO : $(2, 2)$ (b) FORK : $(2, 2)$

Figure 4.37: Dining Philosophers component nets

To form a “dining table” of philosophers, a sequence of alternating philosopher/forks is created, before connecting the first philosopher to the last fork to close the ‘loop” of the

table. The following expression embodies this specification:

$$\mathsf{DPH}(k) \stackrel{\text{def}}{=} \textbf{bind } philoFork = \mathsf{PHILO} \textbf{ ; } \mathsf{FORK} \textbf{ in}$$
$$\mathsf{ETA}_2 \textbf{ ; } (\textbf{n\_sequence } k\ philoFork \otimes \mathsf{ID}_2) \textbf{ ; } \mathsf{EPSILON}_2$$

The schematic for $\mathsf{DPH}(2)$ is shown in Fig. 4.38.

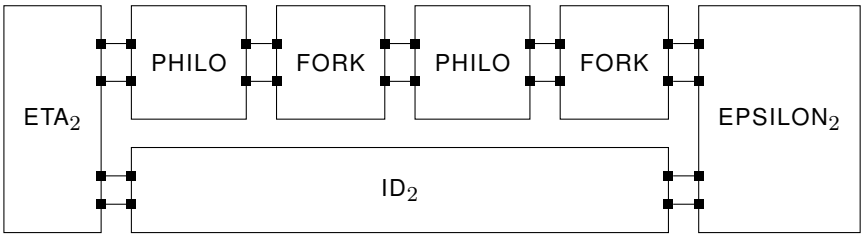


Figure 4.38: Schematic of $\mathsf{DPH}(2)$

4.2.7 Milner's Cyclic Scheduler

Milner's CYCLIC(k) system models a set of k processes arranged in a cycle; each process starts in turn and notifies the next process in the cycle to start. Upon finishing, the processes may start again, but only if they have been signalled to do so by the previous process; similarly, if a process is signalled whilst still running, it must wait to finish before starting and passing the signal along.

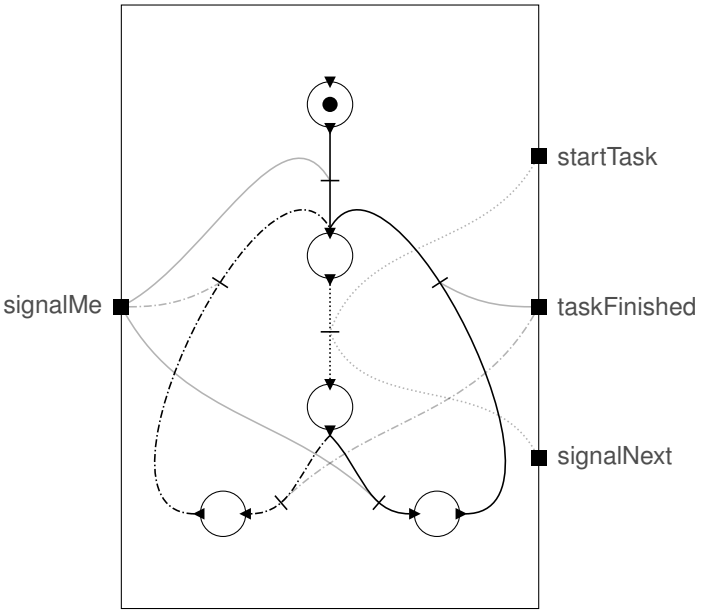


Figure 4.39: SCHEDULER : $(2, 2)$

The component-wise specification of CYCLIC(k) makes use of the same task component net as HARTSTONE(k) (illustrated in Fig. 4.27b). Each SCHEDULER component, illustrated in Fig. 4.39 has a corresponding TASK component that it controls. Each

scheduler/task component feeds its signalNext into the next component; the cycle is initially started by the INJECTOR component from TOKENRING(k) (shown in Fig. 4.8a). The specification is then:

$$\begin{aligned}\text{CYCLIC}(k) \stackrel{\text{def}}{=}\ & \textbf{bind } schedTask = \text{SCHEDULER ; TASK} \otimes \text{ID } \textbf{in} \\ & \textbf{bind } tasks = \text{INJECTOR ; } \textbf{n\_sequence } k\ schedTask \textbf{ in} \\ & \text{ETA ; } (tasks \otimes \text{ID}) \text{ ; EPSILON}\end{aligned}$$

The schematic of CYCLIC(2) is shown in Fig. 4.40.

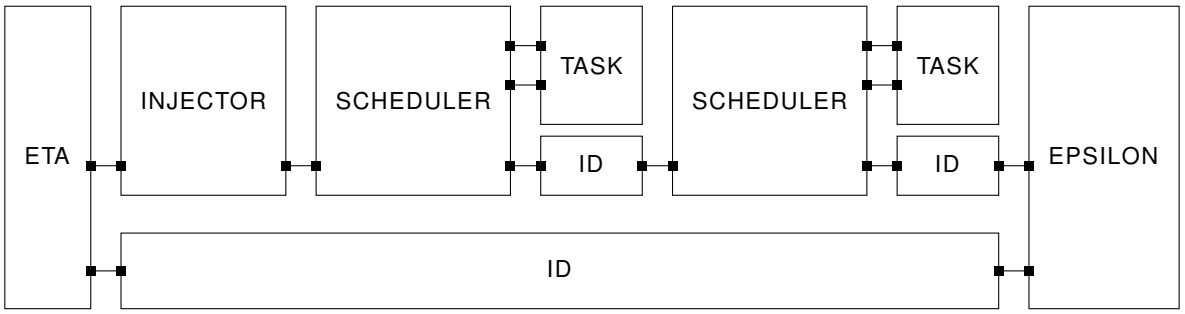


Figure 4.40: Schematic of CYCLIC(2)

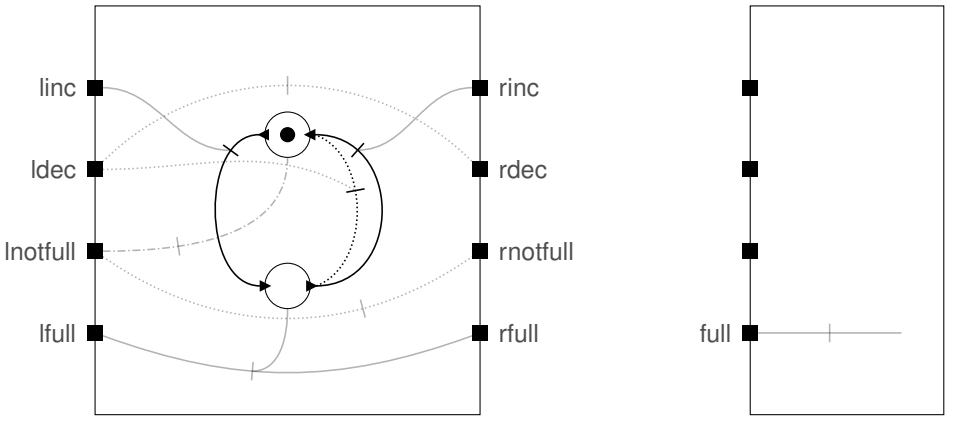
4.2.8 k-bit Counter

The COUNTER(k) system models a "counter" system that counts from $0 - n$ allowing increment/decrement. Intuitively, a k-bit counter is formed by synchronously composing k of the single-bit counter components from Fig. 4.41a, and terminating with a ZEROBIT component, shown in Fig. 4.41b, which always reports as being full and can't be incremented or decremented. The intuitive description of a 1-bit component is that it is either "empty" or "full". A full component may be directly decremented, or may pass its token to the next component, in either case it becomes empty. Passing tokens along a chain of components allows the chain to become full — a k-bit counter is not full if any component has a token in the empty place; it is full if all places have tokens in the full place.

The specification of COUNTER(k) is simple:

$$\text{COUNTER}(k) \stackrel{\text{def}}{=} \text{TESTER ; } \textbf{n\_sequence } k \text{ ONEBIT ; ZEROBIT}$$

Clients of COUNTER(k) interact with the 4 boundary ports on its left boundary: they may increment, decrement and test for not-full/full (if the counter is full to capacity, transitions connected to $lfull$ may fire and otherwise not). Indeed, while the counter sequence "reports" that it is not full, the tester component repeatedly fires the sequence's



(a) ONEBIT : $(4, 4)$ (b) ZEROBIT : $(4, 0)$

Figure 4.41: k-bit Counter component nets

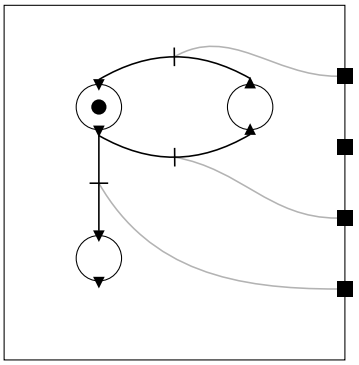


Figure 4.42: TESTER component net

inc transition and then tests that the sequence "reports" as being full. The schematic for COUNTER(3) is shown in Fig. 4.43.

Figure 4.43: Schematic of COUNTER(3)

4.3 Specification Domain Specific Language

In this section we give a formal description of the language PNBml for specifying net compositions. In particular, we show how PNBml expressions are evaluated to nets, and how the type system ensures that only valid expressions are processed, via the slogan *well-typed programs are well-composed* (Thm. 4.2), in the style of Milner [150].

PNBml is a monomorphic, call-by-value functional language; it follows the tradition in classic functional programming languages, such as ML, of extending the syntax of the lambda calculus with convenient programming constructs. In particular, we add natural

numbers, net literals/composition, syntax for variable binding and recursion over natural numbers.

The abstract syntax of PNBml is given in Fig. 4.44: x is drawn from a countable set of variables, n is a net literal (defined using a low-level syntax that we describe in §4.3.3). Natural number literals are written as standard: $0, 1, \ldots$, but semantically are in Peano form: p is either 0 or $S(p')$ for some natural literal p'. Function application is indicated by simple juxtaposition: $e_1\ e_2$. Following mathematical convention, ';' and '$\otimes$' associate to the left, with '$\otimes$' binding tighter than ';'.

$$
\begin{array}{lr}
e = x & \text{(variable)} \\
\mid p & \text{(natural number literal)} \\
\mid n & \text{(net literal)} \\
\mid \textbf{bind}\ x = e_1\ \textbf{in}\ e_2 & \text{(variable binding)} \\
\mid \boldsymbol{\lambda} x : \tau\ \textbf{.}\ e & \text{(function abstraction)} \\
\mid e_1\ e_2 & \text{(function application)} \\
\mid e_1\ \textbf{;}\ e_2 & \text{(sequential net composition)} \\
\mid e_1 \otimes e_2 & \text{(tensor net composition)} \\
\mid \textbf{fold}_{\mathbb{N}}\ e_n\ e_z\ e_s & \text{(recursion over natural numbers)}
\end{array}
$$

Figure 4.44: Syntax of PNBml

Note that the sequence construction, **n_sequence** $e_n\ e_1$ for an expression e_n, discussed in §4.1.3, does not appear in the formal syntax, since it is a syntactic sugar for a fold:

$$\textbf{n\_sequence}\ e_n\ e_1 \stackrel{\text{def}}{=} \textbf{fold}_{\mathbb{N}}\ (e_n - 1)\ e_1\ \left(\boldsymbol{\lambda} x : \mathrm{Net}\langle k,\, k\rangle\ \textbf{.}\ e_1\ \textbf{;}\ x\right)$$

where $S(p) - 1 \stackrel{\text{def}}{=} p$. Notice however, that **n_sequence** $0\ e_1$ cannot appear in any expression: $0 - 1$ is undefined[3]. We discuss the importance of the type annotation, $\mathrm{Net}\langle k,\, k\rangle$ on the lambda expression in §4.3.2, but briefly, for the sequence to be correct, e_1 must be an expression representing a net with k boundary ports on each side.

4.3.1 Operational Semantics

We define the big-step operational semantics of PNBml in Fig. 4.46, using an explicit variable binding environment E, which we discuss shortly. The language is call-by-value, and the evaluation rules reduce PNBml expressions to values, which have three forms: (i) (Composite) nets, (ii) Natural numbers, and (iii) Function closures (consisting

[3] An alternative, would be to consider n^0 as ID_k, assuming $n : (k,\, k)$

of an environment and lambda abstraction). A variable binding environment, E, is simply a map from variables, x, to values, v, as shown in Fig. 4.45.

$$
\begin{array}{lr}
v = n & \text{(Net Literal)} \\
\quad \mid p & \text{(Natural Number Literal)} \\
\quad \mid \langle E,\, \boldsymbol{\lambda} x : \tau \,.\, e \rangle & \text{(Function Abstraction)} \\
\\
E = \bullet & \text{(Empty Environment)} \\
\quad \mid E, x \mapsto v & \text{(Environment Extension)}
\end{array}
$$

Figure 4.45: PNBml Values and Operational Environments

The evaluation rules are of the form: $E \vdash e \Downarrow v$, meaning that in environment E, expression e reduces to value v. Evaluation intuitively proceeds as follows: variables are simply looked up in the environment, lambdas evaluate to a closure over the current environment, and numeric/net literals evaluate to themselves. Bindings evaluate their body in an extended environment, similarly for applications. Synchronous and tensor composition evaluate both expressions to net literals, before applying the appropriate net operation. Structural recursion over the natural numbers (commonly referred to as a *fold*, e.g. see [151]) is implemented via repeated function application of e_s to e_z.

For example, considering the PNBml expression given at the end of §4.1.3; after expanding syntactic sugar, the following example reduction holds.

Example 4.1.
For expression

$$
\begin{aligned}
e = \; & \textbf{bind}\ \mathit{makeRing} = \boldsymbol{\lambda} x : \mathrm{Net}\langle 1,\, 1\rangle \,.\, \textsc{eta} \,;\, (x \otimes \textsc{id}) \,;\, \textsc{epsilon}\ \textbf{in} \\
& \textbf{bind}\ \mathit{procs} = \textbf{fold}_{\mathbb{N}}\ 2\ \textsc{worker}\ \left(\boldsymbol{\lambda} x : \mathrm{Net}\langle 1,\, 1\rangle \,.\, \textsc{worker} \,;\, x\right)\ \textbf{in} \\
& \mathit{makeRing}\ (\textsc{injector} \,;\, \mathit{procs})
\end{aligned}
$$

and value

$$
v = \textsc{eta} \,;\, \Big(\textsc{injector} \,;\, \big(\textsc{worker} \,;\, (\textsc{worker} \,;\, \textsc{worker})\big)\Big) \otimes \textsc{id} \,;\, \textsc{epsilon}
$$

we have that

$$
\varnothing \vdash e \Downarrow v
$$

Proof. Direct, by the definition of the evaluation rules. □

4.3.2 Static Type Checking

As mentioned in §4.1.3, *run-time* synchronous composition errors are encountered when attempting to synchronously compose nets with differently-sized boundaries. Such

$$(\textsc{EVar})\ \frac{}{E \vdash x \Downarrow E(x)} \qquad (\textsc{ELam})\ \frac{}{E \vdash \boldsymbol{\lambda} x : \tau \,.\, e \Downarrow \langle E,\, \boldsymbol{\lambda} x : \tau \,.\, e \rangle}$$

$$(\textsc{EBind})\ \frac{E \vdash e_1 \Downarrow v_1 \qquad E, x \mapsto v_1 \vdash e_2 \Downarrow v_2}{E \vdash \textbf{bind}\ x = e_1\ \textbf{in}\ e_2 \Downarrow v_2}$$

$$(\textsc{EApp})\ \frac{E \vdash e_1 \Downarrow \left\langle E',\, \boldsymbol{\lambda} x : \tau \,.\, e_3 \right\rangle \qquad E \vdash e_2 \Downarrow v_2 \qquad E', x \mapsto v_2 \vdash e_3 \Downarrow v_3}{E \vdash e_1\, e_2 \Downarrow v_3}$$

$$(\textsc{ENat})\ \frac{}{E \vdash p \Downarrow p} \qquad (\textsc{ENet})\ \frac{}{E \vdash n \Downarrow n}$$

$$(\textsc{ESeq})\ \frac{E \vdash e_1 \Downarrow n_1 \qquad E \vdash e_2 \Downarrow n_2}{E \vdash e_1 \,;\, e_2 \Downarrow n_1 \,;\, n_2}$$

$$(\textsc{ETen})\ \frac{E \vdash e_1 \Downarrow n_1 \qquad E \vdash e_2 \Downarrow n_2}{E \vdash e_1 \otimes e_2 \Downarrow n_1 \otimes n_2}$$

$$(\textsc{EFoldZ})\ \frac{E \vdash e_n \Downarrow 0 \qquad E \vdash e_z \Downarrow v}{E \vdash \textbf{fold}_{\mathbb{N}}\ e_n\ e_z\ e_s \Downarrow v}$$

$$(\textsc{EFoldS})\ \frac{E \vdash e_n \Downarrow S(p) \qquad E \vdash e_s\ (\textbf{fold}_{\mathbb{N}}\ p\ e_z\ e_s) \Downarrow v}{E \vdash \textbf{fold}_{\mathbb{N}}\ e_n\ e_z\ e_s \Downarrow v}$$

Figure 4.46: Operational Semantics of PNBml

errors can be ruled out *statically*, that is, before evaluation is performed, by using a *type system*. Such a type system ascribes a *type* to each expression, with a type being a static approximation of the dynamic (after evaluation, at run-time) form of the expression [152]. Indeed, other run-time errors can be ruled out by using a static type system; such errors include treating a net as a function or trying to tensor two lambda expressions.

In this subsection, we introduce a simple, *monomorphic* type system, giving rules for ascribing types to expressions, before proving a safety property: if a expression can be typed, it cannot fail to evaluate to a value with the same type.

4.3.2.1 Monomorphic Type System

A monomorphic type system ascribes a *single* type to any particular expression. Our monomorphic type system uses only two types, as shown in Fig. 4.47. As we will prove, these types can be used to rule out *all* possible run-time failures for PNBml expressions. The novel feature is that component boundary sizes are tracked, achieved by

parameterising the net base type: $\mathrm{Net}\langle l,\, r\rangle$ by $l, r \in \mathbb{N}$, which describe the left and right boundary sizes respectively. Indeed, the net component type, $\mathrm{Net}\langle k,\, l\rangle$, will be ascribed to any expression, that, when evaluated, results in a (composite) net with boundaries k and l, for a particular choice of k and l. Thus the type $\mathrm{Net}\langle k,\, l\rangle$ approximates the form of the resulting expression in that it only records the boundary sizes.

The deductive rules for assigning types to PNBml expressions are shown in Fig. 4.48. As is standard (e.g. [153]) for a monomorphic language, Γ is a sequence of bindings from variables to types (shown in Fig. 4.47), which is extended in the TBIND and TLAM rules and inspected in the TVAR rule.

It is important to note is that lambda expressions parameters are annotated with their type. This is necessary to *infer* the type of arbitrary expressions; inference involves reading type rules "in reverse", from conclusion to premise, and regarding Γ and e as *inputs*, and τ as an output. Indeed, in TLAM, we require a single *monomorphic* type for x, with which to extend the type environment; without the annotation, we have no such type. For such a reverse reading to be correct, the rules must be *syntax-directed*: each syntactic form of the expression language corresponds to exactly one type rule.

$$
\begin{array}{lr}
\tau = \mathbb{N} & (\text{Natural Number}) \\
\quad \mid \mathrm{Net}\langle k,\, l\rangle \;\; (k, l \in \mathbb{N}) & (\text{Net Component}) \\
\quad \mid \tau_1 \to \tau_2 & (\text{Function Type}) \\
\\
\Gamma = \bullet & (\text{Empty Environment}) \\
\quad \mid \Gamma, x \mapsto \tau & (\text{Environment Extension})
\end{array}
$$

Figure 4.47: Monomorphic Types and Type Environments of PNBml

We say an expression e is *well-typed*, if there exists a type environment binding all free variables appearing in e, Γ, and a type, τ, such that $\Gamma \vdash e : \tau$.

As an example use of the typing rules, consider composing the lambda expression that forms a loop from a sequence of workers, introduced in §4.1.3, reproduced here:

$$\boldsymbol{\lambda} x : \mathrm{Net}\langle 1,\, 1\rangle \;.\; \textsc{eta} \;;\; (x \otimes \textsc{id}) \;;\; \textsc{epsilon}$$

We can confirm that this expression is well-composed (i.e. that it contains no invalid compositions, a consequence of being well-typed) with the proof illustrated in Fig. 4.49 (using η, i, ϵ in place of ETA, ID, EPSILON).

Values can be ascribed types; we write $\vDash v : \tau$ if value v has type τ. Furthermore, an operational environment (mapping lambda-bound-variables to values), E, respects a type environment, Γ, iff the domains of Γ and E are equal and E point-wise respects Γ, as per Fig. 4.50.

$$\text{(TVAR)}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \text{(TBIND)}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x \mapsto \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \textbf{bind}\ x = e_1\ \textbf{in}\ e_2 : \tau_2}$$

$$\text{(TLAM)}\ \frac{\Gamma, x \mapsto \tau_1 \vdash e : \tau_2}{\Gamma \vdash \boldsymbol{\lambda} x : \tau_1\ .\ e : \tau_1 \to \tau_2} \qquad \text{(TAPP)}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}$$

$$\text{(TNAT)}\ \frac{}{\Gamma \vdash p : \mathbb{N}} \qquad \text{(TNET)}\ \frac{n : (k,\, l)}{\Gamma \vdash n : \mathrm{Net}\langle k,\, l\rangle}$$

$$\text{(TTEN)}\ \frac{\Gamma \vdash e_1 : \mathrm{Net}\langle k,\, l\rangle \qquad \Gamma \vdash e_2 : \mathrm{Net}\langle m,\, n\rangle}{\Gamma \vdash e_1 \otimes e_2 : \mathrm{Net}\langle k+m,\, l+n\rangle}$$

$$\text{(TSEQ)}\ \frac{\Gamma \vdash e_1 : \mathrm{Net}\langle k,\, l\rangle \qquad \Gamma \vdash e_2 : \mathrm{Net}\langle m,\, n\rangle \qquad l = m}{\Gamma \vdash e_1\ ;\ e_2 : \mathrm{Net}\langle k,\, n\rangle}$$

$$\text{(TFOLD)}\ \frac{\Gamma \vdash e_n : \mathbb{N} \qquad \Gamma \vdash e_z : \tau \qquad \Gamma \vdash e_s : \tau \to \tau}{\Gamma \vdash \textbf{fold}_{\mathbb{N}}\ e_n\ e_z\ e_s : \tau}$$

Figure 4.48: Syntax-directed Monomorphic Typing Rules for PNBml

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma(x) = \mathrm{Net}\langle 1,\, 1\rangle}{\Gamma \vdash x : \mathrm{Net}\langle 1,\, 1\rangle} \qquad \dfrac{i : (1,\, 1)}{\Gamma \vdash i : \mathrm{Net}\langle 1,\, 1\rangle}}{\Gamma \vdash x \otimes i : \mathrm{Net}\langle 2,\, 2\rangle} \qquad \dfrac{\epsilon : (2,\, 0)}{\Gamma \vdash \epsilon : \mathrm{Net}\langle 2,\, 0\rangle}}{\Gamma \vdash (x \otimes i)\ ;\ \epsilon : \mathrm{Net}\langle 2,\, 0\rangle} \qquad \dfrac{\eta : (0,\, 2)}{\Gamma \vdash \eta : \mathrm{Net}\langle 0,\, 2\rangle}}{\bullet, x \mapsto \mathrm{Net}\langle 1,\, 1\rangle \vdash \eta\ ;\ (x \otimes i)\ ;\ \epsilon : \mathrm{Net}\langle 0,\, 0\rangle}}{\bullet \vdash \boldsymbol{\lambda} x : \mathrm{Net}\langle 1,\, 1\rangle\ .\ \eta\ ;\ (x \otimes i)\ ;\ \epsilon : \mathrm{Net}\langle 1,\, 1\rangle \to \mathrm{Net}\langle 0,\, 0\rangle}$$

For compact presentation, let $\Gamma \stackrel{\text{def}}{=} \bullet, x \mapsto \mathrm{Net}\langle 1,\, 1\rangle$.

Figure 4.49: Example typing proof

Now we can state our formal notion of well-typed expressions can always be evaluated to well-composed values:

Theorem 4.2.
For a well-typed expression, $\Gamma \vdash e : \tau$, and operational environment E such that $\vDash E : \Gamma$, we have:

$$E \vdash e \Downarrow v \textit{ with } \vDash v : \tau$$

Proof. Induction over the structure of the proof of $\Gamma \vdash e : \tau$, uniquely determined by the structure of e. We only state the non-standard cases here:

- In the base cases of e being a net or natural number literal the result is trivial.

$$\frac{}{\vDash \bullet : \bullet} \qquad \frac{\vDash E : \Gamma \qquad \vDash v : \tau}{\vDash E, x \mapsto v : \Gamma, x \mapsto \tau}$$

$$\frac{}{\vDash p : \mathbb{N}} \qquad \frac{n : (k,\, l)}{\vDash n : \mathrm{Net}\langle k,\, l\rangle} \qquad \frac{\vDash E : \Gamma \qquad \Gamma, x \mapsto \tau_1 \vdash e : \tau_2}{\vDash \langle E,\, \boldsymbol{\lambda} x : \tau_1 \,.\, e\rangle : \tau_1 \to \tau_2}$$

Figure 4.50: Typing of Values and Operational Environments

- In the case of a tensor composition, $e_1 \otimes e_2$, we apply the I.H. to e_1 and e_2, obtaining values v_1 and v_2 such that $\vDash v_1 : \mathrm{Net}\langle k,\, l\rangle$ and $\vDash v_2 : \mathrm{Net}\langle m,\, n\rangle$ for some k, l, m and n. Now, by the inversion of the syntax-directed typing rules, v_1 and v_2 must be net literals, $n_1 : (k,\, l)$ and $n_2 : (m,\, n)$. Thus the premises for ETEN are satisfied and we have $e_1 \otimes e_2 \Downarrow n_1 \otimes n_2$, where $n_1 \otimes n_2 : (k + m,\, l + n)$, giving $\vDash n_1 \otimes n_2 : \mathrm{Net}\langle k + m,\, l + n\rangle$, as required.

- The case of synchronous composition is similar: we apply the I.H. to e_1 and e_2, obtaining values v_1 and v_2 such that $\vDash v_1 : \mathrm{Net}\langle k,\, l\rangle$ and $\vDash v_2 : \mathrm{Net}\langle m,\, n\rangle$ for some k, l, m and n. Observe that the conclusion of ESEQ is only defined when $l = m$, which is ensured by the final premise on TSEQ. Thus, by the definition of synchronous composition on net components, v is a net literal, $n_1 \,;\, n_2 : (k,\, n)$, giving $\vDash n_1 \,;\, n_2 : \mathrm{Net}\langle k,\, n\rangle$, as required.

- Finally, for a fold, $\textbf{fold}_{\mathbb{N}}\ e_n\ e_z\ e_s$, we apply the I.H. three times, obtaining values v_n, v_z and v_s such that $\vDash v_n : \mathbb{N}$, $\vDash v_z : \tau$ and $\vDash v_s : \tau \to \tau$.

  Now, we have to consider the the form of v_n, which, by the inversion of the typing rules, must be a natural number literal, either 0 or $S(p)$: in the former case, $E \vdash \textbf{fold}_{\mathbb{N}}\ 0\ e_z\ e_s \Downarrow v_z$, with $\vDash v_z : \tau$, as required. In the latter case, where v_n is $S(p)$, by our assumption that $\Gamma \vdash \textbf{fold}_{\mathbb{N}}\ e_n\ e_z\ e_s : \tau$, we have that $e_s : \tau \to \tau$, and furthermore, that $\Gamma \vdash \textbf{fold}_{\mathbb{N}}\ p\ e_z\ e_s : \tau$ (by substituting a leaf for p for the sub-tree corresponding to e_n). We therefore have the premises for TAPP, giving the conclusion that $\Gamma \vdash e_s\ (\textbf{fold}_{\mathbb{N}}\ p\ e_z\ e_s) : \tau$. Now, by the I.H., we have that $E \vdash e_s\ (\textbf{fold}_{\mathbb{N}}\ p\ e_z\ e_s) \Downarrow v$ and therefore $\textbf{fold}_{\mathbb{N}}\ S(p)\ e_z\ e_s \Downarrow v$, such that $\vDash v : \tau$, as required.

□

4.3.3 Net Literal Specification

To finish this chapter, for completeness, we briefly discuss how component nets are specified, such that they can be referred to within PNBml expressions. Later in this thesis, we will be concerned with checking reachability of net systems specified by

PNBml expressions, therefore our component specification language incorporates the initial/target markings of component places.

The grammar of the net specification language is as illustrated in Fig. 4.51.

```
pnbdef ::= "NET", name, "PLACES", places, "LBOUNDS", bounds,
           "RBOUNDS", bounds, "TRANS", transitions ;

name ::= ?sequence of alphanumeric characters?

places ::= "[", { place, "," }, place, "]"
         | "[", "]" ;

place ::= "<", name, ",", initialmarking, ",", targetmarking ">" ;

bounds ::= "[", { name, "," }, name, "]"
         | "[", "]" ;

transitions ::= "{", { transition, "," }, transition, "}"
              | "{", "}";

transition ::= "{", { port, "," }, port "}"
             | "{", "}" ;

port ::= ">", name
       | name, ">"
       | name, "?"
       | name ;

initialmarking ::= "0"
                 | "1" ;

targetmarking ::= "0"
                | "1"
                | "*" ;
```

Figure 4.51: BNF grammar for the input language

Each component definition specifies a list of places, a list of left and right boundary port names, and a set of transitions. Each place is specified as a triple of a (unique) name, initial marking and target marking, where the target marking can be one of 3 values: “yes” (1), “no” (0) or “don’t care” (*). N.B. since we allow “don’t care” target markings, we are able to specify *coverability* problems in addition to *reachability*. Each transition is specified as the set of ports that it connects, where each port is one of 5 different types:

1. Left boundary (denoted by a (left) boundary name),
2. Right boundary (denoted by a (right) boundary name),
3. Place input (**P**roduce) (denoted by a place name preceded by ’>’),

4. Place output (**C**onsume) (denoted by a place name followed by '>'),

5. Place **R**ead (denoted by a place name followed by '?').

As an example of the component specification format, we show the definition of the PHILO component, from §4.2.6.

```
NET philo
PLACES
    [ <none,  1,*>
    , <left,  0,*>
    , <right, 0,*>
    , <both,  0,1>
    ]
LBOUNDS [takeL, putL]
RBOUNDS [takeR, putR]
TRANS
    { { none>, takeL, >left }
    , { none>, takeR, >right }
    , { left>, takeR, >both }
    , { right>, takeL, >both }
    , { both>, putL, putR, >none }
    }
```

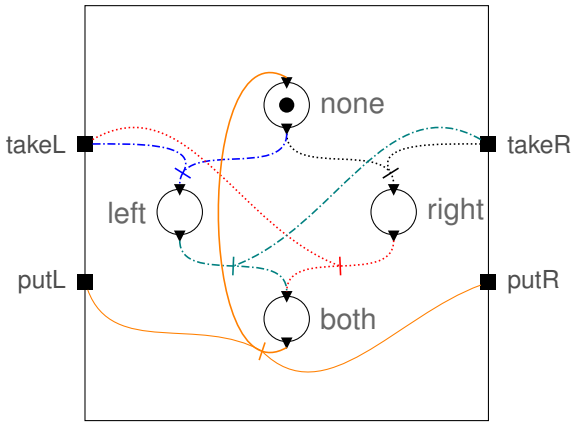


Figure 4.52: PHILO using component specification format

4.4 Summary

In this section we introduced the example systems we use to demonstrate and benchmark our reachability checking techniques. We illustrated that using a suitable DSL for net system specification gives convenient representations, especially for parametric systems, such as the example systems we consider. We illustrated our net system DSL, PNBml, and proved that by using a static type system, we can ensure that well-typed expressions are guaranteed to terminate and construct well-composed net systems.

Chapter 5

Compositional Statespace Generation

In this chapter we introduce a technique for *compositional* generation of the global statespace of systems modelled using the PNB specifications of Chapter 4, *without* first forming the corresponding composite global net. We show that because the boundary interactions of component nets are recorded in their statespaces, we are able to reconstruct the global statespace from the just the component statespaces.

Our goal is to use our method of compositional statespace generation (§5.1) as the basis of a compositional approach for checking reachability. However, the initial compositional approach we introduce here makes no attempt to limit the effects of statespace explosion. Existing approaches also consider the global statespace, but use techniques such as unfoldings or partial-order reduction to avoid generating the entire statespace. We will use compositionality to avoid the statespace explosion problem, but first show that we can indeed generate the global statespace in a compositional manner. While we prove our compositional technique correct (§5.3), evaluating its performance (§5.2) shows degradation due to statespace explosion. However, as we will demonstrate in Chapter 6, with suitable optimisations, our compositional technique is *particularly efficient* for several systems.

5.1 Reachability via Statespace Generation

We begin this section with a simple running example that we will use to illustrate our technique. Recall the simple k-bit buffer system, BUFFER(k), described in §4.1.1. We will consider checking this system for reachability of the marking with tokens in all the lower places, starting from the marking with tokens in all the upper places. The composite system, for $k = 3$, is illustrated with this initial/target marking, in Fig. 5.1.

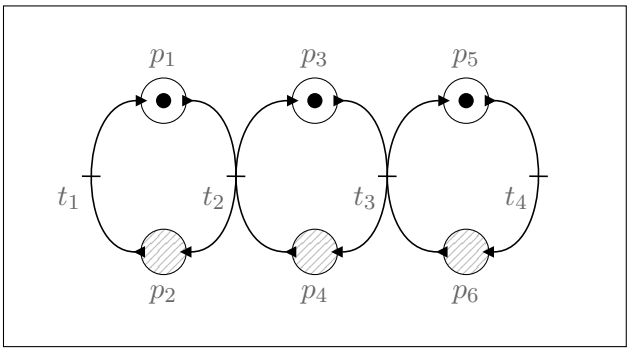


Figure 5.1: Reachability problem for marked PNB BUFFER(3)

Remark 5.1. An important point to note is that while the BUFFER(k) system is specified as the PNBml expression:

$$\text{BUFFER}(k) \stackrel{\text{def}}{=} \top \text{ ; } \textbf{n\_sequence } k \text{ BUFFER ; } \bot$$

for statespace generation, we need only consider the *evaluated* expression — a *PNB* expression. Indeed, since PNBml expressions are compact representations of PNB expressions, we can avoid a level of complexity (in particular, in the proof of correctness) by only considering the underlying PNB expressions. With this in mind, we assume that PNBml expressions have been evaluated as per §4.3.1 throughout this chapter.

5.1.1 Monolithic Statespace Generation

First, we describe the naive, *monolithic* approach: simply compose the PNB specification into a single global net, before generating its 2-NFA semantics; if we encounter an accepting state of the 2-NFA, the marking *is* reachable. This naive approach is embodied in Algorithm 5.1.

Algorithm 5.1 Naive algorithm to check reachability of PNB expression

1: Evaluate the PNB expression to obtain a single marked PNB,
2: Generate the 2-NFA semantics of the marked PNB (as per Defn. 2.43),
3: Check the resulting 2-NFA for emptiness; as per Remark 2.29, if the 2-NFA is empty then the target marking cannot be reached from the initial marking.

We outline running Algorithm 5.1 on BUFFER(3). The input is (up-to associativity) the PNB expression:

$$\top \text{ ; BUFFER ; BUFFER ; BUFFER ; } \bot \tag{5.1}$$

After performing step 1, we obtain the marked PNB shown in Fig. 5.1. Since this PNB has no boundaries, it can be considered as an ordinary Petri net, thus we can calculate the corresponding reachability NFA, illustrated in Fig. 5.2, where the states are labelled with (marked) net states and transitions are labelled with the fired net transitions.

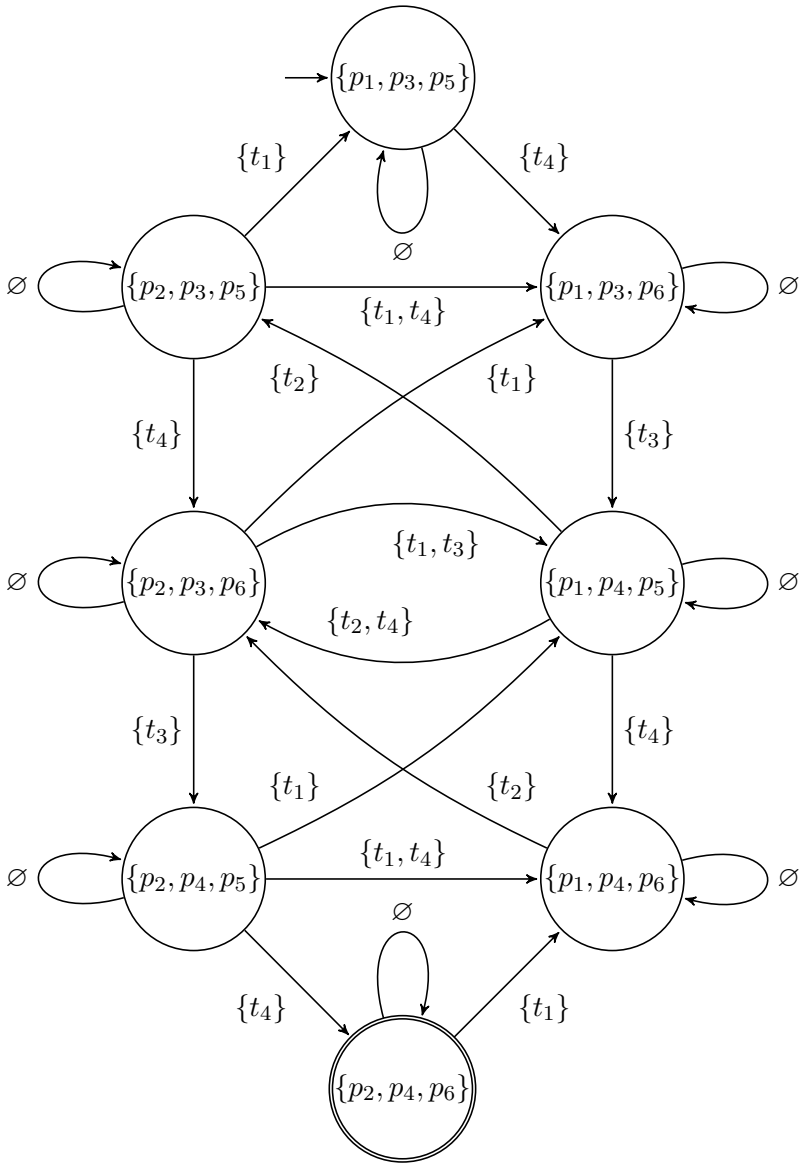


Figure 5.2: Reachability NFA for the marked PNB of Fig. 5.1 when considered as a Petri net

It is easy to confirm that this NFA has a non-empty language; an example word is $\langle \{t_4\}, \{t_3\}, \{t_4\}, \{t_2\}, \{t_3\}, \{t_4\} \rangle$. Simple inspection of the net in Fig. 5.1 confirms that firing the corresponding transitions, in sequence, does indeed transform the net from the initial marking to the target marking. If we do not consider the composite PNB as a Petri net, the generated 2-NFA is homomorphic to the NFA in Fig. 5.2 — the state component of the homomorphism is identity and the label component is the constant function that maps every set of transitions to '/'. The homomorphic 2-NFA is shown in Fig. 5.3, where we omit the unique label for neatness; indeed, for all PNBs with 0 boundaries, the corresponding 2-NFA has a unique, singleton set of labels.

Remark 5.2. The reader should note that we have not formally defined composition operations on marked PNBs, since it is not important for providing intuition. However,

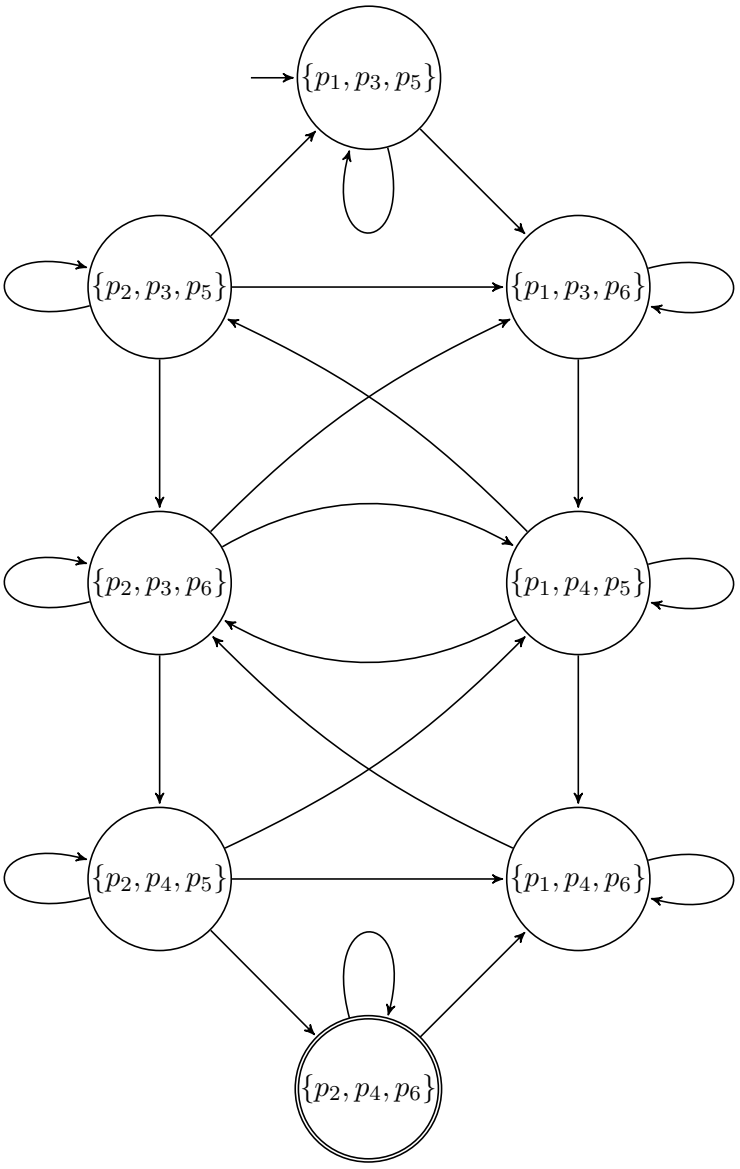


Figure 5.3: 2-NFA that is homomorphic to that in Fig. 5.2

we do define such operations in the proof of correctness, in §5.3.

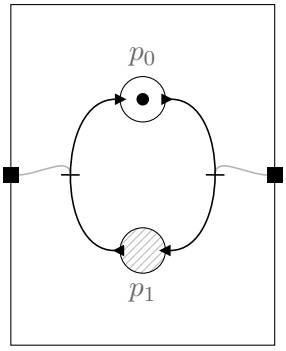
5.1.2 Compositional Statespace Generation

We now move on to discussing our alternative, *compositional*, approach, which will produce an isomorphic 2-NFA to that obtained by the naive algorithm. We compute the 2-NFA semantics of individual marked PNB components, obtaining *local* 2-NFA semantics, which are combined into a *global* 2-NFA semantics of the composite system. This global semantics can be checked for language emptiness, as before. This modified algorithm is outlined in Algorithm 5.2.

Algorithm 5.2 Compositional, *local* algorithm to check reachability of PNB expression

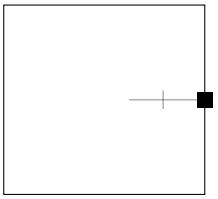
1: Convert each (marked) component PNB to its corresponding local 2-NFA,
2: Combine the 2-NFAs to obtain a global 2-NFA representing reachability of the composite net,
3: Check the resulting 2-NFA for emptiness.

We again illustrate our algorithm with 5.1, however, we now directly consider the marked PNB components, illustrated in Figs. 5.4, 5.5 and 5.6, where we also give the definition in terms of the component specification language of §4.3.3.



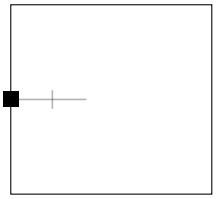

```
NET buffer
PLACES   [ <p0, 1, 0>
         , <p1, 0, 1>
         ]
LBOUNDS  [ left ]
RBOUNDS  [ right ]
TRANS    { {p0>, right, >p1}
         , {p1>, left,  >p0}
         }
```

Figure 5.4: BUFFER : $(1, 1)$



```
NET ⊤
PLACES   []
LBOUNDS  []
RBOUNDS  [ in ]
TRANS    { {in} }
```

Figure 5.5: $\top : (0, 1)$



```
NET ⊥
PLACES   []
LBOUNDS  [ in ]
RBOUNDS  []
TRANS    { {in} }
```

Figure 5.6: $\bot : (1, 0)$

By the left-associativity of ';', this expression is the binary tree with internal nodes being compositions and leaves PNB components illustrated in Fig. 5.7.

As per step 1 of Algorithm 5.2, we traverse the PNB expression tree, converting each component into its local reachability NFA. The translations for each component are illustrated in Fig. 5.8.

Traversing the expression tree of Fig. 5.7 and applying the conversions shown in Fig. 5.8 produces another expression tree. This new tree has identical shape and internal nodes, but 2-NFAs at the leaves instead of PNBs, as illustrated in Fig. 5.9. At this

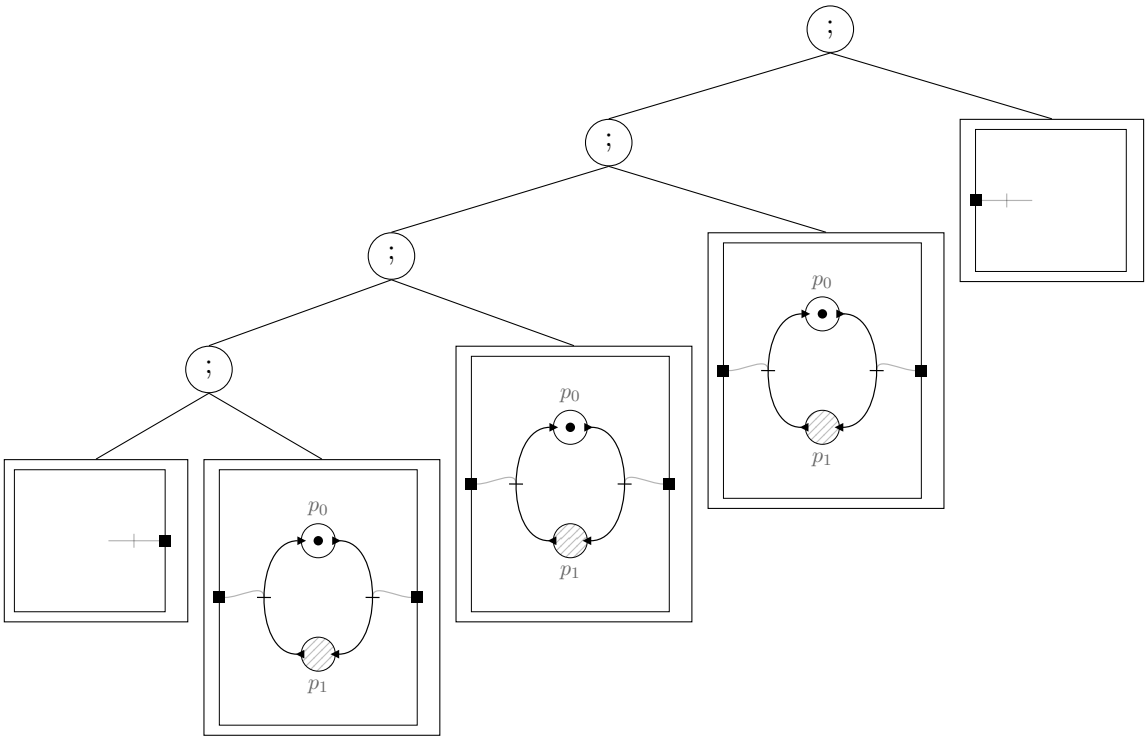


Figure 5.7: PNB BUFFER$(-)$ Expression tree

point, it is important to note that (up-to isomorphism) the *names* of a 2-NFA's states are unimportant, thus we elide them from now on when illustrating 2-NFAs.

For step 2 of Algorithm 5.2, we must collapse the tree of 2-NFAs into a single 2-NFA, representing reachability of the global marking in the composed net. To do so, we traverse the expression tree in a depth-first, left-to-right fashion performing compositions; since there are 4 internal composition nodes, we must perform 4 2-NFA compositions, generating the intermediate trees illustrated in Fig. 5.10, Fig. 5.11 and Fig. 5.12, before obtaining the final result (a tree with a single leaf node) illustrated in Fig. 5.13. In Fig. 5.13, for completeness, we have explicitly labelled the 2-NFA states using the

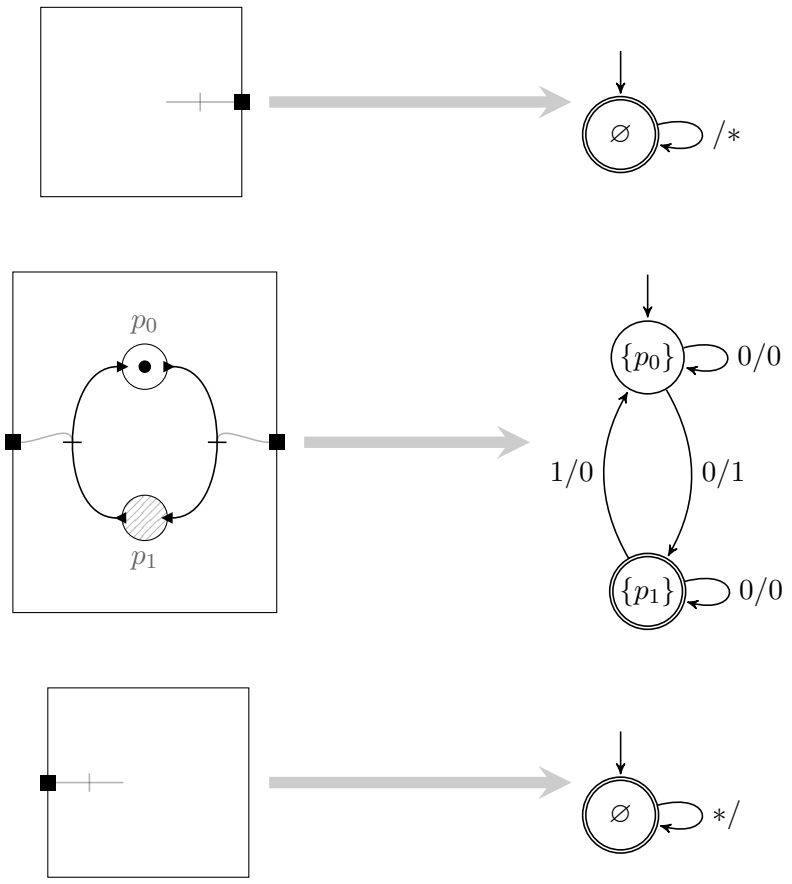


Figure 5.8: Translations of component marked PNBs to their 2-NFA semantics

following mapping to save space[1]:

$$
\begin{aligned}
0 &\mapsto \varnothing \,;\, \{p_0\} \,;\, \{p_0\} \,;\, \{p_0\} \,;\, \varnothing \\
1 &\mapsto \varnothing \,;\, \{p_0\} \,;\, \{p_0\} \,;\, \{p_1\} \,;\, \varnothing \\
2 &\mapsto \varnothing \,;\, \{p_0\} \,;\, \{p_1\} \,;\, \{p_0\} \,;\, \varnothing \\
3 &\mapsto \varnothing \,;\, \{p_0\} \,;\, \{p_1\} \,;\, \{p_1\} \,;\, \varnothing \\
4 &\mapsto \varnothing \,;\, \{p_1\} \,;\, \{p_0\} \,;\, \{p_0\} \,;\, \varnothing \\
5 &\mapsto \varnothing \,;\, \{p_1\} \,;\, \{p_0\} \,;\, \{p_1\} \,;\, \varnothing \\
6 &\mapsto \varnothing \,;\, \{p_1\} \,;\, \{p_1\} \,;\, \{p_0\} \,;\, \varnothing \\
7 &\mapsto \varnothing \,;\, \{p_1\} \,;\, \{p_1\} \,;\, \{p_1\} \,;\, \varnothing
\end{aligned}
$$

As an example, state 5 corresponds to the PNB marking:

$$\big\{\mathrm{inl}\,(\mathrm{inl}\,(\mathrm{inl}\,(\mathrm{inr}\,p_1))), \mathrm{inl}\,(\mathrm{inl}\,(\mathrm{inr}\,p_0)), \mathrm{inl}\,(\mathrm{inr}\,p_1)\big\}$$

[1] Recall that we write $x \,;\, y$ for the states (x, y) of a synchronous composition, with ';' being left associative.

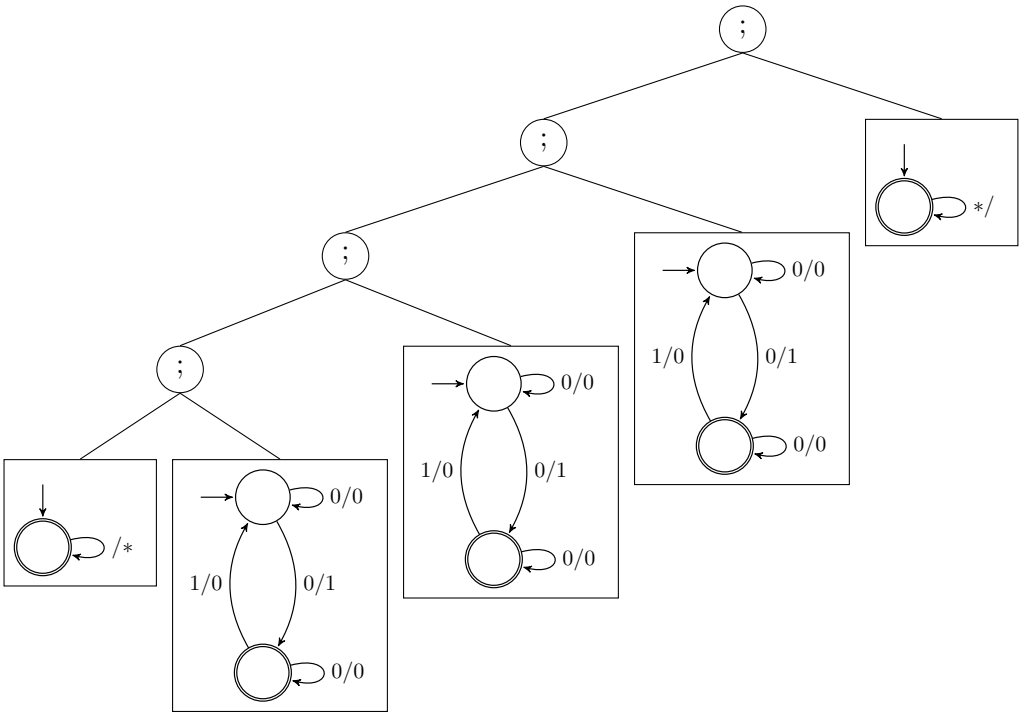


Figure 5.9: Expr. tree of Fig. 5.7, after converting marked PNBs to 2-NFAs

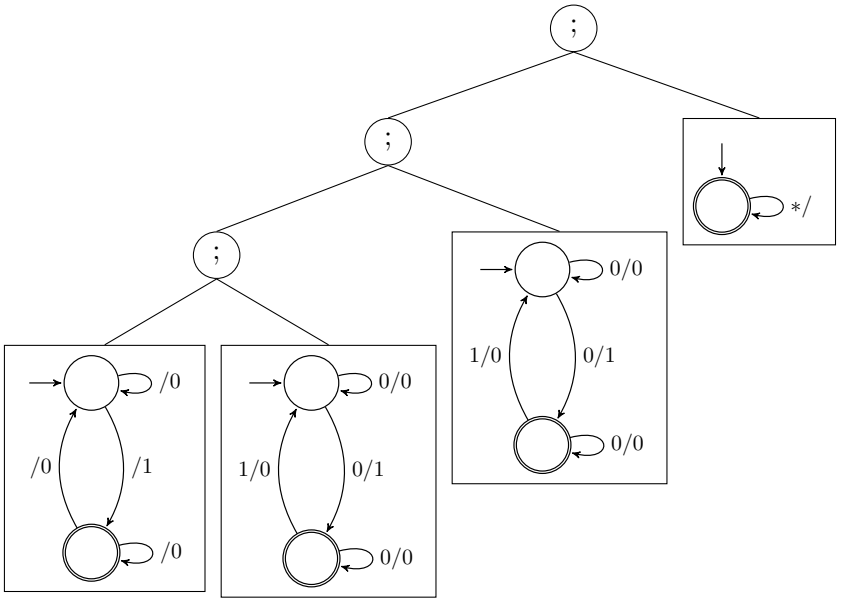


Figure 5.10: Expr. tree of Fig. 5.9, after performing a single 2-NFA composition

At this stage, we have generated two isomorphic NFAs; one (Fig. 5.3) by the naive, *monolithic* Algorithm 5.1 and the other (shown in Fig. 5.13) by the *compositional* Algorithm 5.2. In the next section, we investigate the performance of Algorithm 5.2, while we defer a proof of its correctness—that it and Algorithm 5.1 *always* generate isomorphic NFAs—to the final section.

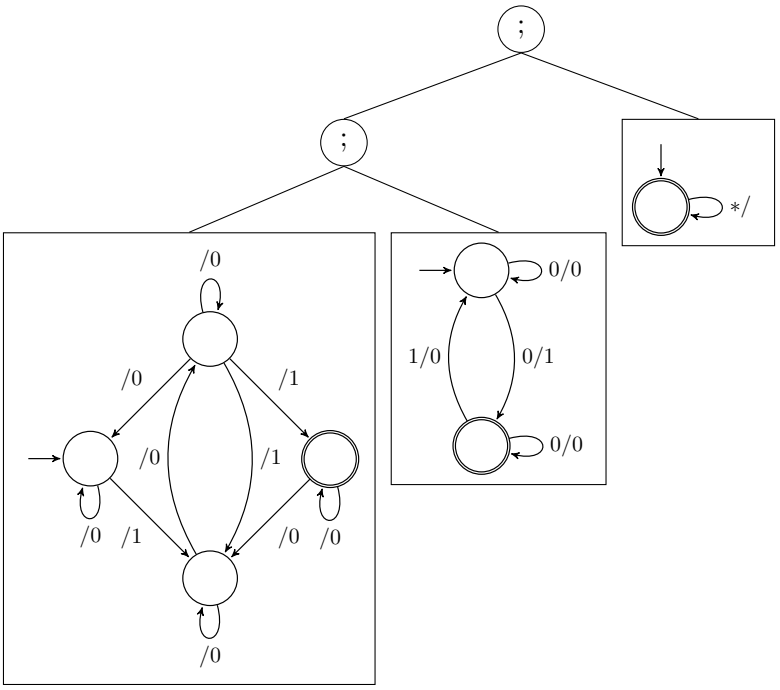


Figure 5.11: Expr. tree of Fig. 5.9, after performing two 2-NFA compositions

5.2 Performance of the Compositional Algorithm

We now turn to examining the performance of Algorithm 5.2, using the example systems introduced in §4.1. To do so, we must first specify the particular initial and target markings that we will use for each system. These markings are specified in Fig. 5.14 (some are expected to be reachable, some not).

Given these marked examples, we execute our implementation, recording the total time taken, averaged over 5 runs. The results are illustrated in Table 5.1.

Consider the first entry in Table 5.1, that of BUFFER(3), which we evaluated by hand at the end of §5.1: the maximum composition size encountered there (illustrated in Fig. 5.12) was $(8, 1)$, agreeing with the recorded result.

For all systems, small parameters lead to run-times of < 1 second; but small increments in the parameter sizes lead to vastly slower run-times, of many seconds, accompanied with large composition sizes. Indeed, moving from a parameter of 4 to 5 for OVERTAKE$(-)$ is catastrophic for the algorithm's performance, with a much larger 2-NFA composition encountered.

Indeed, the largest encountered compositions grows exponentially with the size of the system - indeed, this should be intuitively clear, since the number of markings of a net (and thus the number of states of its 2-NFA) is exponential in the number of places.

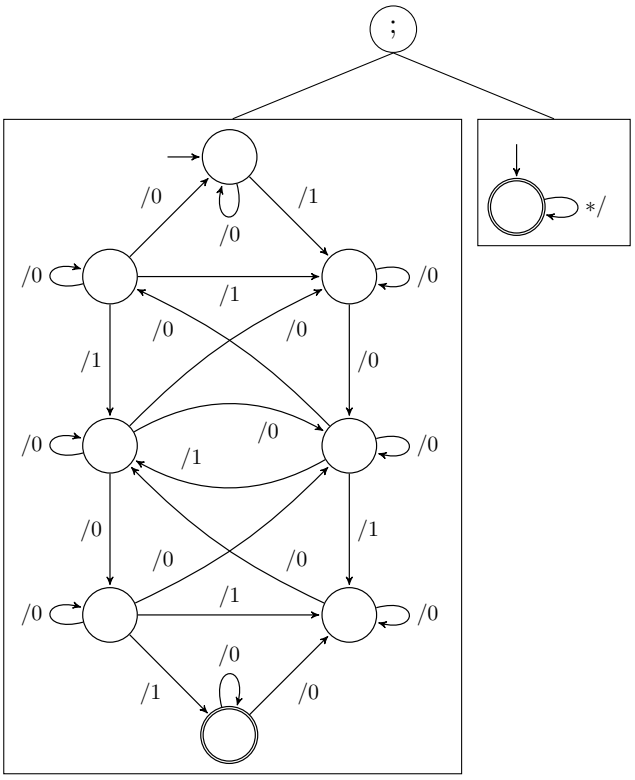


Figure 5.12: Expr. tree of Fig. 5.9, after performing three 2-NFA compositions

A final point to note is that in some cases, in particular, TOKENRING$(-)$,OVERTAKE$(-)$, and CYCLIC$(-)$, the result for $k = 1$ differs to that for higher values of k. Indeed, inspecting the desired markings for these systems, we observe that they specify the target marking as a behaviour of all components, which can/cannot be reached for multiple components, but however *is* easily reachable for single components.

Summarising these results, and to set the scene for the following chapter, we can extract some key points that we will address:

1. Without mitigation, the statespace explosion leads to NFA sizes that grow exponentially. We will investigate *weak language preserving* reductions to reduce NFA sizes, while preserving correctness (§6.1).

2. The behaviour of BUFFER(3) should intuitively be no different to that of BUFFER(4), yet the current compositional algorithm does not exploit this. We will investigate *fixed-points* of behaviour in such systems, and use *memoisation* to avoid repeated work in their presence (§6.2).

3. While PNB composition *is* associative (up to isomorphism) w.r.t. the generated 2-NFA, we will see that when employing reduction, it is *not* associative w.r.t. the (size of) intermediate 2-NFAs generated and thus the *performance* of the algorithm. We will show that the performance is subtly affected by reassociating certain compositions (§6.3).

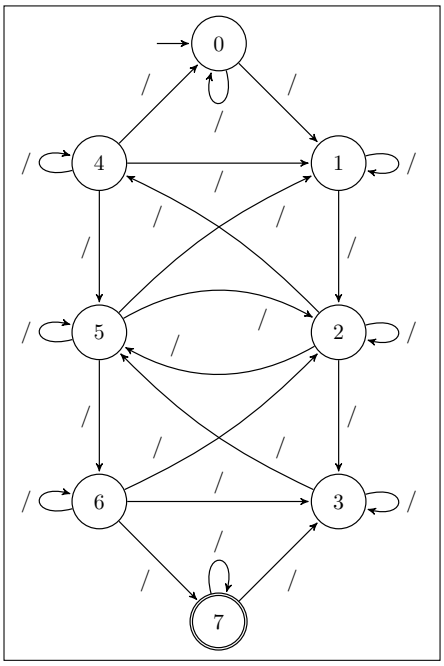


Figure 5.13: Expression tree of Fig. 5.9 after performing all compositions

5.3 Proof of Correctness

In the previous section we gave an intuitive view of the first step of our compositional approach to checking reachability in systems modelled by PNBml expressions. To complete this chapter, we now prove that Algorithm 5.2 is correct. Correctness in this case refers to obtaining isomorphic 2-NFAs by Algorithm 5.2 and Algorithm 5.1.

First, we give a formal definition of PNB expressions. As discussed in Remark 5.1, we evaluate PNBml expressions to PNB expressions, which are formed of two types of composition and component nets. Therefore, we represent PNB expressions using *wiring expressions*; a wiring expression is the abstract syntax tree of a PNB expression, where internal nodes are labelled with either ; or $\otimes$, and leaves are variables. The grammar of wiring expressions is simply:

$$T \ ::= \ x \mid T \ ; T \mid T \otimes T$$

where we again take ';' and '$\otimes$' to be left associative, with '$\otimes$' binding tighter than ';'.

Now, a wiring expression, t, together with an assignment map, $\mathcal{V}$, taking variables to *marked* PNBs, can be evaluated recursively to obtain a single marked PNB, which we write as $[\![t]\!]_{\mathcal{V}}$. However, as mentioned in Remark 5.2, before we can define this evaluation, we require definitions of synchronous and tensor composition on *marked*

system: **BUFFER**$(-)$ (§4.1.1)
initial: "empty": only tokens in upper places
target (✓): "full": only tokens in lower places

system: **TOKENRING**$(-)$ (§4.1.2)
initial: System ready: workers idle
target (✗): System started: *all* workers working

system: $T_\wedge(-,-)$ (§4.1.4)
initial: Root contains the only token
target (✓): Leaves each contain a token

system: $T_\vee(-,-)$ (§4.1.4)
initial: Root contains the only token
target (✗): Leaves each contain a token

system: **CLIQUE**$(-)$ (§4.1.5)
initial: Injector contains the only token
target (✓): Final (rightmost) place contains the only token

system: **POWERSET**$(-)$ (§4.1.6)
initial: Injector contains the only token
target (✓): All places except root contain a token

system: **OVERTAKE**$(-)$ (§4.2.1)
initial: All locks and interfaces unlocked, each car ready
target (✗): All cars overtaking

system: **HARTSTONE**$(-)$ (§4.2.2)
initial: Controllers, tasks and master all ready
target (✓): Each task working

system: **ITER-CHOICE**$(-)$ (§4.2.3)
initial: ADDTOK contains the only token
target (✓): Alternating taken/not-taken marking, ADDTOK has no token

system: **REPLICATORS**$(-)$ (§4.2.4)
initial: ADDTOK contains the only token
target (✓): TAKETOK contains the only token

system: **DAC**$(-)$ (§4.2.5)
initial: Controller and workers ready
target (✗): Every worker and controller awaiting a join

system: **DPH**$(-)$ (§4.2.6)
initial: Philos ready, forks present
target (✓): Every fork taken

system: **CYCLIC**$(-)$ (§4.2.7)
initial: Injector contains token, each scheduler ready
target (✓): Each task working

system: **COUNTER**$(-)$ (§4.2.8)
initial: Counter set to 0: all components zero
target (✓): Counter set to n: tester recognises counter is full

Figure 5.14: Example markings used in benchmarking Algorithm 5.2. Those marked ✓ are expected to be reachable and those ✗ are not

PNBs:

$$(N, m, n)\ ;\ (M, o, p) = (N\ ;\ M, m \uplus o, n \uplus p)$$
$$(N, m, n) \otimes (M, o, p) = (N \otimes M, m \uplus o, n \uplus p)$$

we assure ourselves that the definition is correct by a simple lemma:

Lemma 5.3.
For PNBs, N and M, m and n are markings of N and M, respectively, iff $m \uplus n$ is a marking of $N\ ;\ M$ (and $N \otimes M$).

Table 5.1: Checking reachability of markings in Fig. 5.14, using Algorithm 5.2. Key: M = Maximum # States in a Composition, T = Time (s), R? = Reachable?

Sys	M	T	R?	Sys	M	T	R?
BUFFER(3)	(8,1)	0.004	✓	HARTSTONE(2)	(6,8)	0.039	✗
BUFFER(6)	(64,1)	0.027	✓	HARTSTONE(4)	(6,32)	0.136	✗
BUFFER(9)	(512,1)	0.258	✓	HARTSTONE(8)	(6,512)	2.696	✗
BUFFER(12)	(4096,1)	3.270	✓	HARTSTONE(10)	(6,2048)	12.174	✗
TOKENRING(1)	(12,1)	0.017	✓	ITER-CHOICE(1)	(2,16)	0.008	✓
TOKENRING(2)	(72,1)	0.092	✗	ITER-CHOICE(2)	(2,256)	0.042	✓
TOKENRING(4)	(2592,1)	5.422	✗	ITER-CHOICE(3)	(2,4096)	0.790	✓
TOKENRING(5)	(15552,1)	49.881	✗	ITER-CHOICE(4)	(2,65536)	17.548	✓
$T_\wedge(1,10)$	(2,1024)	1.506	✓	REPLICATORS(1)	(2,4)	0.003	✓
$T_\wedge(2,4)$	(104,104)	3.560	✓	REPLICATORS(3)	(2,32)	0.014	✓
$T_\wedge(4,2)$	(10,4)	0.013	✓	REPLICATORS(6)	(2,851)	0.527	✓
$T_\wedge(12,2)$	(2050,4)	19.455	✓	REPLICATORS(8)	(2,7655)	7.557	✓
$T_\vee(1,8)$	(2,256)	0.372	✓	DAC(10)	(65,5)	0.107	✗
$T_\vee(2,3)$	(2,16384)	10.741	✗	DAC(25)	(350,5)	0.930	✗
$T_\vee(3,2)$	(2,4096)	2.188	✗	DAC(50)	(1325,5)	6.178	✗
$T_\vee(15,1)$	(2,32768)	26.002	✗	DAC(75)	(2925,5)	19.702	✗
CLIQUE(2)	(16,1)	0.030	✓	DPH(1)	(8,1)	0.038	✓
CLIQUE(4)	(64,1)	0.134	✓	DPH(2)	(8,8)	0.114	✓
CLIQUE(7)	(512,1)	2.314	✓	DPH(4)	(72,8)	1.146	✓
CLIQUE(9)	(2048,1)	20.200	✓	DPH(6)	(648,8)	12.689	✓
POWERSET(3)	(2,8)	0.006	✓	CYCLIC(1)	(2,5)	0.016	✗
POWERSET(6)	(2,64)	0.055	✓	CYCLIC(2)	(2,18)	0.036	✓
POWERSET(9)	(2,512)	0.686	✓	CYCLIC(4)	(2,278)	0.512	✓
POWERSET(12)	(2,4096)	12.410	✓	CYCLIC(6)	(2,4438)	11.273	✓
OVERTAKE(1)	(12,7)	0.068	✓	COUNTER(1)	(2,1)	0.003	✓
OVERTAKE(3)	(20,102)	0.596	✗	COUNTER(2)	(4,1)	0.010	✓
OVERTAKE(4)	(20,594)	3.985	✗	COUNTER(4)	(16,1)	0.056	✓
OVERTAKE(5)	(20,3462)	32.502	✗	COUNTER(8)	(256,1)	1.238	✓

Proof. Immediate, since $\text{places}(N \ ; \ M) = \text{places}(N) \uplus \text{places}(M) = \text{places}(N \otimes M)$. □

Now we may define $[\![t]\!]_\mathcal{V}$:

$$[\![x]\!]_\mathcal{V} = \mathcal{V}(x)$$
$$[\![t_1 \ ; \ t_2]\!]_\mathcal{V} = [\![t_1]\!]_\mathcal{V} \ ; \ [\![t_2]\!]_\mathcal{V}$$
$$[\![t_1 \otimes t_2]\!]_\mathcal{V} = [\![t_1]\!]_\mathcal{V} \otimes [\![t_2]\!]_\mathcal{V}$$

where we implicitly assume that variable assignments are compatible with t, in the sense that only nets with compatible boundaries are composed. However, recall that in §4.3, we introduced a type system for the more expressive language, PNBml, which disallows invalid expressions, and could be trivially adapted to wiring expressions.

Example 5.1.
The following are the wiring expression and variable assignment corresponding to (5.1)*:*

$$t = x_1 \;;\; x_2 \;;\; x_2 \;;\; x_2 \;;\; x_3 \qquad \mathcal{V} = \{x_1 \mapsto \top, x_2 \mapsto \textsf{BUFFER}, x_3 \mapsto \bot\}$$

observe that $[\![t]\!]_{\mathcal{V}}$ *is isomorphic to the PNB shown in Fig. 5.1.*

At this point, we have defined the operation to collapse a wiring expression with corresponding variable assignment, into a single composite *PNB*. We now turn to defining an operation to collapse a wiring expression into a single *2-NFA*.

Given a wiring expression t and variable assignment $\mathcal{V}$ we can recursively translate t into a 2-NFA, which we write as $\langle\!\langle t \rangle\!\rangle_{\mathcal{V}}$. Recall from Defn. 2.43, that for a marked PNB N, $\langle\!\langle N \rangle\!\rangle$ is the corresponding 2-NFA encoding reachability. We define $\langle\!\langle t \rangle\!\rangle_{\mathcal{V}}$ as follows:

$$\begin{aligned}
\langle\!\langle x \rangle\!\rangle_{\mathcal{V}} &= \langle\!\langle \mathcal{V}(x) \rangle\!\rangle \\
\langle\!\langle t_1 \;;\; t_2 \rangle\!\rangle_{\mathcal{V}} &= \langle\!\langle t_1 \rangle\!\rangle_{\mathcal{V}} \;;\; \langle\!\langle t_2 \rangle\!\rangle_{\mathcal{V}} \\
\langle\!\langle t_1 \otimes t_2 \rangle\!\rangle_{\mathcal{V}} &= \langle\!\langle t_1 \rangle\!\rangle_{\mathcal{V}} \otimes \langle\!\langle t_2 \rangle\!\rangle_{\mathcal{V}}
\end{aligned}$$

Finally, we can state and prove correctness: converting a wiring expression to a PNB and *then* constructing its 2-NFA semantics is equivalent to *directly constructing* the 2-NFA semantics of the wiring expression. This property is known as *compositionality*.

Proposition 5.4 (Compositionality)**.**
$\langle\!\langle t \rangle\!\rangle_{\mathcal{V}} \cong \langle\!\langle [\![t]\!]_{\mathcal{V}} \rangle\!\rangle$

Proof. We proceed by induction on the structure of t:

- In the base case of a variable, the result is trivial: the 2-NFAs on both sides are equal by definition.
- In the case of $t_1 \;;\; t_2$, we must show that $\langle\!\langle t_1 \;;\; t_2 \rangle\!\rangle_{\mathcal{V}} \cong \langle\!\langle [\![t_1 \;;\; t_2]\!]_{\mathcal{V}} \rangle\!\rangle$. After expanding the definitions of $\langle\!\langle - \rangle\!\rangle_{\mathcal{V}}$ and $[\![-]\!]_{\mathcal{V}}$, we obtain:

$$\langle\!\langle t_1 \rangle\!\rangle_{\mathcal{V}} \;;\; \langle\!\langle t_2 \rangle\!\rangle_{\mathcal{V}} \cong \langle\!\langle [\![t_1]\!]_{\mathcal{V}} \;;\; [\![t_2]\!]_{\mathcal{V}} \rangle\!\rangle \tag{5.2}$$

  To proceed, we apply the I.H. to t_1 and t_2, obtaining

$$\langle\!\langle t_1 \rangle\!\rangle_{\mathcal{V}} \cong \langle\!\langle [\![t_1]\!]_{\mathcal{V}} \rangle\!\rangle \tag{5.3}$$

  and

$$\langle\!\langle t_2 \rangle\!\rangle_{\mathcal{V}} \cong \langle\!\langle [\![t_2]\!]_{\mathcal{V}} \rangle\!\rangle \tag{5.4}$$

Substituting 5.3 and 5.4 into 5.2 gives:

$$\langle\!\langle [\![t_1]\!]_{\mathcal{V}} \rangle\!\rangle \; ; \langle\!\langle [\![t_2]\!]_{\mathcal{V}} \rangle\!\rangle \cong \langle\!\langle [\![t_1]\!]_{\mathcal{V}} \; ; \; [\![t_2]\!]_{\mathcal{V}} \rangle\!\rangle$$

Now, letting, $(N, m, n) = [\![t_1]\!]_{\mathcal{V}}$ and $(M, o, p) = [\![t_2]\!]_{\mathcal{V}}$, we obtain:

$$\langle\!\langle (N, m, n) \rangle\!\rangle \; ; \langle\!\langle (M, o, p) \rangle\!\rangle \cong \langle\!\langle (N, m, n) \; ; \; (M, o, p) \rangle\!\rangle$$

Now, by Prop. 3.44, the underlying 2-LTSs are isomorphic, thus it only remains to show that states are initial/accepting in the LHS iff they are in the RHS. A state of $\langle\!\langle (N, m, n) \rangle\!\rangle \; ; \; \langle\!\langle (M, o, p) \rangle\!\rangle$, is $(x, \, y) \in 2^{\text{places}(N)} \times 2^{\text{places}(M)}$, and is initial iff $x = m$ and $y = o$; applying the state component of the 2-LTS isomorphism, we obtain the state $\{\text{inl}\, p \mid p \in x\} \cup \{\text{inr}\, q \mid q \in y\}$, i.e. $m \uplus n$, which is initial in $\langle\!\langle (N, m, n) \; ; \; (M, o, p) \rangle\!\rangle$ by the definition of composition on marked PNBs. In the reverse direction, a state of $\langle\!\langle (N, m, n) \; ; \; (M, o, p) \rangle\!\rangle$, is $z \in 2^{\text{places}(N) \uplus \text{places}(M)}$, and initial iff $z = m \uplus o$. The state component of the isomorphism partitions this state into $(m, \, o) \in 2^{\text{places}(N)} \times 2^{\text{places}(M)}$, which is initial in $\langle\!\langle (N, m, n) \rangle\!\rangle \; ; \; \langle\!\langle (M, o, p) \rangle\!\rangle$. The case for accepting states follows the same argument.

- Finally, the case for $t_1 \otimes t_2$, is similar, but using Prop. 3.47 in place of Prop. 3.44.

□

5.4 Conclusion

In this chapter we introduced a technique for *compositional* statespace generation of systems specified using PNBs, using it to form the basis of a reachability check. We proved the technique correct and illustrated the performance of the technique using the example systems of Chapter 4. Our results highlighted that without mitigation, statespace explosion leads to poor performance, even in small systems. To combat this, in the next chapter, we show that by exploiting the component-wise specification of systems, we can mitigate the statespace explosion problem and obtain good performance.

Chapter 6

Efficient Compositional Reachability Checking

In the previous chapter we introduced a technique for compositional statespace generation for systems modelled using PNBs. We proved that it was correct, but its performance left something to be desired — indeed, even for reasonably small examples, the generated intermediate 2-NFAs were large, hindering performance of 2-NFA composition and thus the overall technique.

In this chapter, we describe improvements that can be made to the naive compositional algorithm to improve its performance, such that it can be used to efficiently check marking reachability. Briefly, the two key improvements are: we exploit the fact that (weak) language equivalence is a congruence, to reduce intermediate 2-NFAs, and use memoisation to avoid repeated computation.

First, we introduce the notion of *boundary protocol* and internal, or τ, transitions which must be preserved by any structural reduction performed on intermediate 2-NFAs, to ensure correctness of the resulting performance improvements.

6.1 Boundary protocol and τ-transitions

For a simple marked PNB, the possibly empty collection of firing sequences (and thus boundary interactions) that will take it from the initial marking to the desired marking can be readily identified. For example, consider the simple PNB that will be our running example, illustrated in Fig. 6.1; we can see that the firing sequences are those with either a single firing of t_1, or a firing of t_2 then t_3 and t_4. Indeed, we can abstract over the identities of the transitions that must be fired, and only consider their interactions on the boundary ports, giving those sequences with either a $1/0$, or $0/0$ followed by $0/1$

and $0/0$. We refer to this collection as the *boundary protocol* of the PNB. Indeed, the boundary protocol of a (marked) PNB is precisely the *language* of its 2-NFA semantics. Intuitively, the boundary protocol encompasses all sequences of *external interactions*, that take the component between its local initial and target markings.

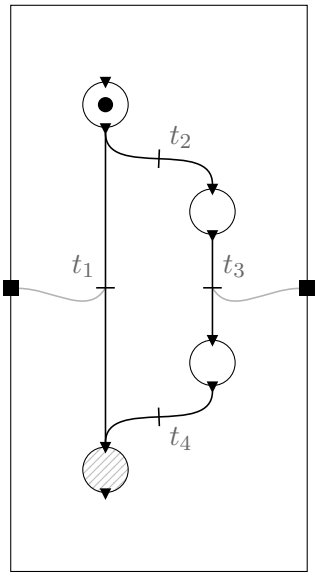


Figure 6.1: Example PNB

Since the boundary protocol of a PNB, N, corresponds to the language of $\langle\!\langle N \rangle\!\rangle$, we can apply language-preserving modifications to $\langle\!\langle N \rangle\!\rangle$ to obtain a more compact representation of N's boundary protocol. Furthermore, we are able to perform such language-preserving modifications, whilst preserving compositions of 2-NFAs:

Theorem 6.1 ((Strong) 2-NFA language equivalence is a congruence)**.**
Suppose that:

(i) N, N' *are* (k, l)*-NFAs, with* $N \sim_{\mathcal{L}} N'$*,*

(ii) M, M' *are* (l, m)*-NFAs, with* $M \sim_{\mathcal{L}} M'$*,*

(iii) O, O' *are* (m, n)*-NFAs, with* $O \sim_{\mathcal{L}} O'$*.*

Then, the following hold:

(i) $N \,;\, M \sim_{\mathcal{L}} N' \,;\, M'$*,*

(ii) $N \otimes O \sim_{\mathcal{L}} N' \otimes O'$*.*

Proof. For (i), suppose that $\overrightarrow{\alpha/\beta} \in \mathcal{L}(N \,;\, M)$. Then, there exists a $\vec{\gamma}$, such that $\overrightarrow{\alpha/\gamma} \in \mathcal{L}(N)$ and $\overrightarrow{\gamma/\beta} \in \mathcal{L}(M)$. By the asssumptions that $N \sim_{\mathcal{L}} N'$ and $M \sim_{\mathcal{L}} M'$, then also $\overrightarrow{\alpha/\gamma} \in \mathcal{L}(N')$ and $\overrightarrow{\gamma/\beta} \in \mathcal{L}(M')$ and thus $\overrightarrow{\alpha/\beta} \in \mathcal{L}(N' \,;\, M')$, as required. For (ii), suppose that $\overrightarrow{\alpha\gamma/\beta\delta} \in \mathcal{L}(N \otimes O)$. Then, we have that $\overrightarrow{\alpha/\beta} \in \mathcal{L}(N)$ and $\overrightarrow{\gamma/\delta} \in \mathcal{L}(O)$.

By the assumptions, we have that $\overrightarrow{\alpha/\beta} \in \mathcal{L}\left(N'\right)$ and $\overrightarrow{\gamma/\delta} \in \mathcal{L}\left(O'\right)$ and thus $\overrightarrow{\alpha\gamma/\beta\delta} \in \mathcal{L}\left(N' \otimes O'\right)$, as required. □

Observe that Thm. 6.1 is the (strong) language equivalence analogue of Prop. 3.33(i) and Prop. 3.35(i); indeed, both language-equivalence and isomorphism are congruences. Intuitively, being a congruence means that we are free to reduce component 2-NFAs, whilst ensuring that we preserve the language of their composition. Thus, to reduce the size of 2-NFAs generated by our technique, and therefore improve its performance, we can replace any 2-NFA that arises with one that is smaller, yet language equivalent.

In fact, we can do even better, by considering the boundary protocol (and thus 2-NFA semantics) only *up-to internal behaviour*. By ignoring internal computations, we can reduce the size of 2-NFAs even further and indeed, in some cases, reach fixed-points of behaviour w.r.t. composition, leading to *linear* time complexity. First, we introduce τ-transitions as the representation of internal behaviour that we wish to ignore.

6.1.1 τ-transitions in 2-NFA semantics

Recall that every state in the (k, l)-NFA semantics of a PNB, $N : (k, l)$ has a self-loop labelled with $0^k/0^l$, since $\varnothing \subseteq \mathrm{trans}(N)$ can always be fired with no effect on the boundaries. In general, such-labelled 2-NFA-transitions need not be self-loops and we refer to them as $\tau_{(k,l)}$-transitions (recall that we drop the subscripts when k and l are clear from the context). Indeed, firing any (set of) net-transitions that do not connect to the left or right boundaries will generate τ-transitions. One particular source of such net-transitions is composition of PNB components with transitions that connect only to *shared* boundary ports.

Our semantic model does not distinguish between those τ-transitions originating from firing the empty set of transitions and those from firing a set of transitions unconnected to the boundaries. In other words, we abstract over internal behaviour in that "no behaviour" and "no externally observable behaviour" are represented the same. It is precisely this abstraction that allows us to reduce the generated statespace: statespace that cannot be distinguished by observable interactions can simply be forgotten.

To describe the behaviour that we wish to preserve, we consider a notion of boundary protocol that only incorporates observable boundary interactions. We say that the *weak boundary protocol* of a PNB is its boundary protocol with all τ labels removed. The removal of τ labels is realised as the unique monoid homomorphism:

$$\widehat{\ } : \left(\mathbb{B}^k \times \mathbb{B}^l\right)^* \to \left(\mathbb{B}^k \times \mathbb{B}^l \setminus \tau_{(k,l)}\right)^*$$

defined on individual elements, e, of $\mathbb{B}^k \times \mathbb{B}^l$ as follows:

$$\widehat{e} = \begin{cases} \epsilon \text{ if } e = \tau_{(k,l)} \\ e \text{ otherwise} \end{cases}$$

Definition 6.2.
Given a (k, l)-NFA, N, the weak language of N, written $\mathcal{L}^{\setminus\tau}(N)$, is:

$$\mathcal{L}^{\setminus\tau}(N) \stackrel{\text{def}}{=} \left\{\widehat{x} \;\middle|\; x \in \mathcal{L}(N)\right\}$$

Definition 6.3.
For (k, l)-NFAs, N and M, we say N and M are weak language-equivalent, written $N \approx_{\mathcal{L}} M$ if:

$$\mathcal{L}^{\setminus\tau}(N) = \mathcal{L}^{\setminus\tau}(M)$$

As an example, consider the 2-NFA semantics of the simple PNB in Fig. 6.1, which illustrated in Fig. 6.2.

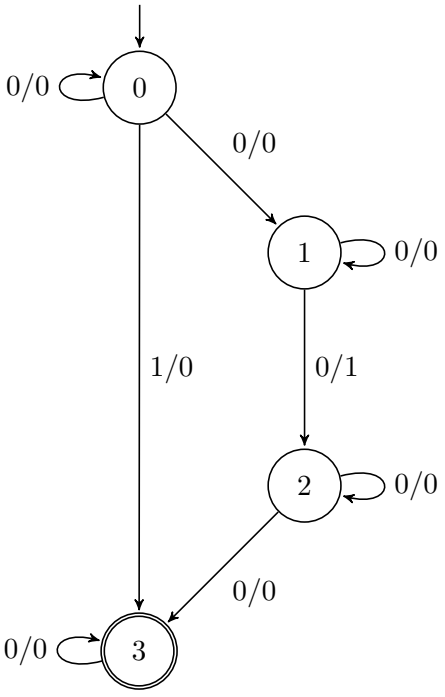


Figure 6.2: $(1, 1)$-NFA semantics, N, of the PNB in Fig. 6.1

The language of this 2-NFA is[1]: $(0/0)^* \left(1/0 \;\middle|\; 0/0\,(0/0)^*\,0/1\,(0/0)^*\,0/0\right)(0/0)^*$, whereas the weak language is much simpler: $(1/0 \mid 0/1)$. A simple observation is that $N \approx_{\mathcal{L}} M$, where M is illustrated in Fig. 6.3.

[1] Using a regular expression notation.

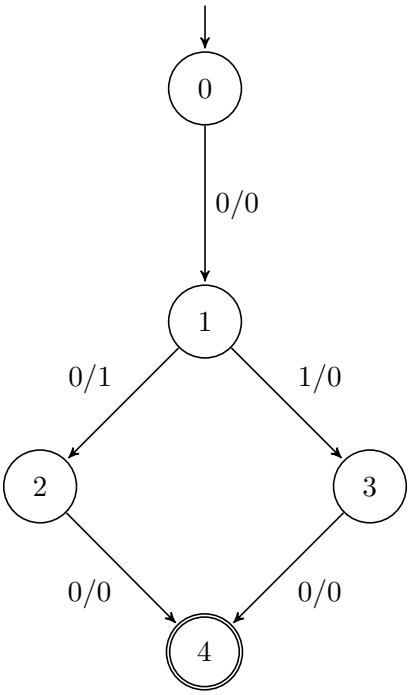


Figure 6.3: M, weakly-equivalent to N shown in Fig. 6.2

Now, we have shown how we can consider equivalence up-to "internal behaviour" of 2-NFAs using weak-language equivalence; however, to improve the performance of 2-NFA composition (and thus our compositional technique), we must reduce the size of component 2-NFAs, by modifying their structure, yet preserving *weak language* to ensure correctness.

The first step of this procedure is to ignore τ-transitions, by *closing* the 2-NFA w.r.t. them, which requires the notion of the τ-closure of a single state. τ-closure is closely related to ϵ-closure from automata theory, as we discuss in Remark 6.9. The τ-closure of a single state is the set of states that can be reached by taking zero or more transitions labelled by τ:

Definition 6.4 (State τ-closure)**.**
For a state, x, of a (k, l)-NFA, $(Q, \Sigma, \rightarrow, i, A)$, we define its τ-closure*, written τ-cl(x) as the least fixed-point of the following recursive definition:*

$$\tau\text{-cl}(x) = \{x\} \cup \bigcup \left\{ \tau\text{-cl}(y) \;\middle|\; x \xrightarrow{\tau_{(k,l)}} y \right\}$$

Given the τ-closure of a single state, we can define the τ-closure of an entire 2-NFA. Intuitively, the τ-closure of a 2-NFA, N, is formed by "saturating" N with transitions that remove τs from traces:

1. Any state whose τ-closure contains an accepting state is also accepting, allowing us to ignore trailing τs in any trace,

2. The transition relation has new entries corresponding to the pre-closure of the transition relation w.r.t. τ-closure, allowing us to ignore leading τs in any trace.

Precisely, the definition is as follows:

Definition 6.5 (2-NFA τ-closure)**.**
For a $(k,\, l)$-NFA, N, formed of $(Q, \Sigma, \rightarrow, i, A)$, its τ-closure, τ-cl(N), is:

$$(Q, \Sigma, \rightarrow \cup \rightarrow', i, A \cup A')$$

where:

$$A' = \left\{ x \;\middle|\; x \in Q, \tau\text{-cl}(x) \cap A \neq \varnothing \right\}$$
$$\rightarrow' = \left\{ (x, \sigma, y) \;\middle|\; \exists x' \in \tau\text{-cl}(x) \text{ s.t. } x' \xrightarrow{\sigma} y \right\}$$

In general, it is obvious that τ-closure changes the strong language of a 2-NFA:

Lemma 6.6 (Strong language is not preserved by τ-closure)**.**
For a 2-NFA, N, it is not necessarily the case that: $N \sim_{\mathcal{L}} \tau$-cl$(N)$.

Proof. For an example, consider again the $(1,\, 1)$-NFA, N, in Fig. 6.2 and its τ-closure, τ-cl(N), in Fig. 6.4. The languages of these 2-NFA are clearly different; for example, the word $\langle 0/1 \rangle$ is in the language of τ-cl(N), but not of N, therefore, $N \not\sim_{\mathcal{L}} \tau$-cl$(N)$. □

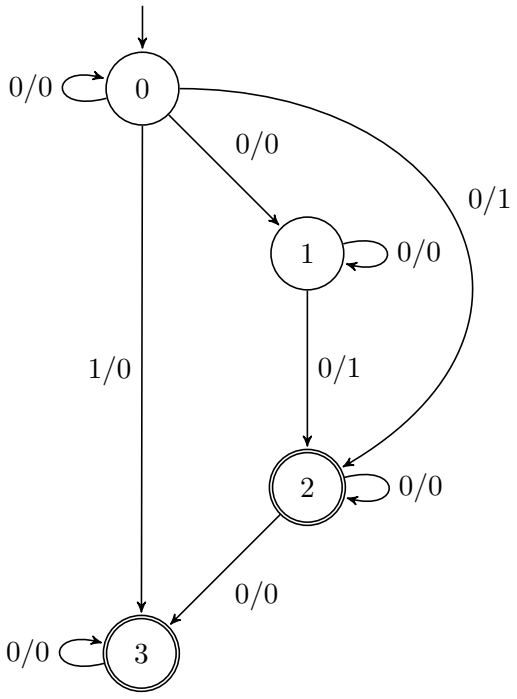


Figure 6.4: $(1,\, 1)$-NFA of Fig. 6.2 after τ-closure

Indeed, there are certain cases where strong language is *not* affected (e.g. a 2-NFA that contains no τ-transitions, or has no accepting states), but we are not concerned with such 2-NFA; we therefore only prove that weak language *is* preserved.

First, before proving that weak language is preserved by τ-closure, we prove a technical lemma. By the construction of τ-cl(N), we have that the only additional words in $\mathcal{L}\left(\tau\text{-cl}(N)\right)$, relative to $\mathcal{L}\left(N\right)$, are words of N with τ labels removed:

Lemma 6.7.
Let $N = (Q, \Sigma, \rightarrow, i, A)$ and $M = \tau$-cl$(N) = (Q, \Sigma, \rightarrow_\tau, i, A')$. For any $w \in \mathcal{L}\left(M\right)$, with $|w| = k$, such that $w \notin \mathcal{L}\left(N\right)$, we can construct a word $w' \in \mathcal{L}\left(N\right)$ by inserting zero or more τ labels into w.

Proof. By definition, $w \in \mathcal{L}\left(M\right)$ implies there are states $x_0, x_1, \ldots, x_k \in Q$, such that there are transitions $x_{i-1} \xrightarrow{w_i}_\tau x_i$, for $1 \leq i \leq k$, with $x_0 = i$, and $x_k \in A'$. Indeed, by the definition of τ-closure, each *single* transition of M, $x_{i-1} \xrightarrow{w_i}_\tau x_i$, for $1 \leq i \leq k$ is either a transition of N, or corresponds to a (not necessarily unique) *sequence* of one or more transitions of N (between the same pair of states: x_{i-1} and x_i), with the final transition of the sequence being labelled with w_i, and *every other* transition by τ. Concatenating these (sequences of) transitions gives a sequence of transitions from x_0 (the initial state), to x_k (an accepting state in M). If $x_k \in A$, we are done, otherwise, by the definition of A', there exists a sequence of τ transitions from x_k, that reach some $x_o \in A$; appending this final sequence of τ transitions, gives the required $w' \in \mathcal{L}\left(N\right)$. □

A graphical presentation of the idea of the proof of Lem. 6.7 is given in Fig. 6.5, the transitions forming word $w \in \mathcal{L}\left(M\right)$ are highlighted in blue. For each such transition, either it is also a transition of N, or there will exist a sequence of transitions in N, between the same start and end states.

Now we prove that weak language *is* preserved by τ-closure:

Lemma 6.8 (Weak language is preserved by τ-closure)**.**
For any 2-NFA, N, we have: τ-cl$(N) \approx_{\mathcal{L}} N$.

Proof. By the definition of weak language equivalence, we must have the following:

$$\left\{\widehat{w} \mid w \in \mathcal{L}\left(N\right)\right\} = \left\{\widehat{v} \;\middle|\; v \in \mathcal{L}\left(\tau\text{-cl}(N)\right)\right\}$$

By construction, $w \in \mathcal{L}\left(N\right) \implies w \in \mathcal{L}\left(\tau\text{-cl}(N)\right)$ since we have not *removed* transitions or initial/accepting states. The converse does not hold; however, for any $w \in \mathcal{L}\left(\tau\text{-cl}(N)\right)$ such that $w \notin \mathcal{L}\left(N\right)$, then by Lem. 6.7, there exists a $v \in \mathcal{L}\left(N\right)$, such w and v only differ in τs; however, since weak language equivalence removes *all* τ labels from words, w and v will be considered equal, as required. □

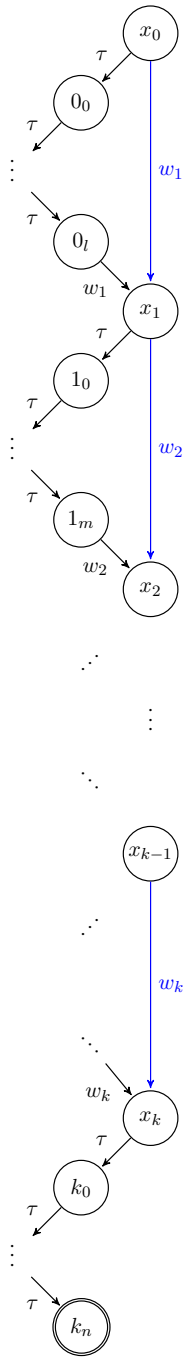


Figure 6.5: Expansion of a word using only additional τ labels

Remark 6.9. The reader may observe that τ transitions and τ-closure are similar to ϵ-moves and closure of ϵ-NFA in Automata theory. Indeed, they are very similar: our lemma regarding weak-language preservation by τ-closure is the analogue of the theorem stating that ϵ-closed ϵ-NFAs recognises the same language as the original. However, there are subtle differences: ϵ is not considered as part of the alphabet of a ϵ-NFA and thus doesn't consume any of the input string when taking the transition. Furthermore, ϵ-closure removes all transitions labels from the ϵ-NFA, whereas our τ-closure simply supplements the existing transitions and accepting states. Indeed, we could

have used the standard definition of ϵ-closure, making an alternative design decision: the theorems required to prove correctness would be slightly altered and indeed, we would need to insert "artificial" transitions to ensure the important property of *reflexivity* (discussed later, in Defn. 6.10) was preserved, but essentially the choice is inconsequential.

We want to be sure that weak language equivalence is also a congruence w.r.t. 2-NFA compositions. However, it is *not* in general, which we demonstrate by way of an example: consider the 2-NFAs, N, shown in Fig. 6.6, M, shown in Fig. 6.7a, and M', in Fig. 6.7b. It is easy to verify that $M \approx_{\mathcal{L}} M'$, thus if weak language equivalence was a congruence, we should have $N \,;\, M \approx_{\mathcal{L}} N \,;\, M'$. This does not hold: the LHS, $N \,;\, M$, is illustrated in Fig. 6.8a, whilst the RHS is illustrated in Fig. 6.8b; these two compositions are not weak language equivalent, since the latter's language is empty.

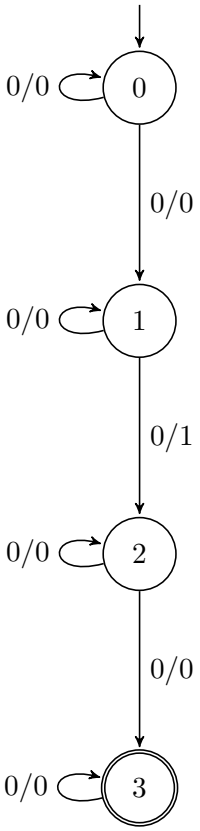
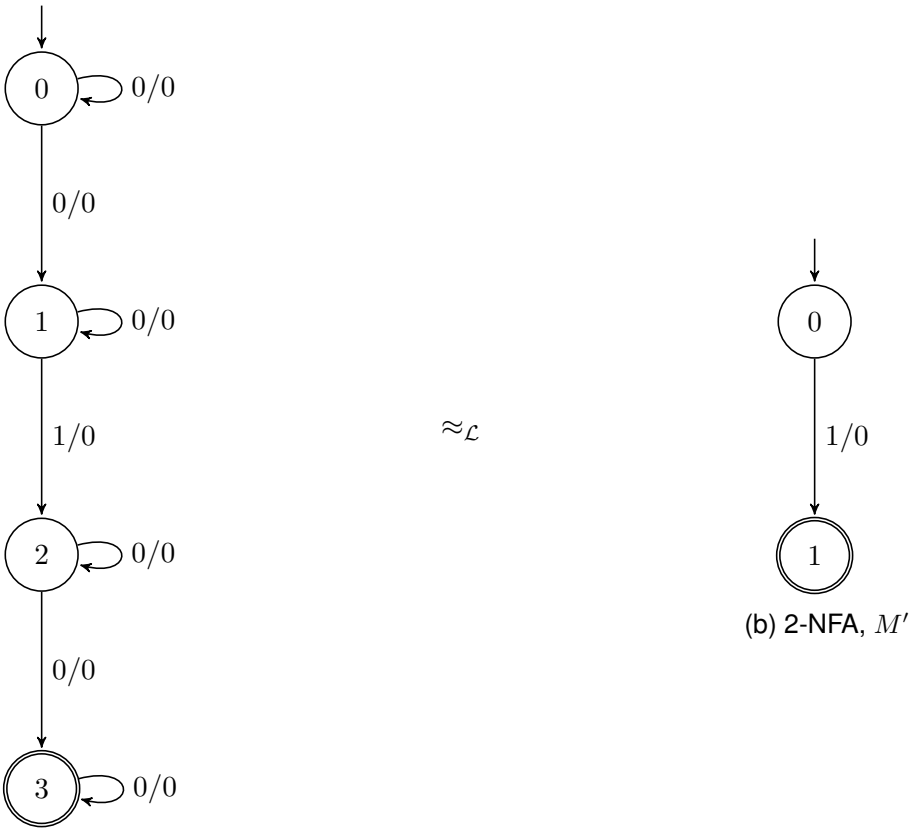


Figure 6.6: 2-NFA, N

The failure of compositionality is due to the missing τ transitions in M', whereby the $0/0$ transitions in N cannot "synchronise" with any transitions in M' and thus there can be no transitions in the composition. Indeed, since τ transitions represent "internal" behaviour, their presence allows either component to perform internal behaviour, whilst the other component "does nothing" (similarly represented by a τ transition), before synchronising on transitions that do have an effect on the common boundary. Indeed, weak language equivalence is too coarse - it equates 2-NFAs that are not equivalent in



(a) 2-NFA, M

(b) 2-NFA, M'

Figure 6.7: Component 2-NFA that are weak language equivalent

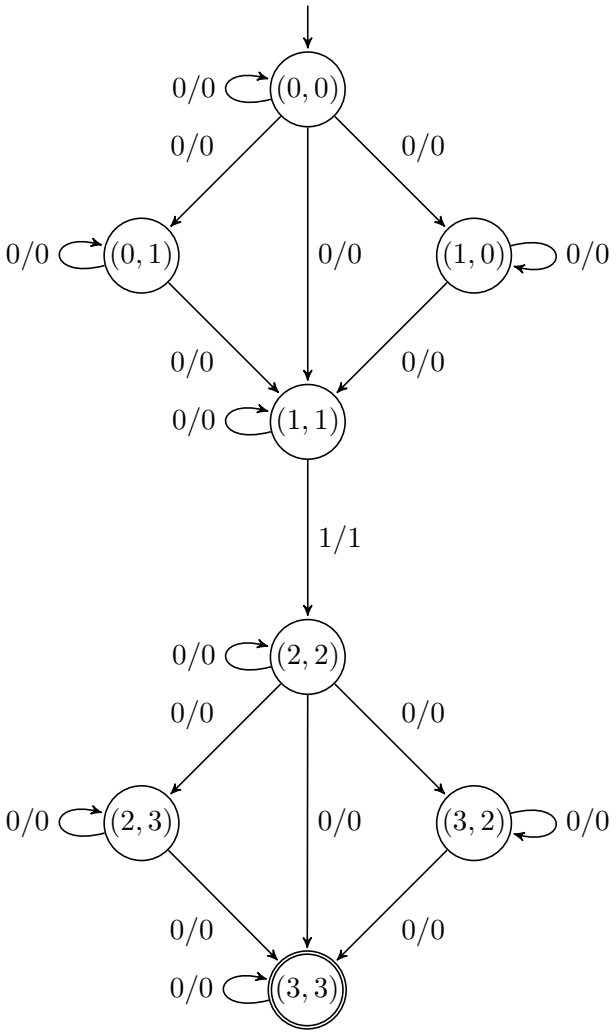
the sense of their compatibility with a common (composition) context. It is important to note that τ-closure should be thought of as abstracting over internal behaviour, rather than "do-nothing" behaviour, which is also represented by τ-transitions. Since weak-language equivalence ignores *any* τ-transition, whether representing internal or do-nothing behaviour, we must ensure that do-nothing behaviour of every state is always preserved to ensure compositionality.

6.1.2 Reflexivity and Compositionality

If were to ensure that all 2-NFAs could perform a τ transition in *each* state, the example of the previous subsection would not fail, and then indeed, weak language equivalence would be a congruence. We call 2-NFAs that can perform a τ-transition in each state *reflexive*, recognising that in each state, there is a self-loop, labelled by τ:

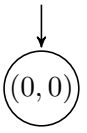
Definition 6.10 (Reflexive 2-NFA)**.**
A (k, l)-NFA, $(Q, \Sigma, \rightarrow, i, A)$ is said to be reflexive, if: $\forall x \in Q, x \xrightarrow{\tau_{(k,l)}} x$.

Ensuring reflexivity ensures that the τ transitions that correspond to "do nothing" behaviour are preserved. Indeed, composing M with the reflexive 2-NFA, N'', illustrated



(a) 2-NFA, $N \,;\, M$

$\not\approx_{\mathcal{L}}$



(b) 2-NFA, $N \,;\, M'$

Figure 6.8: Composed 2-NFA that are not weak-language equivalent

in Fig. 6.9, such that $N \approx_{\mathcal{L}} N''$, gives the required, weak-language equivalent composition: $N \,;\, M \approx_{\mathcal{L}} N \,;\, M''$, as illustrated in Fig. 6.10.

Indeed, weak language-equivalence *is* a congruence w.r.t. 2-NFA compositions, if the 2-NFAs are reflexive.

First, we prove two technical lemmas; the first says that we can place arbitrarily many τ labels inside an accepted word of a reflexive 2-NFA and obtain another accepted word:

Lemma 6.11.
Suppose N is a reflexive 2-NFA and that $w \in \mathcal{L}(N)$. We can obtain another word, $w' \in \mathcal{L}(N)$, that results from inserting finitely-many τ labels into w.

Proof. Since $w \in \mathcal{L}(N)$, any prefix of w corresponds to a state of N, at which, since N is reflexive, we are able to take a τ-labelled transition (multiple times if necessary) and

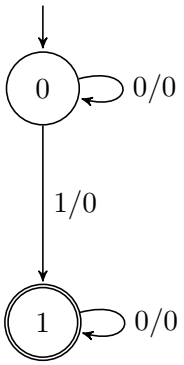


Figure 6.9: 2-NFA, M''

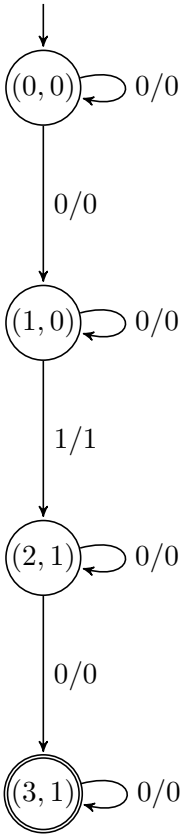


Figure 6.10: 2-NFA, N ; M'

remain in the same state. Thus, at any prefix of (i.e. position within) w we are able to insert finitely-many τ transitions and finish in the same accepting state. □

The second says that, given two *weak*-language-equivalent, reflexive 2-NFAs, and a word, w, in the language of one, we can find a word in the intersection of the two 2-NFA's languages that is weakly-equivalent to w. Since we know that the languages are equal when τ labels are disregarded, when given a word in the *language* of one, we can remove existing τ labels and insert τ labels in (potentially) different positions

to obtain a word in the second 2-NFA. Then we can pad both words with τ labels, to obtain a third word in the intersection of the 2-NFA's languages.

Lemma 6.12.
Suppose N, N' are reflexive 2-NFAs, such that $N \approx_{\mathcal{L}} N'$ and $w \in \mathcal{L}(N)$. Then, there exists a $v \in \mathcal{L}(N) \cap \mathcal{L}(N')$, with $\widehat{w} = \widehat{v}$.

Proof. By our assumption on w, we have that $\widehat{w} \in \mathcal{L}^{\setminus\tau}(N)$, and furthermore, by our assumption that $N \approx_{\mathcal{L}} N'$, $\widehat{w} \in \mathcal{L}^{\setminus\tau}(N')$. Then, there must exist a $w' \in \mathcal{L}(N')$ such that $\widehat{w} = \widehat{w'}$, i.e. w and w' are equal when disregarding τ labels. Then, since N and N' are both reflexive, by Lem. 6.11, we can pad w and w' with τ, to obtain a common $v \in \mathcal{L}(N) \cap \mathcal{L}(N')$, with $\widehat{w} = \widehat{w'} = \widehat{v}$. □

Theorem 6.13 (Weak 2-NFA language equivalence is a congruence for reflexive 2-NFAs)**.**
Suppose that:

(i) N, N' *are reflexive* (k, l)*-NFAs, with* $N \approx_{\mathcal{L}} N'$*,*

(ii) M, M' *are reflexive* (l, m)*-NFAs, with* $M \approx_{\mathcal{L}} M'$*,*

(iii) O, O' *are reflexive* (m, n)*-NFAs, with* $O \approx_{\mathcal{L}} O'$*.*

Then, the following hold:

(i) $N ; M \approx_{\mathcal{L}} N' ; M'$*,*

(ii) $N \otimes O \approx_{\mathcal{L}} N' \otimes O'$*.*

Proof. For (i), suppose that $w \in \mathcal{L}^{\setminus\tau}(N ; M)$, then, by the definition of weak language, there exists $\overrightarrow{\alpha/\beta} \in \mathcal{L}(N ; M)$, such that $\widehat{\overrightarrow{\alpha/\beta}} = w$. Thus, there exists γ such that $\overrightarrow{\alpha/\gamma} \in \mathcal{L}(N)$ and $\overrightarrow{\gamma/\beta} \in \mathcal{L}(M)$.

Now, since N, N', M and M' are reflexive 2-NFAs, with $N \approx_{\mathcal{L}} N'$ and $M \approx_{\mathcal{L}} M'$, by Lem. 6.12, we have:

1. $\overrightarrow{\alpha'/\gamma'} \in \mathcal{L}(N) \cap \mathcal{L}(N')$ with $\widehat{\overrightarrow{\alpha'/\gamma'}} = \widehat{\overrightarrow{\alpha/\gamma}}$, and
2. $\overrightarrow{\gamma''/\beta'} \in \mathcal{L}(M) \cap \mathcal{L}(M')$ with $\widehat{\overrightarrow{\gamma''/\beta'}} = \widehat{\overrightarrow{\gamma/\beta}}$,

N.B. that $\gamma' \neq \gamma''$, but $\widehat{\gamma'} = \widehat{\gamma''}$, thus we need to modify $\overrightarrow{\alpha'/\gamma'}$ and $\overrightarrow{\gamma''/\beta'}$ to obtain a pair of labels with equal common boundary component; by Lem. 6.11, we can pad $\overrightarrow{\alpha'/\gamma'}$ and $\overrightarrow{\gamma''/\beta'}$ to obtain the required labels, namely $\overrightarrow{\alpha''/\gamma'''} \in \mathcal{L}(N')$ and $\overrightarrow{\gamma'''/\beta''} \in \mathcal{L}(M')$, with $\widehat{\overrightarrow{\alpha''/\gamma'''}} = \widehat{\overrightarrow{\alpha/\gamma}}$ and $\widehat{\overrightarrow{\gamma'''/\beta''}} = \widehat{\overrightarrow{\gamma/\beta}}$. Thus we have that $\widehat{\overrightarrow{\alpha''/\beta''}} = \widehat{\overrightarrow{\alpha/\beta}}$; furthermore, since

$\overrightarrow{\alpha''/\gamma'''} \in \mathcal{L}\left(N'\right)$ and $\overrightarrow{\gamma'''/\beta''} \in \mathcal{L}\left(M'\right)$, it follows that $\overrightarrow{\alpha''/\beta''} \in \mathcal{L}\left(N' \,;\, M'\right)$. Indeed, $\widehat{\overrightarrow{\alpha''/\beta''}} = \widehat{\overrightarrow{\alpha/\beta}} = w \in \mathcal{L}^{\setminus\tau}\left(N' \,;\, M'\right)$, as required.

For (ii), suppose that $w \in \mathcal{L}^{\setminus\tau}(N \otimes O)$. Then, by the definition of weak language there exists a $\overrightarrow{\alpha\gamma/\beta\delta} \in \mathcal{L}\left(N \otimes O\right)$ such that $\widehat{\overrightarrow{\alpha\gamma/\beta\delta}} = w$. Then, we have that $\overrightarrow{\alpha/\beta} \in \mathcal{L}\left(N\right)$ and $\overrightarrow{\gamma/\delta} \in \mathcal{L}\left(O\right)$.

Now, since N, N', O and O' are reflexive 2-NFAs, with $N \approx_{\mathcal{L}} N'$ and $O \approx_{\mathcal{L}} O'$, by Lem. 6.12, we have:

1. $\overrightarrow{\alpha'/\beta'} \in \mathcal{L}\left(N\right) \cap \mathcal{L}\left(N'\right)$ with $\widehat{\overrightarrow{\alpha'/\beta'}} = \widehat{\overrightarrow{\alpha/\beta}}$, and
2. $\overrightarrow{\gamma'/\delta'} \in \mathcal{L}\left(O\right) \cap \mathcal{L}\left(O'\right)$ with $\widehat{\overrightarrow{\gamma'/\delta'}} = \widehat{\overrightarrow{\gamma/\delta}}$,

Now, since $\overrightarrow{\alpha'/\beta'} \in \mathcal{L}\left(N'\right)$ and $\overrightarrow{\gamma'/\delta'} \in \mathcal{L}\left(O'\right)$ we have that $\overrightarrow{\alpha'\gamma'/\beta'\delta'} \in \mathcal{L}\left(N' \otimes O'\right)$, and furthermore, $\widehat{\overrightarrow{\alpha'\gamma'/\beta'\delta'}} = \widehat{\overrightarrow{\alpha\gamma/\beta\delta}} = w \in \mathcal{L}^{\setminus\tau}\left(N' \otimes O'\right)$, as required. □

Now, before we can employ weak-language equivalence as a quotient in our compositional reachability algorithm (Algorithm 5.2), we must assure ourselves that the initial 2-NFA semantics are reflexive, and that reflexivity is preserved by composition:

Lemma 6.14 (2-NFA semantics of PNBs are always reflexive)**.**
For a marked PNB, N, its 2-NFA semantics, $\langle\!\langle N \rangle\!\rangle$ is reflexive.

Proof. For any marking of N, we can always fire the empty set of transitions, which has no effect on the boundaries. Thus, for every state of the 2-NFA, we have a self-loop with τ label, i.e. $\langle\!\langle N \rangle\!\rangle$ is reflexive. □

Lemma 6.15 (Reflexivity of 2-NFAs is preserved under composition)**.**
If N, a $(k,\, l)$-NFA, M a $(l,\, m)$-NFA, and O a $(m,\, n)$-NFA are reflexive then:

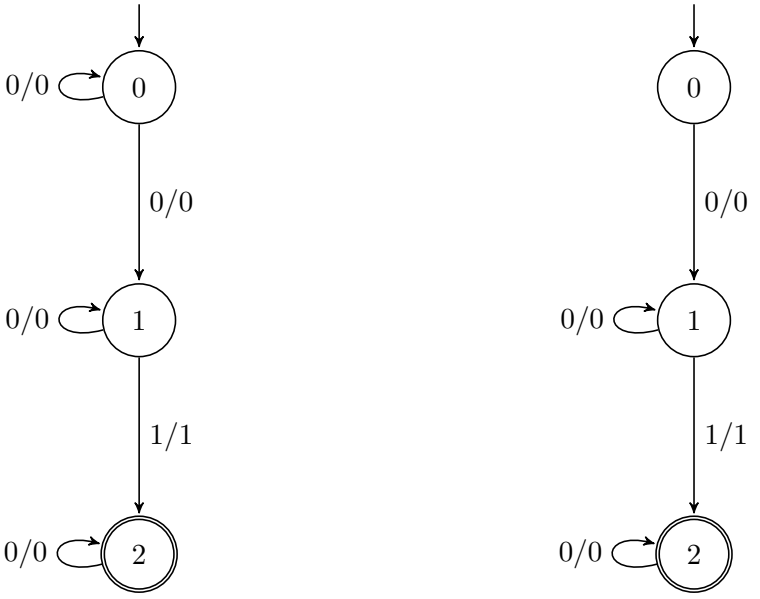
- *$N \,;\, M$ is reflexive*
- *$N \otimes O$ is reflexive*

Proof. The states of $N \,;\, M$ and $N \otimes O$ are pairs of states of the underlying 2-NFAs. Since the components are reflexive, in any state (pair of states), we can take a τ transition in both components, leading to a τ transition in the composite, as required. □

Now we have shown that we can arbitrarily replace reflexive 2-NFAs with (reflexive) weak-language equivalent 2-NFAs, and retain the weak language of the composite. At this point, it is worth noting that after quotienting by weak-language equivalence,

the states of a 2-NFA semantics will not necessarily correspond to markings of the underlying PNB; indeed, we only preserve the (weak) boundary protocol.

In fact, we can loosen our restriction requiring reflexive 2-NFAs, and instead require only that 2-NFAs are *strong language equivalent* to reflexive 2-NFAs, i.e. if we have: $N \approx_{\mathcal{L}} N' \sim_{\mathcal{L}} N''$ where N, N' are reflexive, then we should expect, for some compatible M, that $N \,;\, M \approx_{\mathcal{L}} N'' \,;\, M$. However, we might immediately notice that reflexivity is not necessarily preserved by (strong) language equivalence, as illustrated in Fig. 6.11.



(a) N, which is reflexive (b) N', such that $N' \sim_{\mathcal{L}} N$ but N' is not reflexive

Figure 6.11: Reflexivity is not preserved by language equivalence

However, this does not turn out to cause trouble; since strongly equivalent 2-NFAs can replicate language, they must contain an *equivalent* transition(s) to simulate the reflexivity of a reflexive 2-NFA. Indeed, we can prove that weak-language equivalence *up-to* strong-language equivalence is a congruence:

Theorem 6.16.
Suppose that:

- N, N', N'' *are* (k, l)*-NFA (*N*,* N' *reflexive),* $N \approx_{\mathcal{L}} N'$*, and* $N' \sim_{\mathcal{L}} N''$*,*
- M, M', M'' *are* (l, m)*-NFA (*M*,* M' *reflexive),* $M \approx_{\mathcal{L}} M'$*, and* $M' \sim_{\mathcal{L}} M''$*.*

Then, we have:

$$N \,;\, M \approx_{\mathcal{L}} N'' \,;\, M''$$

Proof. We have:

$$
\begin{aligned}
N \,;\, M &\approx_{\mathcal{L}} N' \,;\, M' && \text{by Thm. 6.13} \\
N' \,;\, M' &\sim_{\mathcal{L}} N'' \,;\, M'' && \text{by Thm. 6.1} \\
N' \,;\, M' &\approx_{\mathcal{L}} N'' \,;\, M'' && \text{by definition of } \approx_{\mathcal{L}} \\
N \,;\, M &\approx_{\mathcal{L}} N'' \,;\, M'' && \text{by transitivity of } \approx_{\mathcal{L}}
\end{aligned}
$$

□

Thus, we are justified in using standard language-equivalence preserving transformations to quotient τ-closed 2-NFA to improve performance.

Returning to an earlier example, consider again the τ-closed 2-NFA shown in Fig. 6.4 and observe that the language it accepts is simple, being given by the regular expression: $(0/0)^* (1/0 \mid 0/1) (0/0)^*$. Indeed, there is a smaller 2-NFA with the same language; in general, NFA minimisation techniques aim to find a structurally smaller (but not necessarily minimal) NFA that recognises the same language; an example of such a minimised 2-NFA is shown in Fig. 6.12. This 2-NFA recognises the weak boundary protocol containing only two traces: $\left\{\langle 0/1 \rangle, \langle 1/0 \rangle\right\}$; indeed, inspecting the original PNB (Fig. 6.1) we can see that (modulo internal firings) it can either perform a single $0/1$ or $1/0$ interaction to reach its desired marking.

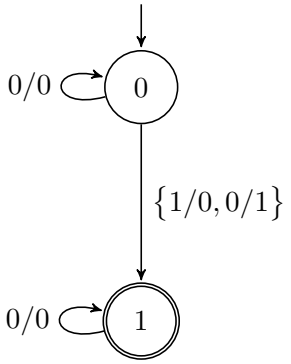


Figure 6.12: Minimised 2-NFA of Fig. 6.4

A final important point to note is that there is a single 2-NFA up-to weak-language, for a $N : (0, 0)$. Indeed, since there are no boundaries, all transitions of the 2-NFA will be τ-labelled. Therefore, if there is a (connected) accepting state, then the 2-NFA is weak-language equivalent to the 2-NFA with a single state self-loop labelled with τ, such that the state is initial, and accepting. If there is no such accepting state, the language is empty and thus is equivalent to that of the same single-state 2-NFA, but with no accepting state. These single-state 2-NFAs are illustrated in Fig. 6.13.

Figure 6.13: 2-NFAs that the semantics of any $N : (0, 0)$ are weak-language equivalent to

6.2 Memoisation, Associativity and Fixed Points

To improve the efficiency of our approach further, we can avoid unnecessary re-computation, by using *memoisation*. Indeed, there are two potential points at which memoisation can improve performance:

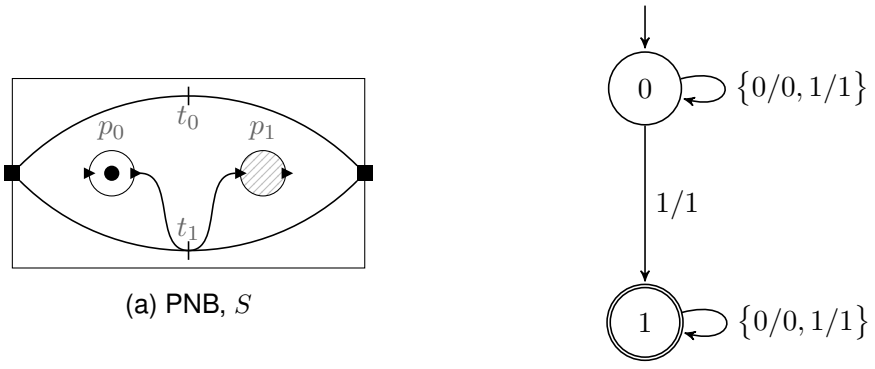
1. When computing the 2-NFA semantics of a arbitrary PNB expression, we may often encounter the same PNB component,
2. When composing 2-NFAs, we may compose "the same" 2-NFAs multiple times.

in both cases, each repeated component/composition only incurs the additional cost of checking for the existence of a pre-computed result. Since memoisation allows us to only compute a single representative of a language-equivalence class of 2-NFAs once, our approach is somewhat similar to symmetry-reduction (see §1.5).

Since weak-language equivalence is a congruence, we can freely substitute 2-NFAs if their weak-language is respected. We say that a 2-NFA is weak-language reduced if it has been structurally reduced while preserving weak-language. Indeed, exploiting case 1 is simple: we maintain a mapping from PNB components to their (weak language reduced) 2-NFA semantics; upon encountering a PNB component, we simply look for its presence in the mapping, returning the previously computed 2-NFA or computing (and reducing) the 2-NFA and updating the mapping, as appropriate. Case 2 leads us to maintain a mapping from (weak language reduced) pairs of 2-NFA to their (weak language reduced) composition. When encountering a composition, we check for membership in the mapping *up-to* weak-language, returning the reduced 2-NFA or computing the reduced composition and updating the mapping as necessary.

As an example of the utility of memoisation, consider the somewhat contrived PNB, S, and its 2-NFA semantics, $\langle\!\langle S \rangle\!\rangle$, illustrated in Fig. 6.14. This PNB can reach its target marking by a single firing of t_1, interacting on both boundaries; in both its initial and target marking the PNB is able to fire the empty set of transitions and also fire t_0, which only connects to the two boundary ports.

Now, consider a sequence of k such PNBs that are synchronously composed; in traversing this sequence from left to right to perform the required compositions, we encounter



(a) PNB, S

(b) 2-NFA, $\langle\!\langle S \rangle\!\rangle$

Figure 6.14: Example PNB and its 2-NFA semantics

k S PNBs. Due to memoisation, we only need compute and reduce S's 2-NFA semantics, $\langle\!\langle S \rangle\!\rangle$, once, obtaining the 2-NFA, M, shown in Fig. 6.15; thus, a mapping $S \mapsto M$ is added. On each subsequent S component we simply lookup this new entry in the mapping.

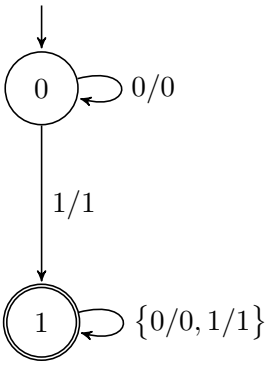


Figure 6.15: M, (weak) language-equivalent to the 2-NFA in Fig. 6.14b

Now, consider the first composition; at this stage the mapping from compositions to 2-NFAs is empty, so we must perform the composition $M \ ; \ M$, obtaining the 2-NFA, O, illustrated in Fig. 6.16, thus, we store the mapping $(M \ ; \ M) \mapsto O$. On the next (and each subsequent) composition, we have $O \ ; \ M$; the only mapping we have is from $M \ ; \ M$, however, we lookup entries up-to (weak) language-equivalence, and $M \sim_{\mathcal{L}} O$, thus we are able to match the mapping's entry and can simply return O rather than perform any further compositions. To summarise:

- We have a single PNB to 2-NFA mapping: $S \mapsto M$,
- We have a single 2-NFA-composition to 2-NFA mapping: $(M \ ; \ M) \mapsto O$,
- The 2-NFA semantics of *any* k S nets in a sequence is O; when determining this, we perform a single PNB to 2-NFA conversion, with $k - 1$ lookups and a single 2-NFA composition with $k - 2$ lookups.

Intuitively, the behaviour of such a sequence of S nets is that each component must perform one firing of t_1 to reach its desired marking, and in doing so interact on both boundaries. Indeed, each component can either fire t_1 at the same time, or can simply "pass the interaction" along the sequence by firing t_0. Any components remaining that need to fire t_1 can do so, as encoded by the $1/1$ self-loop in the accepting state of M.

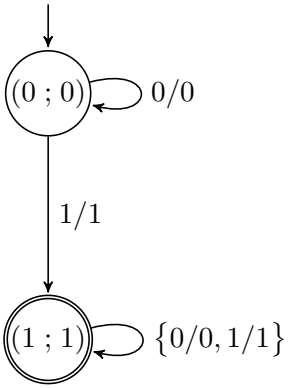


Figure 6.16: 2-NFA $O = M \; ; M$

6.2.1 Behavioural fixed-points

The previous example demonstrated an important feature of certain PNB expressions, that of reaching a *fixed-point* w.r.t. 2-NFA semantics:

Definition 6.17 (Left-composition fixed-point of 2-NFA semantics)**.**
For a (k, l)-NFA, N, and (l, m)-NFA, M, we say that there is a left-composition, n-step fixed-point, if $N \; ; O_n \approx_{\mathcal{L}} O_n$, where:

$$O_0 \stackrel{\text{def}}{=} M$$
$$O_{i+1} \stackrel{\text{def}}{=} N \; ; O_i$$

we refer to O_n as the fixed-point.

Definition 6.18 (Right-composition fixed-point of 2-NFA semantics)**.**
For a (k, l)-NFA, M, and (l, m)-NFA, N, we say that we say that there is a right-composition, n-step fixed-point, if $O_n \; ; N \approx_{\mathcal{L}} O_n$, where:

$$O_0 \stackrel{\text{def}}{=} M$$
$$O_{i+1} \stackrel{\text{def}}{=} O_i \; ; N$$

again, we refer to O_n as the fixed-point.

In other words, after composing M with N some number of times, composing again with N has no effect on the weak language of the resulting 2-NFA.

In our previous example, we reached a left-composition fixed-point of S (Fig. 6.14a) composed with itself, after 0 steps, with the simple fixed-point illustrated in Fig. 6.16. However, fixed-points need not be trivial, for example the sequence of PHILO-with-FORK components in the DPH$(-)$ system reaches a fixed-point, which is illustrated in Fig. 6.17, at 2; indeed, we have:

$$\forall k \geq 2.\ (\text{PHILO}\ ;\ \text{FORK})^k \approx_{\mathcal{L}} (\text{PHILO}\ ;\ \text{FORK})^{k+1}$$

In fact, a independent observation of the fixed-point behaviour of the dining philosophers was made in the setting of Span(Graph) [154].

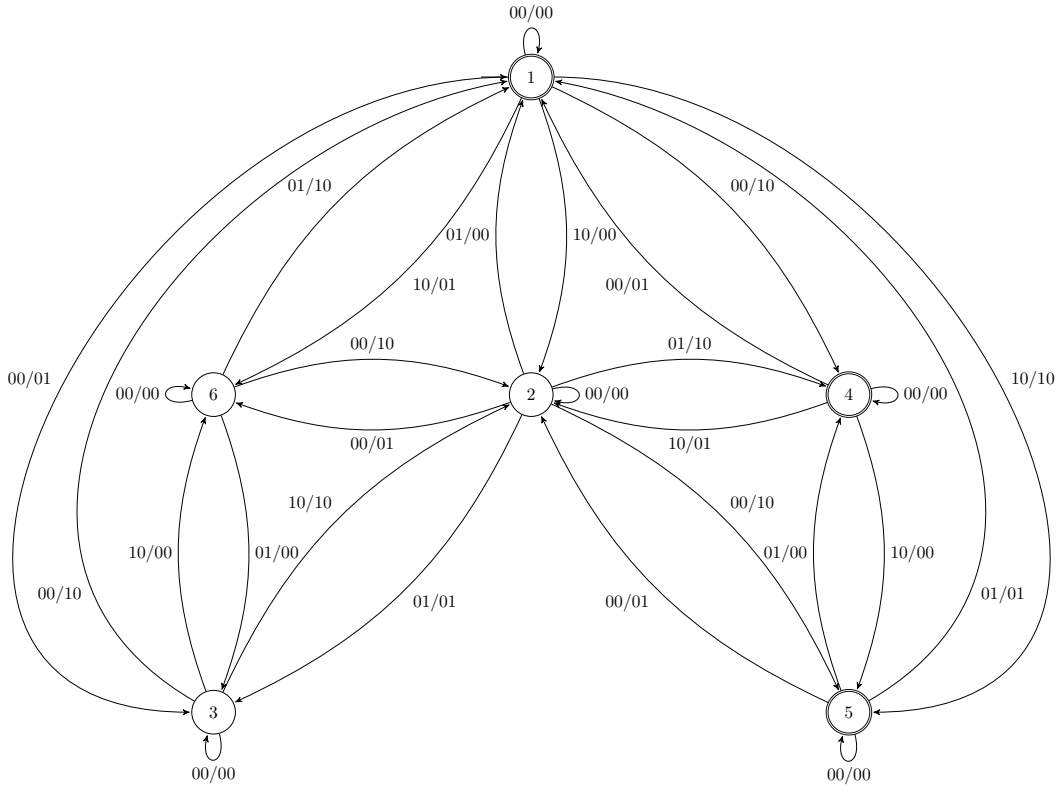
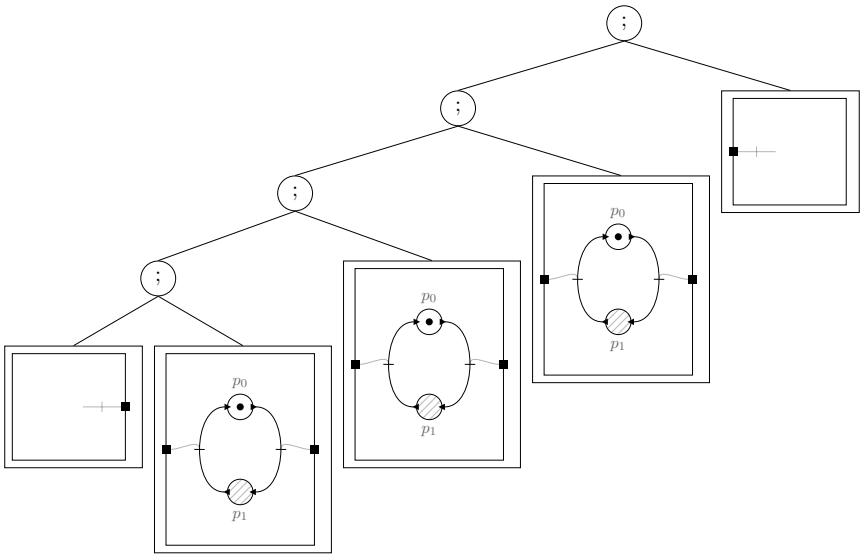


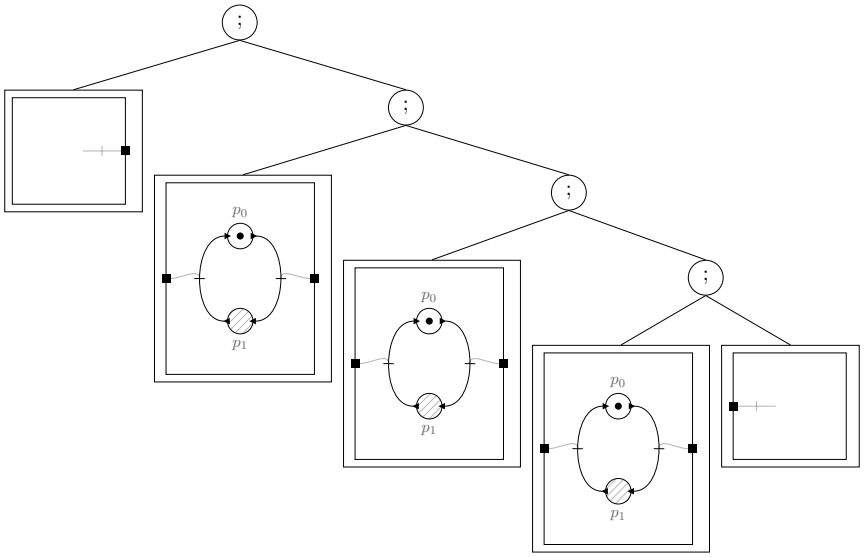
Figure 6.17: Fixed-point of $(\text{PHILO}\ ;\ \text{FORK})^k$, reached at $k = 2$

The performance of our technique is vastly improved when a system reaches a fixed-point — rather than perform additional conversion/compositions, we simply look up and return previously-computed results; additional repetitions become essentially free. However, an interesting property is that not all isomorphic PNB expressions (i.e. when evaluated to a single PNB) will reach fixed-points: indeed, even re-associating an expression can cause a fixed-point to not be reached.

We return once more to the BUFFER(3) system, to illustrate this property of PNB expressions. Consider the two expression trees shown in Fig. 6.18; indeed we can trivially transform one into the other by re-associating the ';' nodes.



(a) Left associative



(b) Right associative

Figure 6.18: Left vs Right associativity of ';' in BUFFER(3) Expression tree

Now, consider a depth-first traversal of the expression trees, performing conversion to 2-NFA, before applying τ-closure and then minimisation. We highlight the intermediate 2-NFAs obtained at each step (i.e. after each composition) in Fig. 6.19 for the left-associative case and Fig. 6.20 for the right-associative case. Indeed, the end result is guaranteed to be equal, since, as a direct corollary of Prop. 3.33(ii), composition of 2-NFAs is associative. However, the intermediate 2-NFAs are vastly different.

As is readily seen in Fig. 6.20, when processing the right-associative expression, we reach a fixed-point after 0 steps (indeed, BUFFER ; $\bot \approx_{\mathcal{L}} \bot$). On the other hand, processing the left-associative expression *does not* reach a fixed-point. Not (quickly) reaching a fixed-point leads to poor performance - at each composition step we check

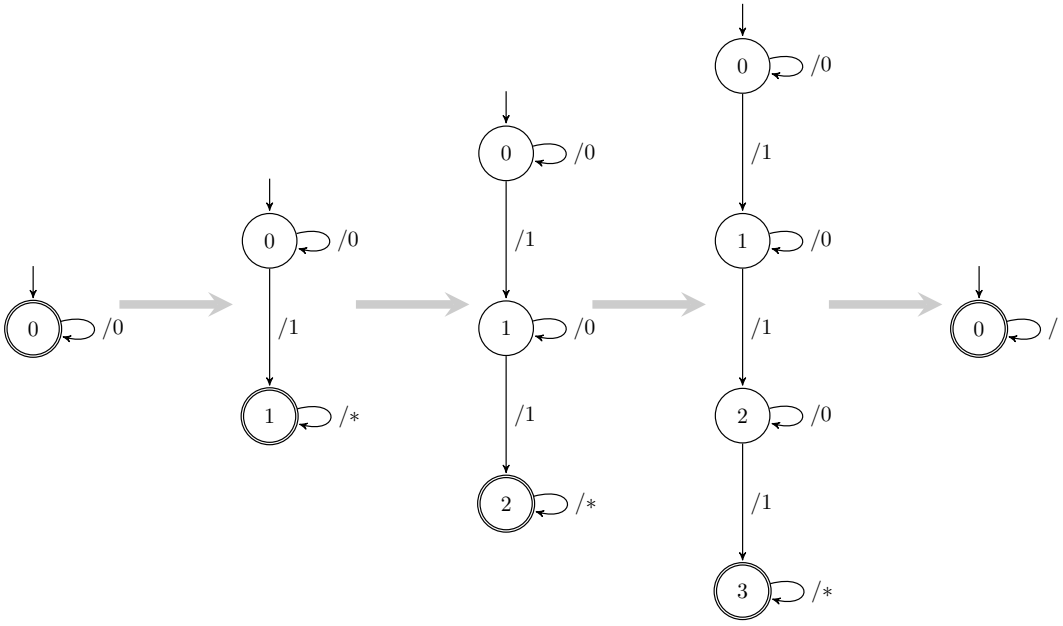


Figure 6.19: Intermediate 2-NFAs encountered when converting the expression of Fig. 6.18a

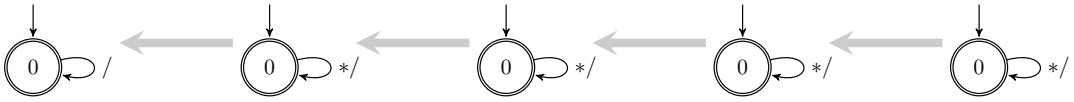


Figure 6.20: Intermediate 2-NFAs encountered when converting the expression of Fig. 6.18b

for weak-language equivalence with *every* previous composition, since the cache monotonically increases in size. Indeed, one might think that expiring infrequently-used cache entries (e.g. using a LRU cache algorithm, commonly found in operating system memory managers) may alleviate the problem, and it does to an extent, but a greater overhead is that not only is the number of entries cache growing, but the *size* of the cache entries is growing, leading to increasing time to check for membership.

The conclusion we draw is that while it is *semantically* unimportant which way an expression's compositions are associated, when using the optimisation techniques outlined in this chapter, different composition associations may lead to vastly different *performance*. Unfortunately, this leads us to conclude that we must carefully consider the *form* of expressions that specify systems. Indeed, it is not clear that we can determine a priori that changing associations in an expression will improve performance (or not!).

6.2.2 Quadratic firing sequence $\rightarrow$ linear reachability check

Consider again the composite BUFFER$(-)$ net, e.g. we illustrate BUFFER(4) in Fig. 6.21.

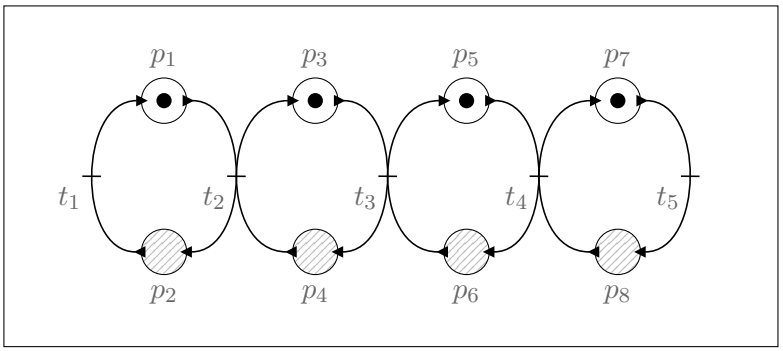


Figure 6.21: BUFFER(4)

Now, the minimal length of a firing sequence required to reach the target marking in BUFFER(k) is quadratic in k; precisely, it is the k^{th} *triangle number*, T_k, with:

$$T_1 \stackrel{\text{def}}{=} 1$$
$$T_{i+1} = i + T_i$$

Indeed, for BUFFER(4) to reach its target marking, t_i must appear $i-1$ times in the firing sequence.

Thus the naive monolithic approach of simply firing enabled transitions until reaching the target marking has to fire a quadratic number of transitions. Improving slightly, using the compositional technique *without* optimisation, each of the k single-buffer components is converted to their 2-NFA semantics, before composing the 2-NFAs. Without reduction w.r.t. weak language, the intermediate 2-NFAs quickly grow large (the final 2-NFA has 2^k states, since each marking is reachable). However, with a right-associated composition, as we have shown, we reach a fixed-point after 0 steps, thus, the only computation we must perform is:

1. Convert the three distinct components to their 2-NFA semantics,
2. Perform one composition of the 2-NFAs corresponding to BUFFER and $\bot$,
3. For each of the remaining $k-1$ BUFFER compositions, lookup the cached composition,
4. Finally, perform a last composition with the 2-NFA of $\top$.

Thus, by a naive cost approximation, we have reduced an inherent quadratic complexity to *linear* complexity, assuming a linear cost for conversions, lookup and compositions.

6.3 Reassociated Examples

Due to the performance penalty due to non-associativity, mentioned in the previous section, we specify some of our benchmarks in a "lower-level" style, to obtain the associativity we require. In each case, the alternative expression leads to right-associated compositions, rather than left-associated compositions.

BUFFER(k)(§4.1.1):

$$\textbf{old: } \textsc{buffer}(k) \stackrel{\text{def}}{=} \top \,;\, \textsc{buffer}^k \,;\, \bot$$
$$\textbf{new: } \textsc{buffer}(k) \stackrel{\text{def}}{=} \top \,;\, \textbf{fold}_{\mathbb{N}}\ k\ \bot\ \left(\boldsymbol{\lambda} x : \mathrm{Net}\langle 1,\, 0\rangle \,.\, \textsc{buffer} \,;\, x\right)$$

ITER-CHOICE(k)(§4.2.3):

$$\textbf{old: } \textsc{iter-choice}(k) \stackrel{\text{def}}{=} \textsc{addtok} \,;\, \textbf{n\_sequence}\ k\ \textsc{choice} \,;\, \bot$$
$$\textbf{new: } \textsc{iter-choice}(k) \stackrel{\text{def}}{=} \textsc{addtok} \,;\, \textbf{fold}_{\mathbb{N}}\ k\ \bot\ \left(\boldsymbol{\lambda} x : \mathrm{Net}\langle 1,\, 0\rangle \,.\, \textsc{choice} \,;\, x\right)$$

OVERTAKE(k) (§4.2.1):

$$\begin{aligned}
\textbf{old: } \textsc{overtake}(k) \stackrel{\text{def}}{=}\ & \textbf{bind } iLock = \textsc{rLockInterface} \,;\, \textsc{lock} \,;\, \textsc{fLockInterface} \textbf{ in} \\
& \textbf{bind } lockCar = iLock \,;\, \textsc{car} \textbf{ in} \\
& \uparrow_4 \,;\, \textbf{n_sequence}\ k\ lockCar \,;\, iLock \,;\, \downarrow_4 \\
\textbf{new: } \textsc{overtake}(k) \stackrel{\text{def}}{=}\ & \textbf{bind } iLock = \textsc{rLockInterface} \,;\, \textsc{lock} \,;\, \textsc{fLockInterface} \textbf{ in} \\
& \textbf{bind } lockCar = iLock \,;\, \textsc{car} \textbf{ in} \\
& \uparrow_4 \,;\, \textbf{fold}_{\mathbb{N}}\ k\ \left(iLock \,;\, \downarrow_4\right)\ \left(\boldsymbol{\lambda} x : \mathrm{Net}\langle 4,\, 0\rangle \,.\, lockCar \,;\, x\right)
\end{aligned}$$

6.4 Optimised Algorithm

We now introduce an improved version of Algorithm 5.2, which incorporates the optimisations introduced in this chapter. We demonstrate the encouraging performance of this algorithm, by evaluating it on the examples of Chapter 4 and §6.3.

The optimised algorithm is presented as Algorithm 6.1; observe that 2-NFAs are reduced up-to weak-language equivalence (lines 6 and 16) and the memoisation map is checked for membership up-to weak-language (since all 2-NFAs are reduced w.r.t. weak-language we simply check for language equivalence at (Line 13)).

Algorithm 6.1 Optimised Algorithm

Require: `knownNetNFAs`, `knownNFAComps` initially empty
1: **procedure** TONFA(t)
2: **if** t is a PNB **then**
3: **if** CONTAINS(`knownNetNFAs`, t) **then**
4: **return** `knownNetNFAs`$[t]$
5: **else**
6: $n \leftarrow$ REDUCE(τ-CLOSE(NETTONFA(t)))
7: `knownNetNFAs`$[t] := n$
8: **return** n
9: **end if**
10: **else** ▷ t is $(t_1, t_2, \texttt{OP})$, where `OP` is ';' or '$\otimes$'
11: $n_1 \leftarrow$ TONFA(t_1)
12: $n_2 \leftarrow$ TONFA(t_2)
13: **if** CONTAINSEQUIV(`knownNFAComps`, $(n_1, n_2, \texttt{OP})$) **then** ▷ Up-to $\approx_{\mathcal{L}}$
14: **return** `knownNFAComps`$[(n_1, n_2, \texttt{OP})]$
15: **else**
16: $n \leftarrow$ REDUCE(τ-CLOSE(n_1 OP n_2))
17: `knownNFAComps`$[(n_1, n_2, \texttt{OP})] := n$
18: **return** n
19: **end if**
20: **end if**
21: **end procedure**

We can now evaluate this algorithm on the examples of Chapter 4 and §6.3, that is, the original benchmark examples, except those with refactored versions using more-performant composition associatations. Indeed, we present the new timing and memory requirements in Table 6.1.

Inspecting the values in Table 6.1 and contrasting with those in Table 5.1, several points are clear:

- For the majority of examples, the performance has *vastly* improved; indeed, for several systems (e.g. OVERTAKE$(-)$), the time taken by the improved algorithm for parameter 32768 is less than the naive algorithm for paremeter 1.
- Indeed, we can observe the effects of reaching fixed points — for most systems (e.g. BUFFER$(-)$), the time/memory use is essentially constant[2]
- The performance is *not* directly proportional to net size: indeed, OVERTAKE$(-)$ is a "large" system, with scalable performance, whereas COUNTER$(-)$ is a "small" system, with poor performance.

To help interpreting our results, we have plotted the data in Fig. 6.22.

[2]Our tool is written in Haskell, a language that uses garbage collection for automatic memory management, which may explain some of the slight variance in memory usage.

Table 6.1: Checking reachability of markings in Fig. 5.14, on examples of §6.3, using Algorithm 6.1. Key: T = Time (s), M = Maximum Resident Memory (MB), *TO* = Time Out (300s)

Sys	T	M	Sys	T	M
OVERTAKE(2)	0.017	21.02	$T_\wedge(2,2)$	0.001	15.22
OVERTAKE(8)	0.017	21.02	$T_\wedge(8,8)$	0.001	15.25
OVERTAKE(32)	0.017	21.02	$T_\wedge(32,32)$	0.002	15.23
OVERTAKE(512)	0.017	21.06	$T_\wedge(128,128)$	0.002	15.26
OVERTAKE(4096)	0.017	21.10	$T_\wedge(512,512)$	0.011	15.33
OVERTAKE(32768)	0.019	22.65	$T_\wedge(2048,2048)$	1.069	20.62
DAC(2)	0.001	15.18	HARTSTONE(2)	0.013	19.33
DAC(8)	0.001	15.26	HARTSTONE(4)	0.039	19.61
DAC(32)	0.001	15.20	HARTSTONE(8)	2.013	21.41
DAC(512)	0.001	15.23	HARTSTONE(10)	4.016	20.62
DAC(4096)	0.002	15.81	HARTSTONE(13)	10.445	25.74
DAC(32768)	0.003	22.18	HARTSTONE(16)	25.252	28.65
DPH(2)	0.008	20.02	TOKENRING(2)	0.008	17.99
DPH(8)	0.011	20.64	TOKENRING(4)	0.035	19.79
DPH(32)	0.011	20.64	TOKENRING(8)	2.043	21.33
DPH(512)	0.011	20.64	TOKENRING(10)	4.072	21.11
DPH(4096)	0.011	19.25	TOKENRING(13)	10.026	25.22
DPH(32768)	0.012	22.61	TOKENRING(16)	19.247	28.65
BUFFER(2)	0.001	14.19	CYCLIC(2)	0.005	16.81
BUFFER(8)	0.001	14.24	CYCLIC(4)	0.017	19.52
BUFFER(32)	0.002	14.22	CYCLIC(8)	0.094	20.57
BUFFER(512)	0.001	14.38	CYCLIC(10)	1.077	20.59
BUFFER(4096)	0.002	14.90	CYCLIC(13)	3.075	22.24
BUFFER(32768)	0.007	21.30	CYCLIC(16)	6.091	24.60
REPLICATORS(2)	0.002	15.22	COUNTER(2)	0.004	16.09
REPLICATORS(8)	0.001	15.23	COUNTER(4)	0.011	20.56
REPLICATORS(32)	0.001	15.22	COUNTER(8)	0.087	21.36
REPLICATORS(512)	0.001	15.31	COUNTER(10)	1.075	22.75
REPLICATORS(4096)	0.002	15.81	COUNTER(13)	4.035	28.59
REPLICATORS(32768)	0.004	22.31	COUNTER(16)	8.649	28.60
ITER-CHOICE(2)	0.002	15.23	$T_\vee(1,1)$	0.002	15.60
ITER-CHOICE(8)	0.002	15.25	$T_\vee(2,2)$	0.004	15.63
ITER-CHOICE(32)	0.002	15.24	$T_\vee(3,3)$	12.455	32.45
ITER-CHOICE(512)	0.002	15.30	$T_\vee(4,4)$	*TO*	*TO*
ITER-CHOICE(4096)	0.002	15.86	$T_\vee(5,5)$	*TO*	*TO*
ITER-CHOICE(32768)	0.004	22.32	$T_\vee(6,6)$	*TO*	*TO*

6.5 Poorly performing Examples

An immediate problem with examples that do not reach fixed points is the monotonic size increase of the 2-NFA-composition cache. One might imagine that expiring old cache entries (for example, using a least-recently used (LRU) strategy) would improve

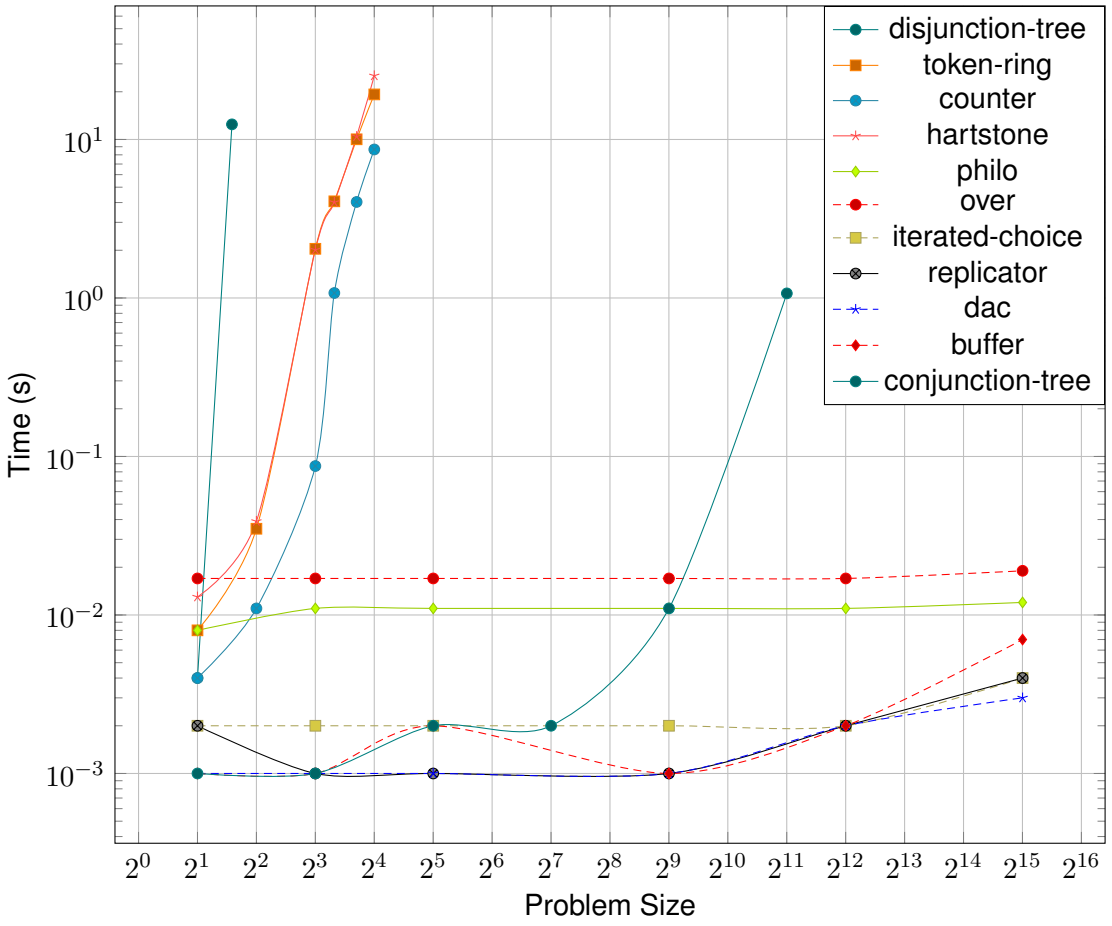


Figure 6.22: Time vs Problem size for Algorithm 6.1

performance; however, if there is also a monotonic increase in the *size* of the 2-NFAs being composed, the cost of checking equivalence and minimisation will dominate.

Indeed, growth of 2-NFAs does not have to be monotonic. In several examples, the growth is "intermediate" in the sense that large 2-NFAs are generated, only to be quotiented later. For example, in the TOKENRING$(-)$ system (§4.1.2), intermediate growth is caused by the sequence of WORKER components being able to receive multiple tokens. However, when composing the WORKERs components with the INJECTOR, which can only input a single token into the system, the behaviour is quotiented, vastly reducing the size of the resulting 2-NFA.

Another example of a system exhibiting intermediate growth is the $T_{\vee}(k, l)$ system, with depth k and width l (§4.1.4); consider the target marking that places a token in each leaf place, having started with only a single token in the root place. Intuitively, such a target marking is not reachable: at each internal node of the tree, we can only pass a token to a single sub-tree, and thus we can't pass tokens down to every leaf from the root. However, until the root node is composed with the rest of the tree, the behaviour of the sub-tree is not constrained to only receive a single token and thus has behaviour that allows it to reach its target marking by receiving a token for each leaf component.

Indeed, the $T_V(3,3)$ system, illustrated in Fig. 6.23, has a component-wise specification (given in §4.1.4) that builds the composite net by recursively building larger complete sub-trees. As an detailed example, let us consider the 2-NFAs obtained during such a procedure.

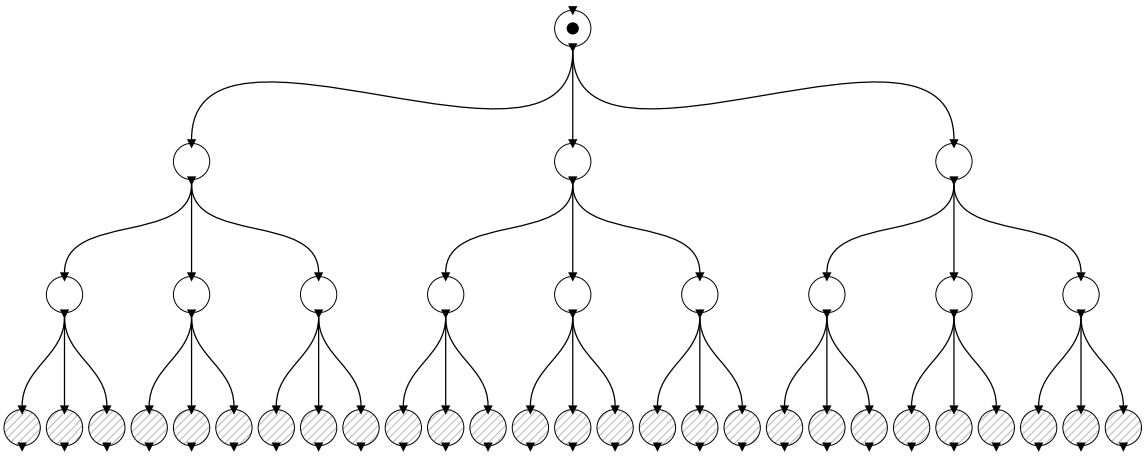


Figure 6.23: $T_V(3,3)$

The leaf sub-tree is illustrated in Fig. 6.24; first, the component PNBs are converted to their 2-NFA semantics, as per Fig. 6.25. Then, 2-NFA compositions are performed, with reduction up to weak-language being performed after each composition. The first composition leads to the 2-NFA[3] that is not in Fig. 6.26; notice that the leaf components can reach their target marking "in either order" since tokens can either be consumed by a leaf, or passed to the next. Indeed, it is this lack of order that leads to a blowup in 2-NFA size. Therefore, a simpler, language-equivalent 2-NFA can be constructed, which disregards the internal state of the composition.

The reduced 2-NFA for two leaf components is then composed with the 2-NFA of a single leaf component, leading to the 2-NFA illustrated in Fig. 6.27. Again, the 2-NFA can be simplified, as shown.

The final composition for the leaf sub-tree composes the reduced three-leaf 2-NFA with the 2-NFA of the terminator, leading to the 2-NFA shown in Fig. 6.28, which is the 2-NFA representing the PNB-expression sub-tree of three leaf components.

The deepest internal node sub-tree has the structure shown in Fig. 6.29; `leaves` is the sub-tree of Fig. 6.24. Again the component PNBs are converted to their 2-NFA semantics, shown in Fig. 6.30. The compositions lead to intermediate 2-NFAs shown in Fig. 6.31 and Fig. 6.32. Indeed this 2-NFA shows that the boundary protocol of the deepest internal-node sub-tree simple requires three tokens to be inserted to the sub-tree (and internally, passed to the leaves), while passing additional tokens to the sub-tree's next sibling.

[3]Here, we use α to stand for either 0 or 1 — N.B. that $\alpha/\alpha = \{0/0, 1/1\} \neq \{0/0, 0/1, 1/0, 1/1\} = */*$.

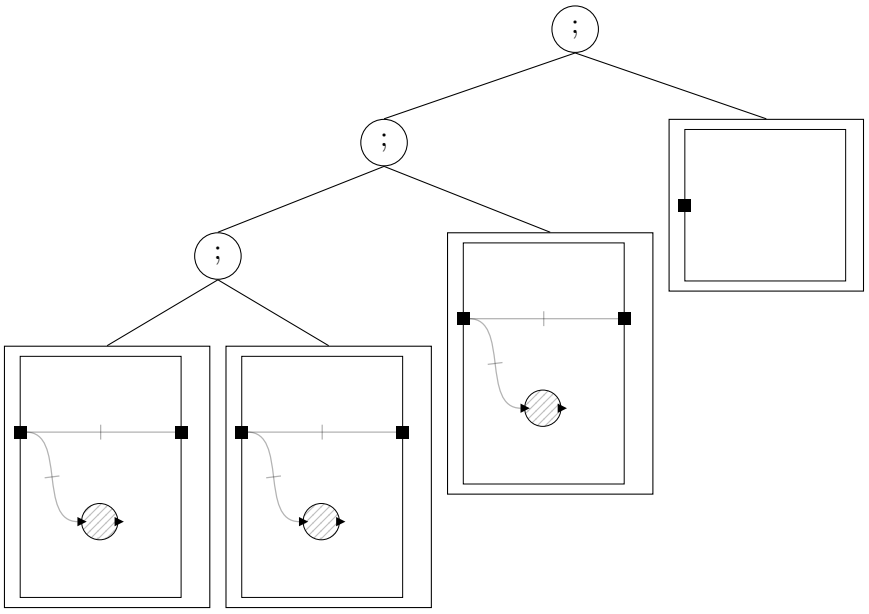


Figure 6.24: Leaves sub-tree of $T_V(3,3)$

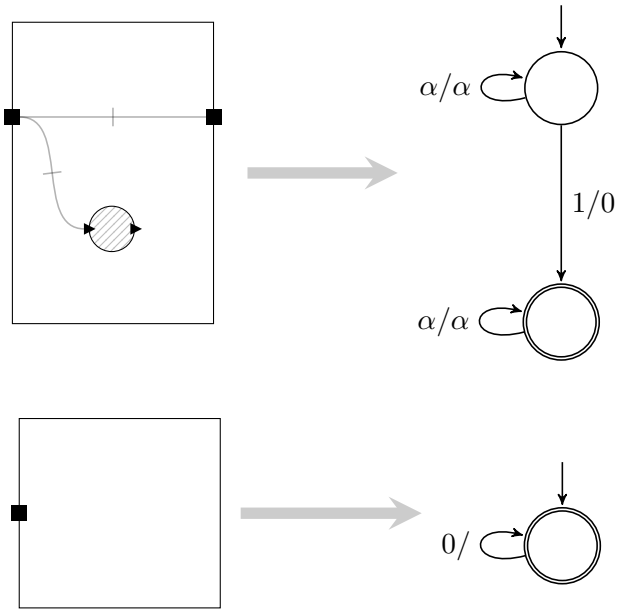


Figure 6.25: 2-NFA semantics of leaf components

Three deepest internal-node sub-trees are composed, as per Fig. 6.33, where `dins` is the sub-tree shown in Fig. 6.29.

Indeed, the reduced 2-NFA shown in Fig. 6.32 has a repeated structure similar to that of a single leaf component, albeit requiring 3 transitions to reach the accepting state. Thus, one would expect that composing two such 2-NFAs in order to perform the compositions of Fig. 6.33 will lead to similar blow up in state space.

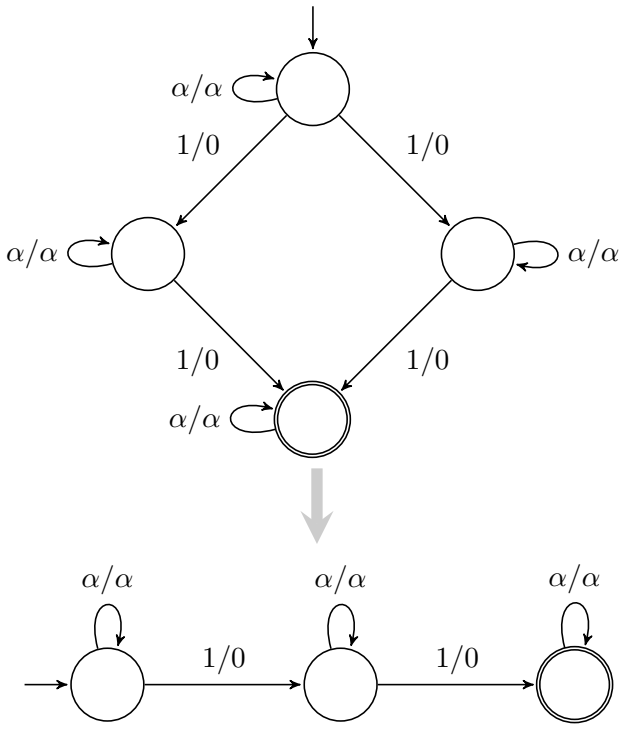


Figure 6.26: 2-NFA composition of two leaf components

Indeed, this is the case: with the first composition leading to the 2-NFA illustrated in Fig. 6.34; again, all possible orderings of the sub-trees reaching their target markings are represented in the composite 2-NFA.

Composing the reduced 2-NFA of Fig. 6.34 with another internal-node 2-NFA leads to the 2-NFA shown in Fig. 6.35. Indeed, such a 2-NFA represents the semantics of the sub-tree containing three internal nodes (and thus, 9 leaf components), as can be witnessed in the language of the reduced 2-NFA.

The preceding pattern of large composite 2-NFAs being generated, that are later reduced, repeats for the final internal-node level of $T_V(3,3)$, with the final composition leading to a (unminimised) 2-NFA of 190 nodes. We show the intermediate 2-NFA sizes in Table. 6.2: for any sub tree containing k leaf components the minimised 2-NFA semantics will have $k+1$ states. Only when composing with the root, does the 2-NFA drastically in size: since the root can only perform a single $/1$-labelled transition, thus the composite 2-NFA only has 2 states, neither of which are accepting, as illustrated in Fig. 6.36. Therefore, the reduced final 2-NFA only has a single non-accepting state, indicating the target marking is not reachable.

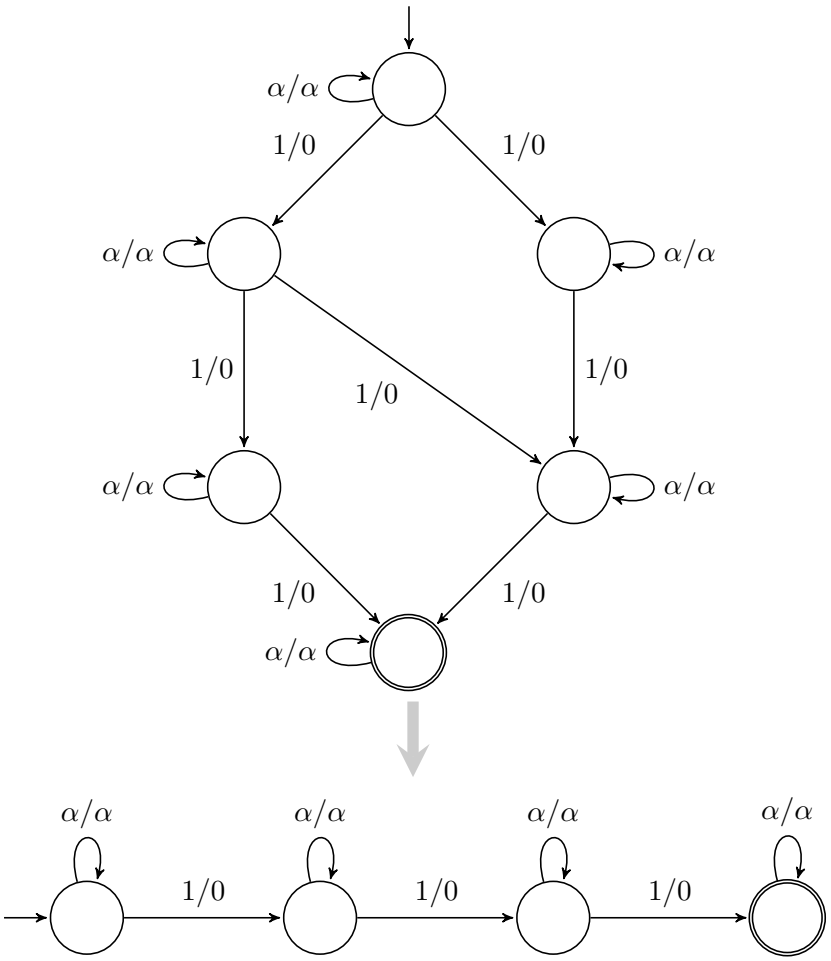


Figure 6.27: 2-NFA composition of three leaf components

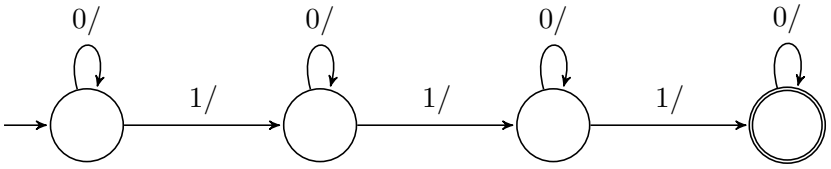


Figure 6.28: 2-NFA semantics of a leaves sub-tree

6.6 Summary

In this chapter, we have introduced and evaluated improvements to our algorithm for compositionally checking reachability in PNB systems, taking advantage of weak-language equivalence and the fixed-points that can be exploited by memoisation. In summary, our approach embodies a simple divide-and-conquer and memoisation strategy, i.e. a bottom-up *dynamic programming* approach. The component-wise specification is exploited to avoid re-computing local reachability 2-NFAs when particular components appear more than once, while intermediate 2-NFAs are minimised to reduce composition costs, possible since 2-NFA (weak) language-equivalence is a congruence w.r.t.

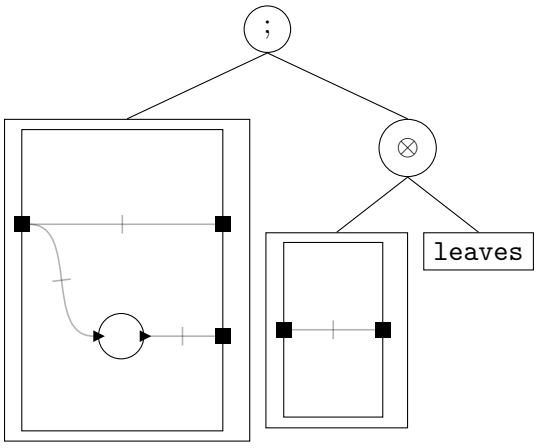


Figure 6.29: Deepest internal-node sub-tree structure

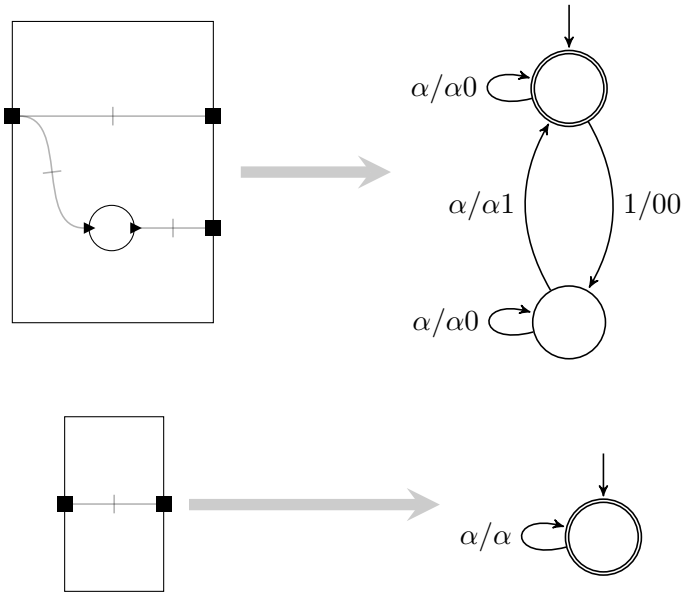


Figure 6.30: 2-NFA semantics of internal components

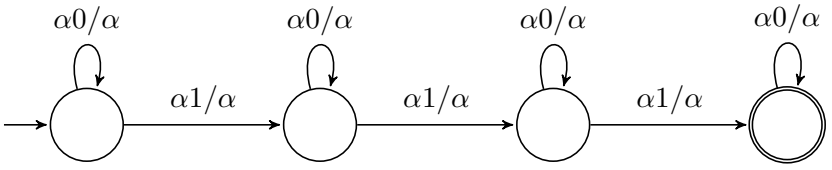


Figure 6.31: 2-NFA of tensor sub-tree shown from Fig. 6.29

the composition operations.

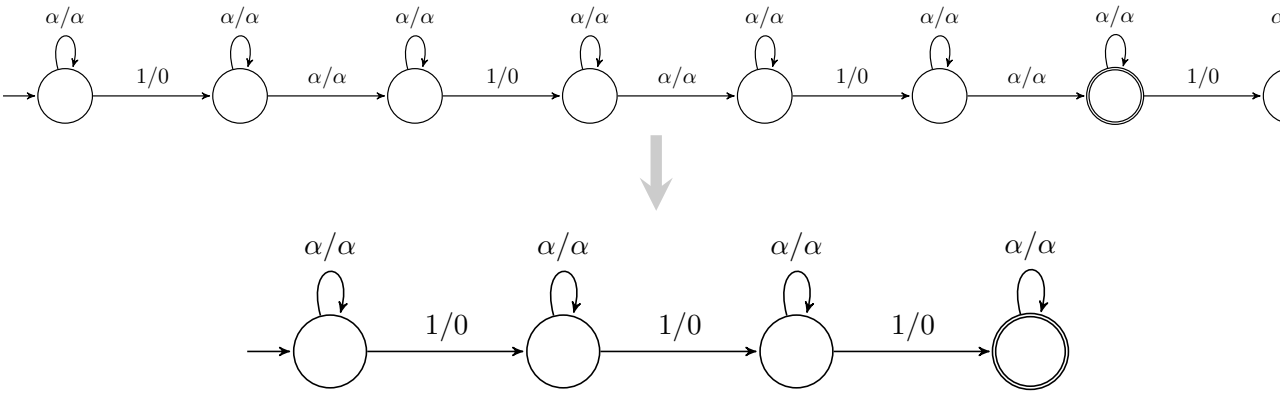


Figure 6.32: 2-NFA semantics of deepest internal-node sub-tree

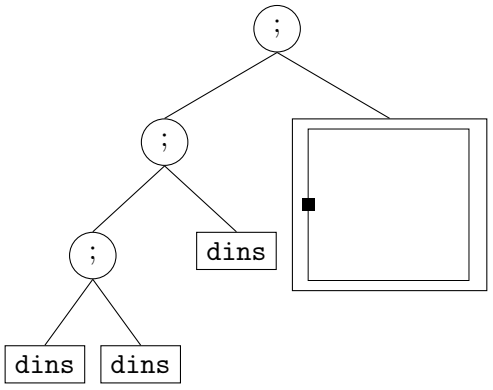


Figure 6.33: Composition of deepest internal node sub-trees

Product Size					Minimised NFA size	Sub-tree level
2	×	2	=	**4**	3	Leaf level
3	×	2	=	**6**	4	
4	×	4	=	**16**	7	1st Node Level
7	×	4	=	**28**	10	
10	×	10	=	**100**	19	2nd Node Level
19	×	10	=	**190**	28	
28	×	2	=	**2**	1	Root Level

Table 6.2: Intermediate Product Sizes for $T_{\mathrm{V}}(3,3)$

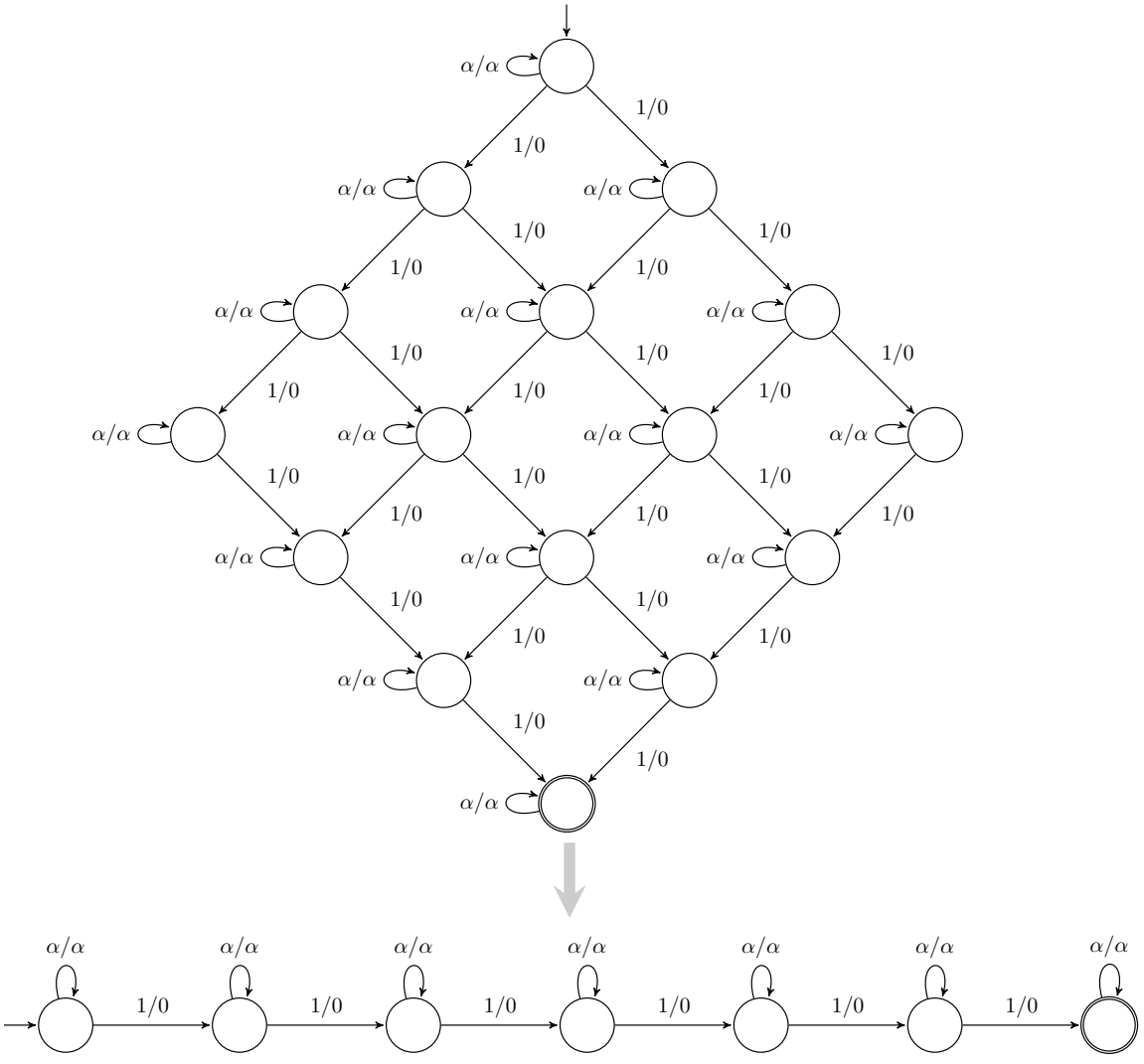


Figure 6.34: 2-NFA semantics of deepest internal-node sub-tree composition

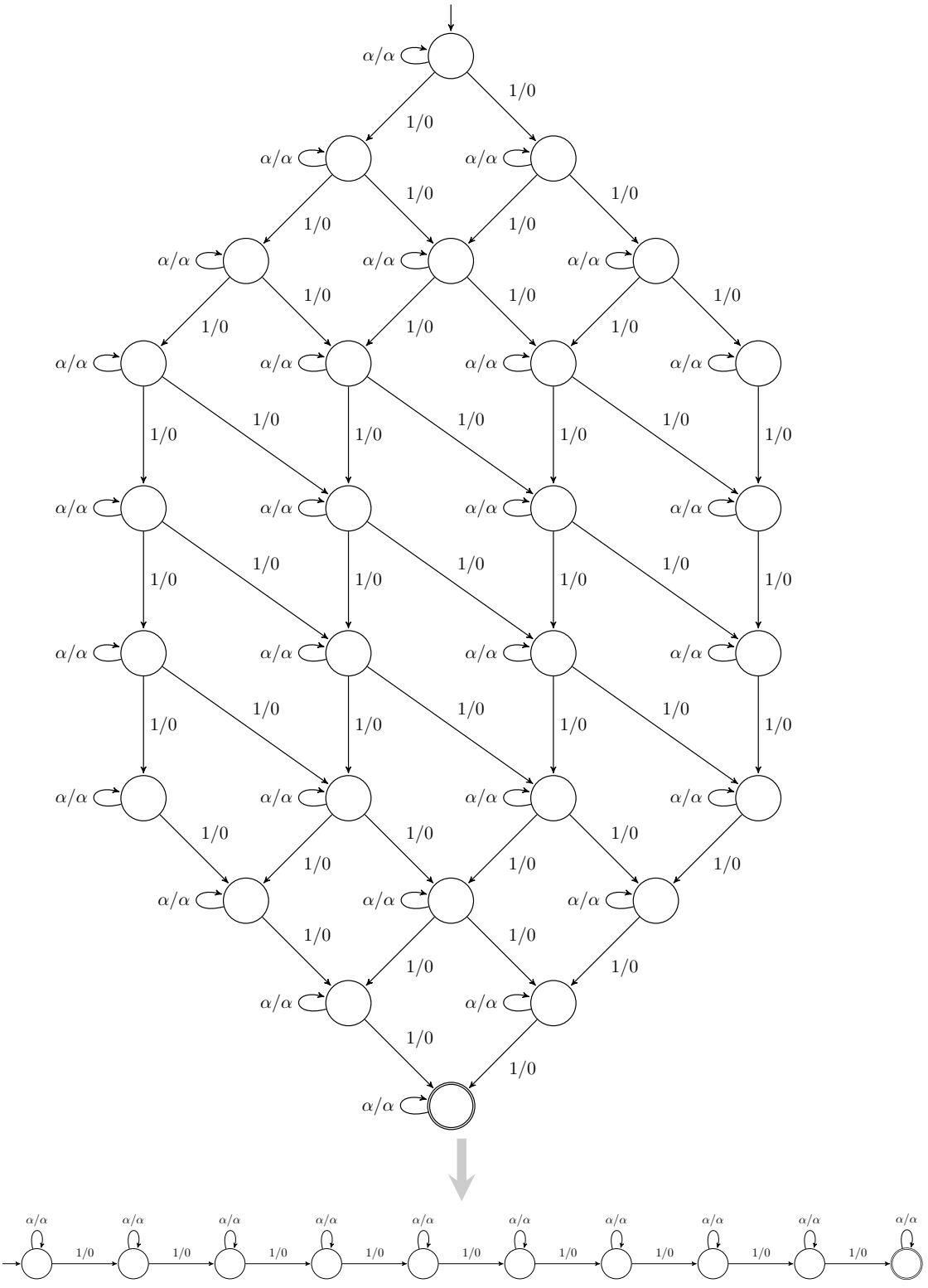


Figure 6.35: 2-NFA semantics of three composed internal-node sub-trees

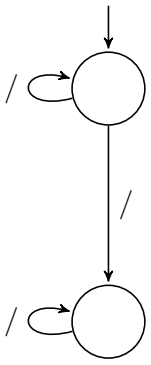


Figure 6.36: 2-NFA after composing with the root component

Chapter 7

Implementation and Comparison

In Chapters 5 and 6, we introduced a *compositional* approach to checking reachability in systems specified using PNBml. Exploiting *weak language equivalence*, we demonstrated effective statespace reductions, whilst preserving nets' *boundary protocol*. In this chapter we discuss the implementation of our tool, `Penrose`[1], which uses this approach. We go on to compare and contrast its performance with existing state-of-the-art tools, giving empirical demonstration of its favourable performance in a variety of example systems.

7.1 Implementation

We discuss three key design points of our implementation, each of which strongly influence its efficiency and performance. As discussed in the previous chapter, our technique uses weak *language-preserving* reductions to reduce component semantics, while ensuring all interactions with neighbouring components are preserved (explored in §6.1). Furthermore, since the component structure is exposed, we are able to use *memoisation* to avoid repeating work by recognising already-computed compositions (up-to language-equivalence of the component 2-NFAs) as discussed in §6.2. Finally, as discussed in the related work section (§1.10), using an efficient representation of transitions can radically improve performance; to this end we use MTBDDs to represent the transitions of 2-NFAs.

The three key design points are elaborated as follows:

1. The algorithm for language-preserving reduction of 2-NFAs in §7.1.1,
2. The algorithm for checking language-equvialence of 2-NFAs in §7.1.2,

[1] `Penrose`: Petri Net Reachability Ose. According to Wikipedia, an *Ose* is a demon that gives true answers to secret things.

3. The data structure used to represent 2-NFA transitions in §7.1.3.

Our tool, `Penrose`, is written in Haskell [155], a statically-typed, functional programming language. Commonly, model checkers are implemented using C or similar low-level programming languages, so the use of Haskell is unorthodox in this respect. Despite a highly performant compiler, the levels of abstraction that are present in Haskell (garbage-collection, data structures, etc.) can lead to an overhead that is not present in C. However, the two major benefits Haskell brings outweigh the potential lack of outright performance (which, in practice, is often very slight):

1. A powerful static type system is particularly beneficial when prototyping new implementation details: a large number of run time bugs are ruled out, before the code is executed.

2. High level of abstraction and concise coding style, leads to more direct translations of abstract design, using data structures such as sets and BDDs as first-class abstractions.

Later, we will see that `Penrose` performs favourably against other tools, implemented in standard languages, such as C.

7.1.1 NFA Reduction

A key component of our technique is the algorithm for NFA reduction, that improves the performance of 2-NFA composition by reducing component 2-NFA sizes, as discussed in §6.1. Indeed, as we showed at the end of Chapter 6, the size of 2-NFAs encountered in certain examples grows rapidly and it is therefore important that we attempt to minimise this growth, to avoid performance degradation. Of course, this state growth phenomena is the well-known statespace explosion problem and minimisation is our approach to avoiding it.

An obvious first choice of NFA minimisation algorithm is to use DFA minimisation techniques: by first converting the NFA to a language-equivalent DFA, for example, using the well-known powerset-construction, we can apply DFA minimisation techniques. There are several well-known techniques for DFA minimisation, such as the partition-refinement approaches of Moore [156] and Hopcroft [157]. A simple-to-describe alternative that does not require prior determinisation is that of Brzozowski [158]; Brzozowski showed that reversing an input NFA, determinising, reversing the resulting DFA (the result of which is not necessarily deterministic) and finally determinising generates a minimal DFA, language equivalent to the input NFA.

Indeed, due to its simplicity, we chose Brzozowski's method in the first implementation of our technique [1]. However, implicit in the use of this minimisation technique is the possibility of exponential blowup due to the (double) determinisation.

On the other hand, a positive aspect of the DFA-minimisation approach is that it renders language equivalence checking almost trivial. Since the minimal DFA for a NFA with a particular language is unique, to check language equivalence of two NFA, it is sufficient to simply convert them both to the corresponding unique minimal DFA. If the resulting DFA are equal, the original NFA are language equivalent. However, as we further discuss in §7.1.2, checking language equivalence in this way is still inefficient, since every pair of states of the determinised automata must be checked.

Due to the potential exponential blowup effect of determinisation, it is preferable to directly minimise NFA, rather than determinise NFA before minimising the resulting DFA. Unfortunately, the minimisation problem is more difficult for NFA than DFA, indeed, it is NP-hard [159] and furthermore, even approximating minimisation of NFA is intractable [160]. However, practical algorithms with generally-high performance do exist, such as the recent technique due to Clemente and Mayr [161], which outperforms all prior techniques. Clemente and Mayr's algorithm operates by removing transitions *and* states, whilst preserving the recognised language: transitions are pruned—those transitions that are subsumed by "better" transitions are removed—and states quotiented by an efficiently-computed under-approximation to language equivalence.

Using NFA minimisation means that we can no-longer rely on uniqueness of minimal automata for checking language equivalence: whereas there necessarily exists a minimal DFA for a particular language, there may be several equally-minimal NFA (an example of such NFA is given by Arnold et al. [162]), thus we must use an alternative method of checking language equivalence; in the next subsection, we describe such a method.

7.1.2 Checking NFA Language Equivalence

As previously mentioned, a naive method of checking language equivalence of NFA is to determinise and check for equality of the resulting DFA. However, once the minimisation technique does not use determinisation, we cannot employ this method. An alternative approach, originally due to Hopcroft and Karp [157], and further improved by Bonchi and Pous [163], is to use an on-the-fly determinisation procedure to check equivalence, aiming to only *partially* calculate the determinised automata. Bonchi and Pous' approach exploits the notion of *bisimulation up-to context*; put briefly, if (sets of) states X and Y have equal language, as do Z and W, then bisimulation up-to context asserts that $X \cup Z$ has equal language to $Y \cup W$. The advantage of such a compositional technique is that the language equivalence of $X \cup Z$ and $Y \cup W$ is not *explicitly*

checked. Thus, in the setting of NFA (where a set of NFA states forms a single set of its determinisation), the full corresponding DFA need not be explored.

In practice, we can further improve the performance of checking language equivalence when reaching *fixed-points* (which were introduced and discussed in §6.2.1). If we *tag* all 2-NFA with identifiers (that can be checked for equality in constant time), we can identify fixed-points *without* explicitly checking language equivalence of the underlying 2-NFA. Recall from §6.2 that we maintain two memoisation maps:

1. From PNBs to their (tagged) 2-NFA semantics,
2. From pairs of (tagged) 2-NFA to their (tagged) 2-NFA composition.

We use the first to ensure that a given PNB is translated to its 2-NFA semantics once, and that we use 2-NFA as the representative of its equivalence class. In particular, any member of the same equivalence class will be assigned the same identifier. The second mapping ensures that we only perform a given (up-to language-equivalence) composition once.

Now, consider composing a (left-associated) chain of PNBs: $N \ ; \ N \ ; \ \ldots \ ; \ N$ that reaches a fixed-point after a single composition. The first memoisation map will contain a single entry[2]:

$$N \mapsto \langle\!\langle N \rangle\!\rangle_0$$

When performing 2-NFA compositions, if, having composed two 2-NFA that *do not* appear in the composition memoisation map, we find a language-equivalent 2-NFA in the first map, we use it as the result of the composition (and the entry in the second memoisation map). To demonstrate, the first composition of 2-NFA is $\langle\!\langle N \rangle\!\rangle_0 \ ; \langle\!\langle N \rangle\!\rangle_0$, however, since we reach a fixed point after one composition, we have that $\langle\!\langle N \rangle\!\rangle \ ; \langle\!\langle N \rangle\!\rangle \cong \langle\!\langle N \rangle\!\rangle$, and thus the composition memoisation map is updated to contain a single entry:

$$\langle\!\langle N \rangle\!\rangle_0 \ ; \langle\!\langle N \rangle\!\rangle_0 \mapsto \langle\!\langle N \rangle\!\rangle_0$$

Now, for every remaining composition in the chain, we will be considering $\langle\!\langle N \rangle\!\rangle_0 \ ; \langle\!\langle N \rangle\!\rangle_0$ and thus simply perform two *identifier* comparisons with the single entry in the composition map, before returning the result $\langle\!\langle N \rangle\!\rangle_0$. In general then, fixed-points w.r.t. language-equivalence lead to (cheap) comparison of identifiers, rather than (expensive) checking of language equivalence, a clear optimisation in the case where the fixed-point is non-trivial, for example in Fig. 6.17.

[2]We write N_i for a 2-NFA N tagged with identifier i.

7.1.3 Representing 2-NFA transitions

Recall that the 2-LTS semantics of a PNB $N : (k, l)$, has labels of the form α/β, syntactic sugar for $(\alpha, \beta) \in \mathbb{B}^k \times \mathbb{B}^l$. Via concatenation, these labels can be written as a single binary string of length $k + l$. Therefore, to represent the transitions of the 2-LTS we require a data structure representing $S \times \mathbb{B}^{k+l} \times S$, where S is the set of states of the 2-LTS. The nature of the transition relation means it can be encoded as a function: $S \to \mathbb{B}^{k+l} \to 2^S$, i.e. for each source state, $x \in S$, we have a function $\mathbb{B}^{k+l} \to 2^S$ encoding the set of states reachable from x, via transitions with labels in $\mathbb{B}^{k+l}$.

A well-known, efficient representation of functions $\mathbb{B}^k \to \mathbb{B}$ (i.e. k binary valued-variables that determine a binary value) are Reduced Ordered Binary Decision Diagrams (simply, BDDs) [102]. However, to use BDDs directly, the 2-LTS transition function co-domain must be suitably encoded as a binary string, to extend the binary label, giving a target of a single binary bit. A more natural representation uses Multi-Terminal BDDs [114], a generalisation of BDDs that represent functions with a co-domain other than the booleans.

MTBDDs encode functions that take an assignment of boolean variables to a *set* of values. Specifically, a MTBDD is a ROBDD, where the two-element Boolean algebra of $\mathbb{B}$ has been replaced with the Boolean algebra of subsets of some non-empty set, S.

For our purposes, S is the set of LTS states. The MTBDD's variables are *boundary positions*: $\{(L, i) \mid 0 \leq i < k\} \cup \{(R, j) \mid 0 \leq j < l\}$, where, e.g. $(L, 0)$ is the 0th left boundary port, $(R, 1)$ the first right boundary port and so on. MTBDDs, like BDDs, require that the variables have a fixed ordering, which we take to be the simple ordering top-down, left-before-right, i.e. the lexicographic order (where $L < R$):

$$(lr_1, k) < (lr_2, l) \implies lr_1 < lr_2 \vee (lr_1 = lr_2 \wedge k < l)$$

Example 7.1.
Consider the 2-NFA semantics of a single BUFFER *component, as illustrated in Fig. 7.1.*

The MTBDD representations of the transitions from states 0 *and* 1 *are shown in Fig. 7.2, where we use the convention that the false branch is a dashed line and the true branch is solid. State* 0*'s MTBDD efficiently encodes that there are no transitions from state* 0 *with a* $1/-$ *label.*

Similarly to BDDs, the particular ordering chosen for the variables can drastically affect the representation size of MTBDDs. For example, consider the MTBDD illustrated in Fig. 7.3, where we get a small size decrease when re-ordering variables. In general the effect can be much more pronounced. Clearly it would be preferable to chose the ordering that gave the most compact MTBDD; unfortunately, the problem of choosing

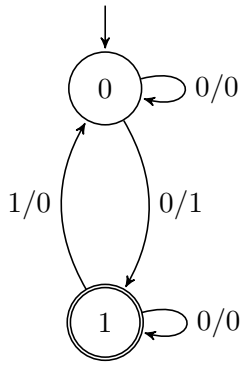
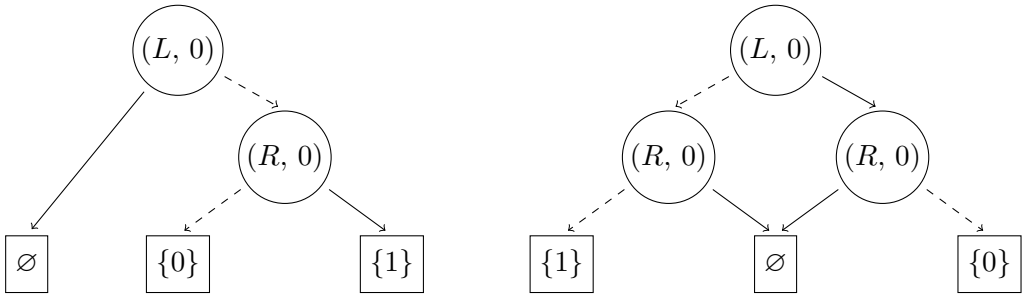


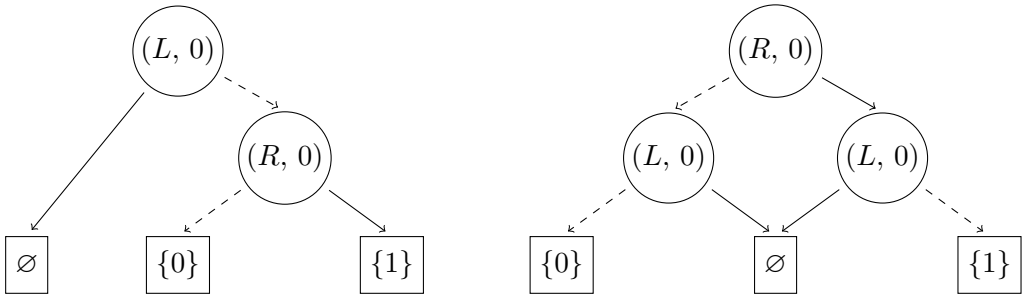
Figure 7.1: 2-NFA semantics of BUFFER



(a) $0 \rightarrow$ (b) $1 \rightarrow$

Figure 7.2: MTBDD representations of the transitions for states 0 and 1, in Fig. 7.1.

an optimal ordering is NP-complete [119], we therefore use the lexicographic ordering since it is simple, and empirical results suggest it performs well.



(a) $(L, 0) < (R, 0)$. (b) $(R, 0) < (L, 0)$.

Figure 7.3: MTBDD size affected by variable ordering: MTBDDs representing $0 \rightarrow$ in Fig. 7.1, with different variable orderings.

7.1.4 Synchronising Composition with MTBDDs

As we have shown, the use of MTBDDs gives efficient representation of transitions in 2-NFA semantics, and furthermore many operations on transitions, such as union of transitions, or ϵ-closure have efficient implementations. However, not all operations are natural and efficient, for example, computing the sequential composition of two states requires computing the *synchronisation* of their transition MTBDDs. Intuitively, we want to compute the MTBDD cross product since the synchronising composition corresponds to taking a transition in each component, before restricting the cross product to transitions that agree on the shared boundary. Thus, when composing a (k, l)-NFA and a (l, m)-NFA, a simple but naive approach is:

1. Generate each of the 2^l variable assignments for the common boundary
2. In turn, instantiate the left and right MTBDDs using each assignment, before performing the cross product on the resulting pair of MTBDDs, generating a *collection* of MTBDDs
3. Finally, collapse the collection of MTBDDs using a pairwise union.

Step 2, involves generating a collection of MTBDDs that record the effect on the non-shared boundaries (i.e. the *outer* boundaries, the left boundary of the first component and the right boundary of the second) when the two 2-NFAs synchronise in each of the 2^l ways. Taking the union of these MTBDDs in step 3 is intuitively ignoring *which* particular synchronisation occurred. This approach is inefficient: it involves explicitly generating a large collection of variable instantiations, even for common variables that do not take part in the synchronisation (and thus do not appear in the MTBDDs), thus we may unnecessarily perform substantial work.

A more efficient approach, that we explore now, is as follows:

1. *Pre-process* the two MTBDDs, renaming variables to tag *synchronisation* variables (those on the common boundaries),
2. Perform the cross product on these MTBDDs,
3. *Remove* any invalid sub-graphs of the MTBDD. We consider a sub-graph to be invalid if it assumes conflicting assignments to a particular boundary variable.
4. Remove the remaining common variables, by taking the union of their sub-graphs.

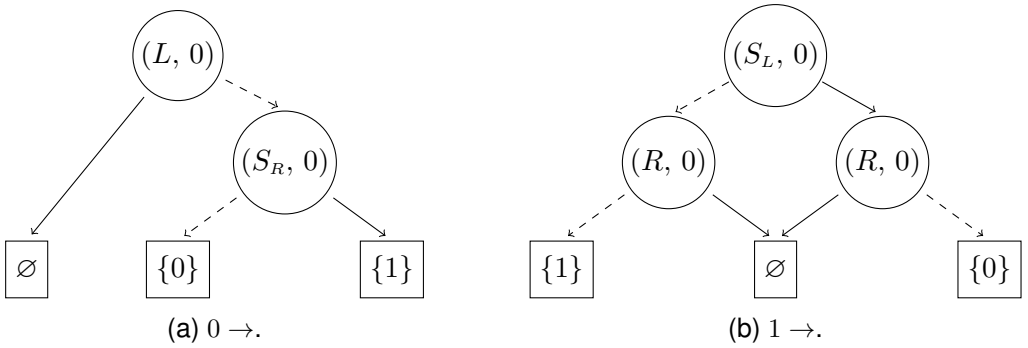
The pre-processing of step 1 renames the first component's variable identifiers from L/R to L/S_R, and the second component's from L/R to S_L/R, i.e. *tagging* synchronisation variables to identify which component they originated from. Importantly, the

order on the resulting pairs is changed, to form the ordering that places all Left variables before the *interleaving* of the Synchronisation variables before all Right variables. For example, this ordering has the following:

$$(L,\,1) < (S_L,\,0) < (S_R,\,0) < (S_R,\,1) < (S_L,\,2) < (R,\,0)$$

This ordering means that in a synchronised MTBDD, synchronisation variables appear in subsequent nodes in the MTBDD, and can easily be checked for conformance.

As an example, we will consider synchronously composing the 2-NFAs of two BUFFER components, with the left component's 2-NFA in state 0, and the right's in state 1. Thus we must compose the MTBDDs shown in Fig. 7.2. First, in step 1, the MTBDDs are pre-processed, leading to the MTBDDs illustrated in Fig. 7.4. Next, step 2 performs the cross product of these two MTBDDs, leading to the MTBDD shown in Fig. 7.5. This MTBDD contains *semantically invalid* sub-graphs, those highlighted red, which are reached by following a path where the synchronisation variables have taken different values. Therefore, in step 3, we remove all such sub-graphs, as shown in Fig. 7.6. Finally, the synchronisations are removed in step 4 by taking the union of the sub-graphs, giving the final result illustrated in Fig. 7.7, which encodes that the only transitions possible from state $(0,\,1)$ are τ transitions, either staying in the same state or reaching $(1,\,0)$.



(a) $0 \rightarrow$. (b) $1 \rightarrow$.

Figure 7.4: "Pre-processed" MTBDDs of Fig. 7.2

Having discussed the three key implementation details for the efficiency of our implementation, we now move onto empirically comparing the resulting performance with existing state-of-the-art tools.

7.2 Comparison with Related Tools

Up to this point, we have explored some of the insights that make our tool efficient. Now, we compare with existing tools that do not take a component-wise view, and thus cannot exploit compositionality as we can, instead, computing global statespaces, or

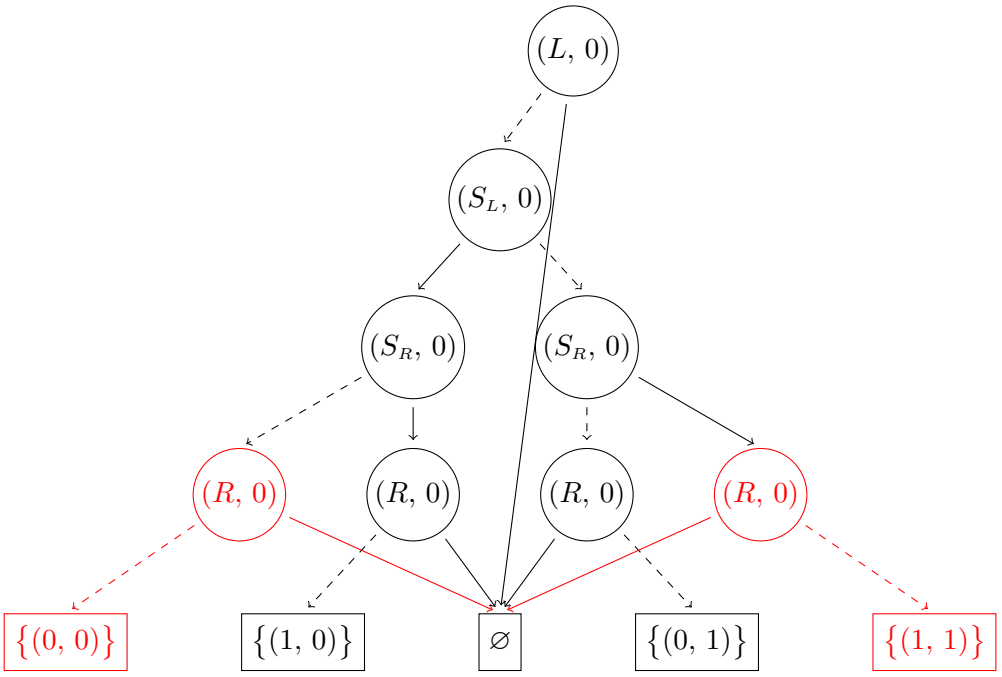


Figure 7.5: MTBDD constructed using the cartesian product of those in Fig. 7.4a and Fig. 7.4b, with invalid sub-graphs highlighted

using unfoldings or other partial-order reduction techniques to avoid the statespace explosion problem. Essentially, we would like to show that while compositionality is a theoretical nicety, it also has significant impact on performance in practice. Therefore, in this section, we quantitatively compare our approach with existing state-of-the-art tools. First, we introduce the tools we are comparing against, before discussing our testing platform and methodology. After presenting the results we interpret them, discussing some key points.

7.2.1 Related Tools

To evaluate the performance of `Penrose`, it was compared with five tools that make up part of the current state-of-the-art:

1. LOLA[3] [164],
2. The PUNF[4] unfolder with CLP[5] checker,
3. The CUNF[6] [165] unfolder, and CNA[7]checker,

[3] http://download.gna.org/service-tech/lola/
[4] http://homepages.cs.ncl.ac.uk/victor.khomenko/tools/punf/
[5] http://homepages.cs.ncl.ac.uk/victor.khomenko/tools/clp/
[6] http://code.google.com/p/cunf/
[7] Available packaged with CUNF

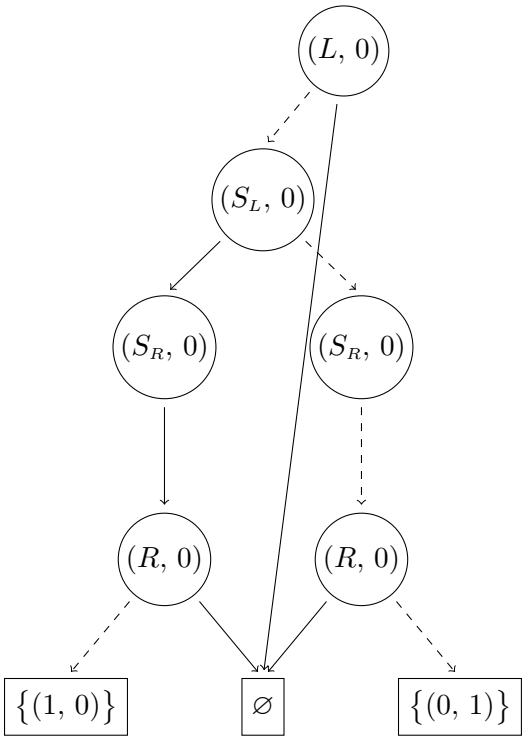


Figure 7.6: MTBDD of Fig. 7.5, with invalid sub-graphs removed

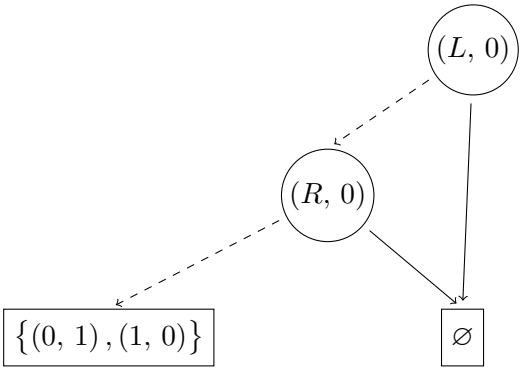


Figure 7.7: MTBDD of Fig. 7.6, with synchronisations removed

4. MARCIE[8] [166],
5. TAPAAL[9].

Several of these tools are *the* current leading choices of tools for Petri net reachability checking and state-space generation. Indeed, in the recent Petri net model-checking competition (MCC'14) [167], LOLA won, and TAPAAL was second in the reachability category, whilst in the state-space generation category MARCIE won, with TAPAAL third. The two other tools, CLP and CNA are both comprised of two distinct tools, one to pre-process (unfold) the input net, and the other to check the required property on

[8] http://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Marcie
[9] http://www.tapaal.net/

the unfolding. CLP's unfolder, PUNF, uses parallelisation techniques from [85], while CLP itself uses linear-programming techniques from [168]. CNA is included since it handles contextual nets (i.e. those with read arcs), and thus is able, like `Penrose`, to calculate results for the OVERTAKE$(-)$ and COUNTER$(-)$ examples.

The tested tools employ different techniques, as reported by the tools' corresponding entries in MCC'14:

- LOLA: stubborn sets, symmetry reduction, unfolding,
- PUNF/CLP: parallelisation, linear programming,
- CUNF/CNA: unfolding, SAT/SMT,
- MARCIE: decision diagrams,
- TAPAAL: symmetry reduction.

For overviews of these techniques, refer back to §1.2.

7.2.2 Testing Platform

All experiments were run on two Ubuntu Linux virtual machines (both with 4GB RAM and a 4-core CPU: one a 32-bit and the other a 64-bit CPU), hosted on an Intel i5-2450 2.50GHz CPU, 8GB of RAM, running 64-bit Ubuntu Linux. CLP required a 32-bit platform, and thus virtual machines were used to give a common-specification platform.

For MARCIE and TAPAAL, we used the virtual machine images provided for the 2014 Petri net Model Checking Competition (MCC 2014) [167], available at `http://mcc.lip6.fr/2014/results.php`. For CUNF, CNA, PUNF and CLP we used pre-compiled binaries from the corresponding authors' websites, and compiled LOLA from source.

7.2.3 Testing Methodology

Tool performance was recorded using the standard Unix `time` command, measuring total (wall-clock) time and peak memory usage[10]. For each tool and example configuration (i.e. example and size parameter(s)) we took the mean of 5 runs, obtaining the average time and memory consumption.

While `Penrose` directly computes using the particular wiring decomposition that specifies each problem; all other tools were provided input that was generated by first computing the composite net and then converting into suitable format[11]. The time taken for

[10] Specifically, using the command `/usr/bin/time -f '%E %M' CMD`.
[11] Either LL_NET format or LOLA's input format.

this conversion was not included in the performance benchmarking—only the processing time of the individual tools was recorded.

7.2.4 Testing Results

In the following results tables we use the key:

T	$\Longrightarrow$	time-out (300 seconds)
M	$\Longrightarrow$	memory-exhaustion (4GB)
/	$\Longrightarrow$	incorrect result
R	$\Longrightarrow$	example skipped due to lack of read arc support

The best performance for each problem instance is highlighted. We present four tables, two recording total time spent, and two recording total memory usage. The tables are split not only for space reasons, but also to differentiate clearly between examples where `Penrose` exhibits scalable performance relative to the competition, and those where it exhibits poor performance. The timing results are presented in Tables 7.1 and 7.2 and the memory results in Tables 7.3 and 7.4.

7.2.5 Discussion

For the systems with scalable performance, `Penrose` is able to check reachability in around 100 milliseconds, even for large parameter-sizes. Similarly, the memory use of `Penrose` is always less than 10MB. A general observation is that other tools do not perform as well for anything other than small parameter sizes. Indeed, the simplest system, BUFFER$(-)$ is only checkable by `Penrose`, LOLA and MARCIE at parameter 512, and only `Penrose` for larger parameters, despite a simple structure and marking.

Since CNA can only check coverability (i.e. it cannot specify that particular places should *not* be marked), we must trivially adapt the FORK component of DPH$(-)$, such that it has two places: one is marked when the fork is present and the other when it is absent. Thus we can specify all forks being taken with the positive marking of all "taken" places, rather than the absence of the token in the original "status" place.

ITER-CHOICE$(-)$ is an example given by Khomenko et al. [96] that is shown to have an exponential unfolding; precisely, ITER-CHOICE(k) has an unfolding with $2^{k+1}-1$ places. Due to this exponentially-sized unfolding, the results show that moderately-sized instances cannot be handled by the tested tools. `Penrose`, on the other hand, is able to handle very large instances quickly. In the originating paper ([96]), an alternative structure, Merged Processes (MPs), were introduced to avoid such exponential unfoldings, however, we are not aware of a model checker that implements MPs.

Table 7.1: Time results

Problem		Time (s)					
name	size	Penrose	CLP	CNA	LOLA	TAAPL	MARCIE
buffer	2	0.009	**0.002**	0.024	0.005	**0.002**	0.045
buffer	8	0.009	**0.002**	0.022	**0.002**	**0.002**	0.045
buffer	32	0.008	0.004	0.279	**0.002**	0.005	0.048
buffer	512	**0.009**	*T*	*M*	0.061	*T*	154.241
buffer	4096	**0.011**	*T*	*M*	*T*	*T*	*M*
buffer	32768	**0.016**	*T*	*M*	*T*	*T*	*M*
over	2	0.028	*R*	**0.021**	*R*	*R*	*R*
over	8	**0.027**	*R*	0.062	*R*	*R*	*R*
over	32	**0.028**	*R*	*M*	*R*	*R*	*R*
over	512	**0.028**	*R*	*M*	*R*	*R*	*R*
over	4096	**0.027**	*R*	*M*	*R*	*R*	*R*
over	32768	**0.030**	*R*	*M*	*R*	*R*	*R*
dac	2	0.009	**0.002**	0.021	**0.002**	**0.002**	0.046
dac	8	0.009	0.002	0.019	**0.001**	0.002	0.044
dac	32	0.009	**0.002**	0.034	**0.002**	0.004	0.053
dac	512	**0.010**	*T*	*T*	4.034	0.083	25.451
dac	4096	**0.009**	*T*	*T*	*T*	*M*	*M*
dac	32768	**0.011**	*T*	*T*	*T*	*M*	*M*
philo	2	0.017	/	0.021	0.008	**0.002**	0.044
philo	8	0.022	/	0.021	0.015	**0.004**	0.047
philo	32	0.021	/	0.025	*M*	**0.009**	0.083
philo	512	**0.020**	*T*	1.067	*M*	*T*	197.861
philo	4096	**0.021**	*T*	*M*	*M*	*T*	*M*
philo	32768	**0.022**	*T*	*M*	*M*	*T*	*M*
iter-choice	2	0.009	0.002	0.022	**0.001**	0.002	0.044
iter-choice	8	0.010	19.247	19.054	0.006	**0.003**	0.046
iter-choice	32	0.010	*T*	*M*	*M*	**0.006**	0.055
iter-choice	512	**0.009**	*T*	*M*	*M*	6.066	36.059
iter-choice	4096	**0.010**	*T*	*M*	*M*	*M*	*M*
iter-choice	32768	**0.012**	*T*	*M*	*M*	*M*	*M*
replicator	2	0.009	/	0.024	**0.002**	**0.002**	*T*
replicator	8	0.008	/	0.022	**0.001**	0.003	*T*
replicator	32	0.009	/	0.023	**0.002**	0.005	*T*
replicator	512	0.010	/	1.025	**0.004**	111.428	*T*
replicator	4096	**0.009**	/	74.057	0.086	*M*	*T*
replicator	32768	**0.012**	/	*M*	231.852	*M*	*T*
$T_\wedge(-,-)$	1,1	0.008	**0.002**	0.019	**0.002**	**0.002**	0.044
$T_\wedge(-,-)$	2,2	0.009	0.002	0.019	**0.001**	0.003	0.044
$T_\wedge(-,-)$	3,3	0.010	0.002	0.019	**0.001**	0.004	0.046
$T_\wedge(-,-)$	4,4	0.009	0.002	0.025	**0.001**	0.012	2.065
$T_\wedge(-,-)$	5,5	0.009	0.009	0.077	**0.006**	9.070	*M*
$T_\wedge(-,-)$	6,6	**0.009**	10.024	12.449	12.087	*M*	*M*
$T_\wedge(-,-)$	7,7	**0.009**	*T*	*M*	*T*	*M*	*M*

Table 7.2: Time results continued

Problem		Time (s)					
name	size	Penrose	CLP	CNA	LOLA	TAAPL	MARCIE
counter	2	**0.011**	*R*	0.020	*R*	*R*	*R*
counter	4	**0.021**	*R*	0.022	*R*	*R*	*R*
counter	8	0.640	*R*	**0.047**	*R*	*R*	*R*
counter	10	1.278	*R*	**1.045**	*R*	*R*	*R*
counter	13	**4.038**	*R*	14.057	*R*	*R*	*R*
counter	16	**8.050**	*R*	*T*	*R*	*R*	*R*
cyclic	2	0.012	/	0.021	**0.001**	0.003	0.043
cyclic	4	0.026	/	0.020	**0.001**	0.003	0.045
cyclic	8	1.014	/	0.022	**0.001**	0.004	0.053
cyclic	10	1.274	/	0.024	**0.002**	0.005	0.069
cyclic	13	4.004	/	0.023	**0.002**	0.006	2.035
cyclic	16	7.048	/	0.021	**0.003**	0.005	19.220
hartstone	2	0.017	**0.002**	/	**0.002**	/	/
hartstone	4	0.037	0.002	/	**0.001**	/	/
hartstone	8	1.084	0.002	/	**0.001**	/	/
hartstone	10	3.270	**0.001**	/	**0.001**	/	/
hartstone	13	10.045	**0.002**	/	**0.002**	/	/
hartstone	16	25.031	**0.002**	/	**0.002**	/	/
token-ring	2	0.015	0.002	0.020	**0.001**	**0.001**	0.043
token-ring	4	0.048	0.002	0.025	**0.001**	**0.001**	0.044
token-ring	8	2.063	0.007	0.080	**0.002**	**0.002**	0.045
token-ring	10	4.466	0.053	4.637	**0.002**	0.003	0.047
token-ring	13	10.650	*T*	95.065	0.016	**0.003**	0.050
token-ring	16	20.460	*T*	*T*	2.030	**0.003**	0.048
$T_V(-,-)$	1,1	0.009	**0.001**	0.020	**0.001**	**0.001**	0.045
$T_V(-,-)$	2,2	0.011	/	/	**0.001**	/	/
$T_V(-,-)$	3,3	13.478	/	/	**0.001**	/	/
$T_V(-,-)$	4,4	*T*	/	/	**0.002**	/	/
$T_V(-,-)$	5,5	*T*	/	/	**0.009**	/	*M*
$T_V(-,-)$	6,6	*T*	/	/	**18.669**	*M*	*M*

Some proportion of the overhead encountered by other tools in large examples can be attributed to the size of the input file that must be processed (e.g. the ll_net format is ≈ 500 KB for OVERTAKE(512) and LOLA's input is ≈ 1.3 MB for DAC(4096)). Indeed, this is part of the point of this thesis: using component-wise specification and PNBml, we can specify certain systems in a vastly more concise way.

Furthermore, since we interpret the benchmark systems as 1-bounded (recall that PNBs use elementary-net firing semantics) and that the tested tools do not give the ability to restrict place token bounds, is likely that a large proportion of the time spent is wasted exploring unnecessary states (where places have >1 token).

An observation regarding `Penrose`'s performance is that certain markings are certainly

Table 7.3: Memory results

Problem		Max Resident (MB)					
name	size	Penrose	CLP	CNA	LOLA	TAAPL	MARCIE
buffer	2	4.66	8.23	14.54	**2.60**	6.47	506.98
buffer	8	4.67	8.23	14.87	**2.86**	11.07	507.24
buffer	32	4.66	8.50	19.75	**2.87**	29.69	509.56
buffer	512	**4.69**	*T*	*M*	31.52	*T*	1087.10
buffer	4096	**4.93**	*T*	*M*	*T*	*T*	*M*
buffer	32768	**6.71**	*T*	*M*	*T*	*T*	*M*
over	2	**7.39**	*R*	14.77	*R*	*R*	*R*
over	8	**7.39**	*R*	28.36	*R*	*R*	*R*
over	32	**7.39**	*R*	*M*	*R*	*R*	*R*
over	512	**7.42**	*R*	*M*	*R*	*R*	*R*
over	4096	**7.75**	*R*	*M*	*R*	*R*	*R*
over	32768	**8.69**	*R*	*M*	*R*	*R*	*R*
dac	2	4.73	8.24	14.59	2.61	**2.12**	507.23
dac	8	4.73	8.26	15.08	2.87	**2.37**	507.50
dac	32	4.72	8.55	20.43	3.12	**3.04**	510.66
dac	512	**4.76**	*T*	*T*	52.00	75.49	1092.43
dac	4096	**5.01**	*T*	*T*	*T*	*M*	*M*
dac	32768	**6.73**	*T*	*T*	*T*	*M*	*M*
philo	2	6.20	/	14.62	**2.87**	9.55	507.24
philo	8	6.23	/	15.01	**3.38**	23.60	508.11
philo	32	**6.22**	/	16.52	*M*	79.86	523.23
philo	512	**6.22**	*T*	196.01	*M*	*T*	1094.48
philo	4096	**6.22**	*T*	*M*	*M*	*T*	*M*
philo	32768	**9.73**	*T*	*M*	*M*	*T*	*M*
iter-choice	2	4.75	8.24	14.71	**2.61**	8.54	507.24
iter-choice	8	**4.75**	116.25	783.72	15.92	17.70	507.49
iter-choice	32	**4.76**	*T*	*M*	*M*	55.05	512.76
iter-choice	512	**4.78**	*T*	*M*	*M*	864.47	1160.22
iter-choice	4096	**5.02**	*T*	*M*	*M*	*M*	*M*
iter-choice	32768	**6.75**	*T*	*M*	*M*	*M*	*M*
replicator	2	4.70	/	14.57	**2.61**	7.25	*T*
replicator	8	4.69	/	14.72	**2.86**	11.88	*T*
replicator	32	4.68	/	15.36	**2.86**	30.66	*T*
replicator	512	**4.71**	/	35.98	6.47	818.19	*T*
replicator	4096	**4.96**	/	1582.63	37.59	*M*	*T*
replicator	32768	**6.73**	/	*M*	504.84	*M*	*T*
$T_\wedge(-,-)$	1,1	4.69	8.21	14.52	**2.61**	5.56	506.98
$T_\wedge(-,-)$	2,2	4.74	8.22	14.55	**2.61**	7.78	506.98
$T_\wedge(-,-)$	3,3	4.74	8.23	14.75	**2.86**	20.45	507.49
$T_\wedge(-,-)$	4,4	4.73	8.30	16.42	**3.12**	136.62	589.97
$T_\wedge(-,-)$	5,5	**4.73**	9.54	36.46	7.76	1556.56	*M*
$T_\wedge(-,-)$	6,6	**4.73**	55.53	1291.74	81.94	*M*	*M*
$T_\wedge(-,-)$	7,7	**4.73**	*T*	*M*	*T*	*M*	*M*

Table 7.4: Memory results continued

Problem		Max Resident (MB)					
name	size	Penrose	CLP	CNA	LOLA	TAAPL	MARCIE
counter	2	**5.33**	*R*	14.59	*R*	*R*	*R*
counter	4	**6.14**	*R*	14.80	*R*	*R*	*R*
counter	8	**8.00**	*R*	25.38	*R*	*R*	*R*
counter	10	**9.73**	*R*	62.12	*R*	*R*	*R*
counter	13	**11.84**	*R*	464.33	*R*	*R*	*R*
counter	16	**13.62**	*R*	*T*	*R*	*R*	*R*
cyclic	2	5.77	/	14.65	**2.86**	10.89	507.24
cyclic	4	6.03	/	14.82	**2.86**	16.17	507.50
cyclic	8	6.25	/	15.14	**2.86**	27.29	510.95
cyclic	10	7.67	/	15.28	**2.86**	32.78	520.14
cyclic	13	8.04	/	15.52	**3.12**	40.96	603.46
cyclic	16	9.73	/	15.76	**3.41**	48.96	1041.87
hartstone	2	6.02	8.23	/	**2.65**	/	/
hartstone	4	6.09	8.23	/	**2.86**	/	/
hartstone	8	6.46	8.24	/	**2.87**	/	/
hartstone	10	8.02	8.24	/	**2.86**	/	/
hartstone	13	8.02	8.24	/	**2.86**	/	/
hartstone	16	8.96	8.25	/	**2.86**	/	/
token-ring	2	5.82	8.23	14.64	2.76	**2.12**	507.24
token-ring	4	6.26	8.24	15.41	2.87	**2.19**	507.24
token-ring	8	7.04	9.78	42.08	2.87	**2.31**	507.49
token-ring	10	8.96	28.87	151.99	3.51	**2.42**	507.75
token-ring	13	9.73	*T*	1403.45	16.85	**2.51**	508.35
token-ring	16	10.76	*T*	*T*	145.44	**2.61**	508.79
$T_{\vee}(-,-)$	1,1	4.69	8.21	14.52	**2.61**	5.56	506.98
$T_{\vee}(-,-)$	2,2	5.07	/	/	**2.61**	/	/
$T_{\vee}(-,-)$	3,3	13.67	/	/	**2.87**	/	/
$T_{\vee}(-,-)$	4,4	*T*	/	/	**3.90**	/	/
$T_{\vee}(-,-)$	5,5	*T*	/	/	**16.60**	/	*M*
$T_{\vee}(-,-)$	6,6	*T*	/	/	**213.56**	*M*	*M*

going to be much faster to check for a given system than others; a simple example being $T_{\vee}(-,-)$. The tested marking requires a token in *every* leaf place, given a single starting token in the root place. `Penrose` is extremely slow to check this reachability problem, due to the statespace explosion problem: large intermediate 2-NFAs are constructed, as discussed at the end of Chapter 6. Indeed, `Penrose` calculates the intermediate results that record *all* possible ways to introduce enough tokens into the sub-systems, to reach the desired marking. However, it is only at the final composition step that `Penrose` discovers that there will only ever be a single token in the system, and thus all intermediate results are discarded. Indeed, changing to the target marking that requires a token in *any one* leaf place *is* fast to check. Observe that `Penrose` currently implements a strict evaluation strategy, sub-system reachability 2-NFAs are calculated, without them necessarily being required. Indeed, this is necessary for the

current memoisation implementation — to know if we have seen a particular 2-NFA composition before, we must fully evaluate each of the arguments. Future work is to explore the possibility of using a lazy evaluation strategy, whereby evaluation is only performed on demand — in the poorly performing example, one could imagine only evaluating two far enough to observe that the sub-system requires at least two tokens to reach its target marking, which cannot be provided by the root's single token, thus the global marking is not reachable.

Similarly, HARTSTONE$(-)$ and TOKENRING$(-)$ (which are fundamentally quite similar), contain sub-nets that can store n tokens, thus the 2-NFA must represent that tokens can be (arbitrarily) added or removed. In fact, only one "token" is ever inserted/removed into the system. Future work will investigate the applicability of techniques such as Larsen's relativized bisimulation [169], that control the allowable interactions with an environment (in this case, ensuring at most one token is taken from the context).

Our final comments are to draw attention to the fact that tools based on unfolding or other partial-order reduction are mature, with years of development, and established tools such as LOLA have been highly optimised. `Penrose` is implemented in the high level functional language Haskell, and further, has had little to no optimisation effort applied — it is a proof-of-concept. Despite this, it is able to out-perform the tools in several examples, as demonstrated by our results. It could be argued that the playing field is unfair: `Penrose` uses a formal description of the decomposition of a problem at hand into smaller components while other tools take a global, monolithic net as input. This is, however, precisely our point: there is no reason for model checkers not to take advantage of compositional descriptions — it is how real systems are specified/designed.

We see significant room for future improvement in `Penrose`; indeed, our approach makes no use of the partial-order reduction techniques of other tools, which we conjecture are orthogonal to and therefore compatible with our compositional approach.

7.3 Summary

In this chapter, we have shown that in addition to being a theoretical nicety, compositionality also leads to impressive reachability checking performance of our tool, `Penrose`, for many example systems. However, while in many cases fixed-points of behaviour exist, and lead to exponential increases in performance, there are examples where the statespace explosion problem still presents itself, hampering our compositional approach. Future work will investigate integrating existing approaches to avoiding statespace explosion with our compositional approach, but we think that the initial results presented show that compositionality, and the component-wise approach, is a promising *orthogonal* method of attacking statespace explosion.

Chapter 8

Conclusion

In this thesis, we investigated and advocated compositional system specification and an alternative approach to reachability checking that uses the structural compositional information to its advantage, in order to vastly improve efficiency in many examples.

The contributions presented in this thesis were:

1. The elucidation of the categorical structure of PNBs and their semantics: we showed the well-known property of compositionality in a new light, as an instance of functoriality for suitable categories.
2. The introduction of *contextual PNBs*, which naturally model behaviour that *non-destructively* reads the token state of a place.
3. The motivation and introduction of a type-checked specification programming language for PNBs, which ensures that only correct compositions are expressible.
4. We demonstrated *compositional* statespace generation for PNB systems, and that it can be used to check reachability, *without* constructing the global net.
5. We showed that compositional specifications can be exploited, to attack the statespace explosion problem, and improve the efficiency of reachability checking of systems modelled using PNBs. We showed that by considering *weak language equivalence* of PNB semantics, we are able to reduce the representation size of PNB semantics, whilst ensuring global behaviour is preserved. We demonstrated that *memoisation* allows us to avoid repeated computation.
6. We gave compositional specification of existing benchmarks in a more natural component-wise specification, with explicit specification of repeated structure.

Many of these contributions had been introduced in the author's papers [1, 2, 3].

Chapter 2 contained the required preliminaries. In Chapter 3, the categorical structure of PNBs and the LTS that form their semantics was presented, exposing the notion of compositionality as functoriality. Chapter 4 introduced the example systems that we used to demonstrate and evaluate our technique, and a specification DSL that uses a static type system to ensure that only valid component-wise specifications can be constructed. In Chapter 5 we introduced a compositional technique for generating the statespace of systems specified using our DSL, and thus checking marking reachability. We proved the technique correct and give some example timings of a tool implementing the technique. Chapter 6 showed how to exploit the fact that language equivalence is a congruence to vastly improve the performance of our reachability-checking technique, introducing the notion of internal behaviour that we *ignore* in order to aggressively prune statespace. We proved the more-efficient algorithm correct. In Chapter 7, we discussed the implementation of our technique, and compared and discussed its performance relative to current state-of-the-art tools.

8.1 Future Work

We now briefly discuss several avenues of work to extend the compositional approach introduced in this thesis.

1. *What is a good decomposition? What characterises good performance?*

   When discussing our example systems in §4.1, we did not discuss *how* we arrived at the particular decompositions. In practice, the decompositions were natural, and were generated by hand. However, an automated decomposition search might be preferred, which could detect and abstract out repeated components. In our original paper [1] introducing our technique, we described a naive decomposition algorithm; unfortunately, in practice the decomposition algorithm tended to not obtain the natural *by hand* decompositions, and did not give generally acceptable performance. We will further investigate decomposition as a method of applying our technique to existing *monolithic* models. The size of a PNB's boundaries and the number of places it contains influences the size of the corresponding 2-NFA and thus the performance of our technique. As per our pre-print paper [146] the *rank width* of the underlying hypergraph is important in determining behaviour; we will further investigate structurally characterisations of PNBs.

2. *How to produce witnesses?*

   A drawback of using τ-closure and minimisation to combat the statespace explosion problem is that we cannot immediately produce witnessing transitions that confirm the yes/no answer to the reachability question. The difficulty would be to preserve

the minimal information required to generate a witnessing transition sequence, without forcing all possibly valid sequences to be remembered.

3. *Directed exploration of statespaces*

   In the style of Bonet et al. [86], the conversion of a PNB expression to a 2-NFA might be *directed* towards sub-expressions with not-reachable markings. Recall that if any component's local marking is unreachable, the global marking is unreachable; if the evaluation can quickly identify such components, then it can quickly determine unreachability in the global net.

4. *Prevention of invalid intermediate behaviour*

   As noted in §6.5 and by Graf and Steffan [50] intermediate results may exhibit behaviours that are ultimately not performed in the composite system. Here, the ideas of contextual bisimulation [169], or interface components of Clarke et al. [42] will help alleviate intermediate 2-NFA sizes. However, the main difficulty will be determining the correct restrictions that the context should enforce — for example, in the case of $\textsc{tokenring}(-)$, how should the procedure determine that only a single token should be emitted from the context? Additionally, lazy evaluation of 2-NFA may be helpful, such that only the structure that takes part in successful computation is evaluated. For example, since the token ring context only emits a single token, the token ring itself should not evaluate transitions that rely on more than one token being emitted.

5. *Unfoldings/Merged processes for PNBs*

   Similarly to how an unfolding is a (suitably restricted) Petri net, we expect that with suitable restrictions, the unfolding of a PNB will itself be a PNB. Furthermore, we expect that compositionality should also hold for unfoldings of PNB components (with a suitable equivalence relation).

   We can foresee at least three subtle points that must be accounted for:

   (a) Unfoldings will likely require complement places, to record token *absence*

   (b) Composition of unfoldings will induce additional contention

   (c) Equivalence must be suitably defined, isomorphism is likely too strong

   To demonstrate these points, consider the left PNB in Fig. 8.1; we postulate that its unfolding will necessarily contain a *complement* place, $\bar{p}$, and transitions to ensure that when the p is empty, $\bar{p}$ is full and vice-versa. An unfolding is *seeded* with a place for each place assigned a token by the initial marking; if there is no such places, the unfolding will be empty. Furthermore, there should be additional *contention* between transitions of composed PNB unfoldings: consider synchronously composing the two leftmost unfoldings shown in Fig. 8.2, the composition should not allow for example the dashed transition in one component unfolding to synchronise with a solid transition in the other component. Since unfoldings record the *history* of

a net, allowing such synchronisations would be akin to one component going back in time. Finally, the notion of equivalence would need to be suitably defined; using a naive approach, the composition of unfoldings is *not* isomorphic to the unfolding of a composed net. For example, composing the two leftmost unfoldings in Fig. 8.2 should obtain an equivalent PNB to the rightmost PNB, which is the unfolding of the composite net of Fig. 8.1; immediately, the composition of unfoldings contains too many places, arising from the complement places $\bar{p}$ and $\bar{q}$.

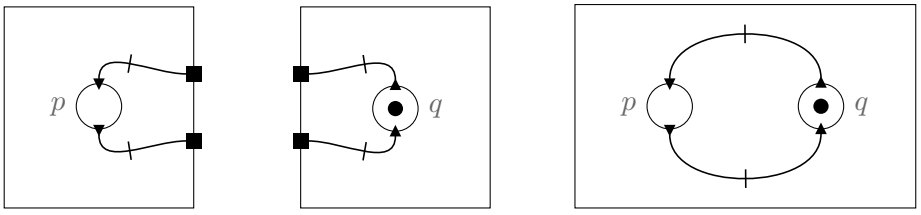


Figure 8.1: Two example PNBs and their composition

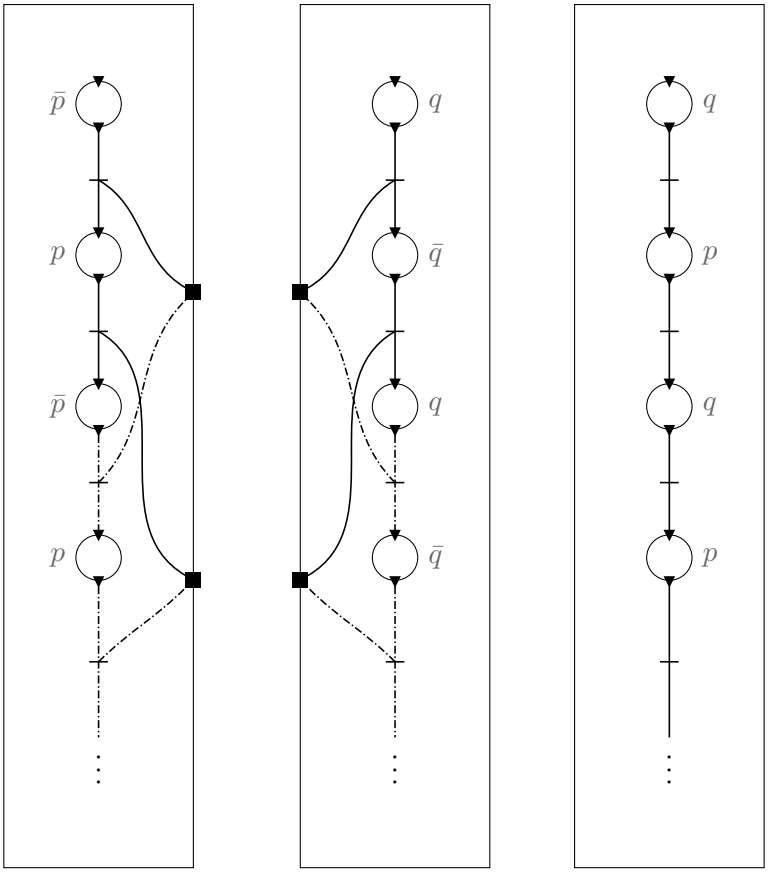


Figure 8.2: Unfoldings of the PNBs in Fig. 8.1

6. *Applying existing statespace avoidance techniques to PNBs*

Since our approach to avoiding statespace explosion (minimisation w.r.t. weak language-equivalence) is orthogonal to existing approaches discussed in §1.2, we

will investigate integrating these methods, to improve the performance of our technique, especially when fixed points of behaviour are *not found*.

References

[1] P. Sobociński and O. Stephens, “Penrose: Putting Compositionality to Work for Petri Net Reachability,” in *Algebra and Coalgebra in Computer Science*, 2013, pp. 346–352. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-40206-7_29

[2] ——, “A Programming Language for Spatial Distribution of Net Systems,” in *Application and Theory of Petri Nets and Concurrency*, G. Ciardo and E. Kindler, Eds. Springer International Publishing, 2014, pp. 150–169.

[3] J. Rathke, P. Sobociński, and O. Stephens, “Compositional Reachability in Petri Nets,” in *Workshop on Reachability Problems*. Springer International Publishing, 2014, pp. 230–243.

[4] C. A. Petri, “Kommunication mit Automaten,” Ph.D. dissertation, Technischen Hoschule Darmstadt, 1962.

[5] T. Murata, “Petri Nets: Properties, Analysis and Applications,” *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=24143

[6] E. W. Mayr, “An Algorithm for the General Petri Net Reachability Problem,” in *Symposium on Theory of Computing*. New York, New York, USA: ACM Press, 1981, pp. 238–246. [Online]. Available: http://portal.acm.org/citation.cfm?doid=800076.802477

[7] S. R. Kosaraju, “Decidability of Reachability in Vector Addition Systems,” in *Symposium on Theory of Computing*. ACM, 1982, pp. 267–281.

[8] J. Lambert, “A Structure to Decide Reachability in Petri Nets,” *Theoretical Computer Science*, vol. 99, pp. 79–104, 1992.

[9] R. J. Lipton, “The Reachability Problem Requires Exponential Space,” Department of Computer Science, Yale University, Technical Report 63, 1976.

[10] J. Esparza and M. Nielsen, “Decidability Issues for Petri nets,” *Bulletin of the EATCS*, vol. 52, no. May, pp. 244–262, 1994. [Online]. Available: http://home.ifi.uio.no/andersmo/petrinet/papers/complexity/brics_98_8.pdf

[11] A. Cheng, J. Esparza, and J. Palsberg, “Complexity Results for 1-safe Nets,” *Theoretical Computer Science*, vol. 147, no. 1-2, pp. 117–136, Aug. 1995. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/0304397594002317

[12] K. Wolf, “The Petri net Twist in Explicit Model Checking,” *Software & Systems Modelling*, 2014.

[13] A. Valmari, “The State Explosion Problem,” *LNCS - Lectures on Petri Nets I: Basic Models*, vol. 1491, pp. 429–528, 1998. [Online]. Available: http://www.cs.vsb.cz/kot/download/Texts/StateSpace.pdf

[14] ——, “Stubborn Sets for Reduced State Space Generation,” in *Application and Theory of Petri Nets*, vol. 483, 1990, pp. 491–515.

[15] P. Godefroid, “Using Partial Orders to Improve Automatic Verification Methods,” in *Computer Aided Verification*. Springer Berlin Heidelberg, 1990, pp. 176–185. [Online]. Available: http://portal.acm.org/citation.cfm?id=735044

[16] D. Peled, “All from One, One for All: on Model Checking using Representatives,” in *Computer Aided Verification*. Springer Berlin Heidelberg, 1993, pp. 409–423. [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-56922-7_34

[17] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer Berlin Heidelberg, 1996. [Online]. Available: http://portal.acm.org/citation.cfm?id=547238

[18] E. M. Clarke, O. Grumberg, M. Minea, and D. Peled, “State Space Reduction using Partial Order Techniques,” *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 3, pp. 279–287, 1999. [Online]. Available: http://link.springer.com/article/10.1007/s100090050035

[19] A. Valmari and H. Hansen, “Can Stubborn Sets be Optimal?” in *Application and Theory of Petri Nets*, 2010, pp. 43–62.

[20] K. Schmidt, “Stubborn Sets for Standard properties,” in *Application and Theory of Petri Nets*. Springer Berlin Heidelberg, 1999, pp. 46–65. [Online]. Available: http://www.springerlink.com/index/mtj3p6183xchedr9.pdf

[21] P. Godefroid and D. Pirottin, “Refining Dependencies Improves Partial-order Verification Methods,” *Computer Aided Verification*, vol. 697, no. 6021, pp. 438–449, 1993. [Online]. Available: http://dx.doi.org/10.1007/3-540-56922-7_36

[22] P. Wolper and P. Godefroid, “Partial-Order Methods for Temporal Verification,” in *CONCUR*, vol. 715. Springer-Verlag, 1993, pp. 233–246.

[23] R. Alur, R. Brayton, T. Henzinger, S. Qadeer, and S. Rajamani, “Partial-order Reduction in Symbolic State Space Exploration,” in *Computer Aided Verification*. Springer Berlin Heidelberg, 1997, pp. 340–351.

[24] H. Hansen and X. Wang, "Compositional Analysis for Weak Stubborn Sets," in *Application of Concurrency to System Design*. IEEE, 2011, pp. 36–43.

[25] F. Vernadat, P. Azéma, and F. Michel, "Covering Step Graph," in *Application and Theory of Petri Nets*. Springer Berlin Heidelberg, 1996, pp. 516–535.

[26] P.-O. Ribet, B. Berthomieu, and F. Vernadat, "On Combining the Persistent Sets Method with the Covering Steps Graph Method," in *Formal Techniques for Networked and Distributed Systems*, 2002, pp. 344–359.

[27] P. Godefroid, "On the Costs and Benefits of Using Partial-Order Methods for the Verificiation of Concurrent Systems," in *DIMACS workshop on Partial Order Methods in Verification*, 1996, pp. 289–303.

[28] P. H. Starke, "Reachability analysis of petri nets using symmetries," *Systems Analysis Modeling Simul*, vol. 8, no. 4-5, pp. 293–303, Aug. 1991. [Online]. Available: http://dl.acm.org/citation.cfm?id=115220.115224

[29] E. Clarke, R. Enders, T. Filkorn, and S. Jha, "Exploiting Symmetry in Temporal Logic Model Checking," in *Computer Aided Verification*, 1993, pp. 450–462. [Online]. Available: http://link.springer.com/article/10.1007/BF00625969

[30] E. A. Emerson and A. P. Sistla, "Symmetry And Model Checking," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, C. Courcoubetis, Ed., vol. 697, no. 1-2. Berlin, Heidelberg: Springer Berlin Heidelberg, Aug. 1993, pp. 105–131. [Online]. Available: http://link.springer.com/10.1007/BF00625970http://www.springerlink.com/index/10.1007/3-540-56922-7

[31] C. Norris Ip and D. L. Dill, "Better Verification Through Symmetry," *Formal Methods in System Design*, vol. 9, no. 1-2, pp. 41–75, Aug. 1996. [Online]. Available: http://link.springer.com/article/10.1007/BF00625968http://link.springer.com/10.1007/BF00625968

[32] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla, "Symmetry Reductions in Model Checking," in *Computer Aided Verification*, A. J. Hu and M. Y. Vardi, Eds., no. 388. Vancouver BC, Canada: Springer Berlin Heidelberg, 1998, pp. 147–158. [Online]. Available: http://link.springer.com/chapter/10.1007/BFb0028741

[33] R. C. Read and D. G. Corneil, "The Graph Isomorphism Disease," *Journal of Graph Theory*, vol. 1, no. 4, pp. 339–363, Jan. 1977. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/jgt.3190010410/abstracthttp://doi.wiley.com/10.1002/jgt.3190010410

[34] A. Sistla, V. Gyuris, and E. Emerson, "SMC: a Symmetry-Based Model Checker for Verification of Safety and Liveness Properties," *ACM Transactions on Software ...*, vol. 9, no. 2, pp. 133–166, 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=350891

[35] M. Leuschel and M. Butler, "ProB: A Model Checker for B," in *FME 2003: Formal Methods*, 2003, pp. 855–874. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-540-45236-2_46

[36] G. Holzmann, "The Model Checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, May 1997. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=588521

[37] K. Schmidt, "How to Calculate Symmetries of Petri nets," *Acta Informatica*, vol. 36, no. 7, pp. 545–590, Jan. 2000. [Online]. Available: http://link.springer.com/article/10.1007/s002360050002http://link.springer.com/10.1007/s002360050002

[38] T. A. Junttila, "Computational Complexity of the Place / Transition-Net Symmetry Reduction Method," *Journal of Universal Computer Science*, vol. 7, no. 4, pp. 307–326, 2001.

[39] K. Schmidt, "Integrating Low Level Symmetries into Reachability Analysis," in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 1785, 2000, pp. 315–330. [Online]. Available: http://www.springerlink.com/index/D8DCKGMUF9P3BQ3C.pdf

[40] T. Wahl and A. Donaldson, "Replication and Abstraction: Symmetry in Automated Formal Verification," *Symmetry*, vol. 2, no. 2, pp. 799–847, Apr. 2010. [Online]. Available: http://www.mdpi.com/2073-8994/2/2/799/

[41] A. Miller, A. Donaldson, and M. Calder, "Symmetry in Temporal Logic Model Checking," *ACM Computing Surveys*, vol. 38, no. 3, pp. 1–40, Sep. 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1132962http://portal.acm.org/citation.cfm?doid=1132960.1132962

[42] E. M. Clarke, D. E. Long, and K. L. McMillan, "Compositional Model Checking," in *Proceedings of the Fourth IEEE Symposium on Logic in Computer Science*, no. 4976, 1989, pp. 353–362. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=39190

[43] R. Milner, *A Calculus of Communicating Systems*. Springer Berlin Heidelberg, 1990, vol. 92.

[44] W. J. Yeh and M. Young, "Compositional Reachability Analysis using Process Algebra," in *International Symposium on Testing, Analysis and Verification*. Victoria, BC, Canada: ACM New York, 1991, pp. 49–59. [Online]. Available: http://dl.acm.org/citation.cfm?id=120812

[45] A. Valmari, "Compositional State Space Generation," in *Advances in Petri Nets 1993*, G. Rozenber, Ed. Springer Berlin Heidelberg, 1993, pp. 427–457. [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-56689-9_54

[46] ——, "Compositionality in State Space Verification Methods," in *Application and Theory of Petri Nets*, J. Billington and W. Reisig, Eds. Osaka, Japan: Springer Berlin Heidelberg, 1996, pp. 29–56.

[47] ——, "Compositional Analysis with Place-Bordered Subnets," in *Application and Theory of Petri Nets*, R. Valette, Ed. Zaragoza, Spain: Springer Berlin Heidelberg, 1994, pp. 531–547. [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-58152-9_29

[48] E. Kindler, "A Compositional Partial Order Semantics for Petri Net Components," in *Application and Theory of Petri Nets*, P. Azéma and G. Balbo, Eds. Toulouse, France: Springer Berlin Heidelberg, 1997, pp. 235–252.

[49] P. Baldan, F. Bonchi, F. Gadducci, and G. Valentina, "Modular Encoding of Synchronous and Asynchronous Interactions Using Open Petri Nets," *Science of Computer Programming*, vol. In Press, 2015. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2014.11.019

[50] S. Graf and B. Steffen, "Compositional Minimization of Finite State Systems," in *Computer Aided Verification*. Springer-Verlag, 1990, pp. 186–196. [Online]. Available: http://link.springer.com/chapter/10.1007/BFb0023732

[51] S. C. Cheung and J. Kramer, "Enhancing Compositional Reachability Analysis with Context Constraints," *ACM SIGSOFT Software Engineering Notes*, vol. 18, no. 5, pp. 115–125, 1993. [Online]. Available: http://dl.acm.org/citation.cfm?id=167071

[52] J.-P. Krimm and L. Mounier, "Compositional State Space Generation From Lotos Programs," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 1997, pp. 239–258.

[53] W. J. Lee, S. D. Cha, Y. R. Kwon, and H. N. Kim, "A Slicing-Based Approach to Enhance Petri net Reachability Analysis," *Journal of Research Practices and Information Technology*, vol. 32, no. 2, pp. 131–143, 2000. [Online]. Available: http://pdf.aminer.org/000/564/122/scalable_compositional_reachability_analysis_of_real_time_concurrent_systems.pdf

[54] A. Rakow, "Slicing Petri Nets with an Application to Workflow Verification," in *SOFSEM 2008: Theory and Practice of Computer Science*, V. Geffert, J. Karhumäki, A. Bertoni, B. Preneel, Pavol Návrat, and M. Bieliková, Eds. Novy Smokovec, Slovakia: Springer Berlin Heidelberg, 2008, pp. 436–447. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-540-77566-9_38

[55] M. Llorens, J. Oliver, J. Silva, S. Tamarit, and G. Vidal, "Dynamic Slicing Techniques for Petri Nets," *Electronic Notes in Theoretical Computer*

Science, vol. 223, pp. 153–165, Dec. 2008. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S1571066108005008

[56] S. Christensen and L. Petrucci, "Modular Analysis of Petri Nets," *The Computer Journal*, vol. 43, no. 3, pp. 224–242, 2000. [Online]. Available: http://comjnl.oxfordjournals.org/content/43/3/224.abstract

[57] M. Notomi and T. Murata, "Hierarchical Reachability Graph of Bounded Petri Nets for Concurrent-Software Analysis," *IEEE Transactions on Software Engineering*, vol. 20, no. 5, pp. 325–336, 1994.

[58] P. Buchholz and P. Kemper, "Hierarchical Reachability Graph Generation for Petri Nets," *Formal Methods in System Design*, vol. 21, pp. 281–315, 2002.

[59] A. Mazurkiewicz, "Compositional Semantics of Pure Place/Transition Systems," in *Advances in Petri Nets*, vol. 340. Springer-Verlag, 1988, pp. 307–330.

[60] P. Sobociński, "Representations of Petri net Interactions," in *21st International Conference on Concurrency Theory*, F. Laroussinie and P. Gastin, Eds. Springer Berlin Heidelberg, 2010, pp. 554–568. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-15375-4_38

[61] R. Bruni, H. Melgratti, U. Montanari, and P. Sobociński, "Connector algebras for C/E and P/T nets' Interactions," *Logical Methods in Computer Science*, vol. 9, no. 3, Sep. 2013. [Online]. Available: http://eprints.soton.ac.uk/339065/http://www.lmcs-online.org/ojs/viewarticle.php?id=1189

[62] E. Best, R. Devillers, and J. G. Hall, "The Box Calculus: a New Causal Algebra with Multi-label Communication," *Advances in Petri Nets*, pp. 21–69, 1992. [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-55610-9_167http://www.springerlink.com/index/p8g077k2p883013r.pdf

[63] M. Koutny and E. Best, "Operational and Denotational Semantics for the Box Algebra," *Theoretical Computer Science*, vol. 211, no. 1-2, pp. 1–83, 1999.

[64] E. Best, M. Koutny, and U. Hildesheim, "A Refined View of the Box Algebra," in *Application and Theory of Petri Nets*. Springer-Verlag, 1995, pp. 1–20.

[65] W. Reisig, "Simple Composition of Nets," in *Application and Theory of Petri Nets*. Springer Berlin Heidelberg, 2009, pp. 23–42.

[66] P. Baldan, A. Corradini, H. Ehrig, and R. Heckel, "Compositional Modeling of Reactive Systems Using Open Nets," in *Concurrency Theory*, vol. 2154. Springer-Verlag Berlin Heidelberg, 2001, pp. 502–518. [Online]. Available: http://dx.doi.org/10.1007/3-540-44685-0_34

[67] L. Priese and H. Wimmel, "A Uniform Approach to True-concurrency and Interleaving Semantics for Petri Nets," *Theoretical Computer Science*, vol. 206, pp. 219–256, 1998.

[68] P. Katis, N. Sabadini, and R. F. C. Walters, "Span (Graph): A Categorical Algebra of Transition Systems," in *Algebraic Methodology and Software Technology*, M. Johnson, Ed. Sydney, Australia: Springer Berlin Heidelberg, 1997, pp. 307–321. [Online]. Available: http://link.springer.com/chapter/10.1007/BFb0000479

[69] ——, "Representing Place/Transition Nets in Span(Graph)," in *Algebraic Methodology and Software Technology*, M. Johnson, Ed. Springer Berlin Heidelberg, 1997, pp. 322–336. [Online]. Available: http://link.springer.com/chapter/10.1007/BFb0000480

[70] R. Bruni, H. Melgratti, and U. Montanari, "A Connector Algebra for P/T Nets Interactions," in *CONCUR 2011 - Concurrency Theory*, J.-P. Katoen and B. König, Eds., no. Prin 2008. Aachen, Germany: Springer Berlin Heidelberg, 2011, pp. 312–326. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-23217-6_21

[71] J. Esparza and C. Schröter, "Net Reductions for LTL Model-checking," in *Correct Hardware Design and Verification Methods*. Springer Berlin Heidelberg, 2001, pp. 310–324. [Online]. Available: http://www.springerlink.com/index/26vhetvhhg2drtkh.pdf

[72] G. Berthelot, "Checking Properties of Nets using Transformations," in *Advances in Petri Nets*. Springer Berlin Heidelberg, 1985, pp. 19–40.

[73] S. Haddad and J.-F. Pradat-Peyre, "New Efficient Petri Nets Reductions for Parallel Programs Verification," *Parallel Processing Letters*, vol. 16, no. 1, pp. 101–116, 2006.

[74] A. Rakow, "Decompositional Petri Net Reductions," in *Integrated Formal Methods*, M. Leuschel and H. Wehrheim, Eds. Düsseldorf, Germany: Springer Berlin Heidelberg, 2009, pp. 352–366.

[75] E.-R. Olderog, "Strong Bisimilarity on Nets: A New Concept for Comparing Net Semantics," in *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*. Springer Berlin Heidelberg, 1989, pp. 549–573.

[76] C. Autant, Z. Belmesk, and P. Schnoebelen, "Strong Bisimilarity on Nets Revisited," in *Parallel Architectures and Languages Europe*. Springer Berlin Heidelberg, 1991, pp. 717–734.

[77] C. Autant, W. Pfister, and P. Schnoebelen, "Place Bisimulations for the Reduction of Labeled Petri nets with Silent Moves," in *International Conference on Computing and Information*, 1994, pp. 230–246.

[78] P. Schnoebelen and N. Sidorova, “Bisimulation and the Reduction of Petri Nets,” in *Application and Theory of Petri Nets*, vol. 1825, 2000, pp. 409–423. [Online]. Available: http://www.springerlink.com/content/y1m5wx4btvxnv20d/

[79] M. Nielsen, G. Plotkin, and G. Winskel, “Petri Nets, Event Structures and Domains, Part I,” *Theoretical Computer Science*, vol. 13, no. 1, pp. 85–108, Jan. 1981. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/0304397581901122

[80] J. Engelfriet, “Branching Processes of Petri nets,” *Acta Informatica*, vol. 28, no. 6, pp. 575–591, 1991. [Online]. Available: http://link.springer.com/article/10.1007/BF01463946

[81] K. L. McMillan, “Using Unfoldings to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits,” in *Computer Aided Verification*, G. von Bochmann and D. K. Probst, Eds. Springer Berlin Heidelberg, 1992, pp. 164–177. [Online]. Available: http://link.springer.com/content/pdf/10.1007/3-540-56496-9_14.pdf

[82] J. Esparza, S. Römer, and W. Vogler, “An Improvement of McMillan’s Unfolding Algorithm,” *LNCS - Tools and Algorithms for the Construction and Analysis of Systems*, vol. 1055, pp. 87–106, 1996. [Online]. Available: http://www.springerlink.com/index/77863336g3776g88.pdf

[83] K. Heljanko, “Minimizing Finite Complete Prefixes,” in *Workshop on Concurrency, Specification & Programming*, P. Starke, H.-S. Nguyen, L. Czaja, and H.-D. Burkhard, Eds. Warsaw, Poland: Warsaw University, 1999, pp. 83–95. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.3131

[84] V. Khomenko and M. Koutny, “Towards an Efficient Algorithm for Unfolding Petri Nets,” in *CONCUR 2001 - Concurrency Theory*, K. G. Larsen and M. Nielsen, Eds. Springer Berlin Heidelberg, 2001, pp. 366–380. [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-44685-0_25

[85] K. Heljanko, V. Khomenko, and M. Koutny, “Parallelisation of The Petri net Unfolding Algorithm,” in *Tools and Algorithms for the Construction and Analysis of Systems*, J.-P. Katoen and P. Stevens, Eds. Grenoble, France: Springer Berlin Heidelberg, 2002, pp. 371–385. [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-46002-0_26

[86] B. Bonet, P. Haslum, S. Hickmott, and S. Thiébaux, “Directed Unfolding of Petri Nets,” in *Transactions on Petri Nets and Other Models of Concurrency*, K. Jensen, W. M. P. Aalst, and J. Billington, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 172–198.

[87] V. Khomenko, M. Koutny, and W. Vogler, "Canonical Prefixes of Petri net Unfoldings," *Acta Informatica*, vol. 40, no. 2, pp. 95–118, Oct. 2003. [Online]. Available: http://link.springer.com/10.1007/s00236-003-0122-y

[88] C. Neumair, J. Desel, and G. Junhás, "Finite Unfoldings of Unbounded Petri Nets," in *Application and Theory of Petri Nets*. Springer Berlin Heidelberg, 2004, pp. 157–176.

[89] J. Esparza and C. Schröter, "Unfolding Based Algorithms for the Reachability Problem," *Fundamenta Informaticae*, vol. 47, no. 3-4, pp. 231–245, 2001. [Online]. Available: http://iospress.metapress.com/index/wc92whqm04tx5q0u.pdf

[90] K. Heljanko, "Using Logic Programs with Stable Model Semantics to Solve Deadlock and Reachability Problems for 1-Safe Petri Nets," *Fundamenta Informaticae*, vol. 37, pp. 247–268, 1999. [Online]. Available: http://iospress.metapress.com/index/652W44V7H34816N0.pdf

[91] K. Heljanko and J. Esparza, "A New Unfolding Approach to LTL Model Checking," in *International Colloquium On Automata, Languages and Programming1*. Springer Berlin Heidelberg, 2000, pp. 475–486.

[92] F. Wallner, "Model Checking LTL Using Net Unfoldings," in *Computer Aided Verification*, A. J. Hu and M. Y. Vardi, Eds., no. 342. Springer Berlin Heidelberg, 1998, pp. 207–218. [Online]. Available: http://link.springer.com/chapter/10.1007/BFb0028746

[93] J. Esparza and K. Heljanko, *Unfoldings: A Partial-Order Approach to Model Checking*, 2008.

[94] J. Esparza, "A False History of True Concurrency: from Petri to Tools," *Model Checking Software*, 2010. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-16164-3_13

[95] V. Khomenko, A. Kondratyev, M. Koutny, and W. Vogler, "Merged Processes: a New Condensed Representation of Petri net Behaviour," *Acta Informatica*, vol. 43, no. 5, pp. 307–330, Oct. 2006. [Online]. Available: http://link.springer.com/10.1007/s00236-006-0023-y

[96] ——, "Merged Processes — A New Condensed Representation of Petri Net Behaviour," in *CONCUR*, M. Abadi and L. de Alfaro, Eds. Springer Berlin Heidelberg, 2005, pp. 338–352.

[97] V. Khomenko and A. Mokhov, "An Algorithm for Direct Construction of Complete Merged Processes," in *32nd International Conference, PETRI NETS*, L. Petrucci and L. M. Kristensen, Eds. Springer Berlin Heidelberg,

2011, pp. 89–108. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-21834-7_6

[98] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, "Symbolic Model Checking: 10ˆ20 States and Beyond," *Information and Computation*, vol. 98, no. 2, pp. 142–170, Jun. 1992. [Online]. Available: http://link.springer.com/chapter/10.1007/978-1-4615-3190-6_3http://linkinghub.elsevier.com/retrieve/pii/089054019290017A

[99] O. Coudert, C. Berthet, and J. C. Madre, "Verification of Synchronous Sequential Machines Based on Symbolic Execution," in *International Workshop on Automatic Verification Methods for Finite State Systems*. Springer-Verlag, 1990, pp. 365–373. [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-52148-8_30

[100] G. H. Mealy, "A Method for Synthesizing Sequential Circuits," *Bell System Technical Journal*, vol. 34, pp. 1045–1079, 1955. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/j.1538-7305.1955.tb03788.x/abstract

[101] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra, "Automatic Verification of Sequential Circuits Using Temporal Logic," pp. 1035–1044, 1986.

[102] R. E. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," *Computers, IEEE Transactions on*, vol. C-35, no. 8, pp. 677–691, 1986. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1676819

[103] H. R. Andersen, "An Introduction to Binary Decision Diagrams," Technical University of Denmark, Tech. Rep. October 1997, 1997. [Online]. Available: http://www-sst.informatik.tu-cottbus.de/~db/doc/People/Anderson/An_Introduction_to_Binary_Decision_Diagrams.bdd.gz.ps

[104] R. Drechsler and D. Sieling, "Binary Decision Diagrams in Theory and Practice," *International Journal on Software Tools for Technology Transfer*, vol. 3, no. 2, pp. 112–136, 2001. [Online]. Available: http://link.springer.com/article/10.1007/s100090100056

[105] E. Pastor, O. Roig, J. Cortadella, and R. Badia, "Petri Net Analysis Using Boolean Manipulation," in *Application and Theory of Petri Nets*. Springer Berlin Heidelberg, 1994, pp. 416–435. [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-58152-9_23

[106] E. Pastor and J. Cortadella, "Efficient Encoding Schemes for Symbolic Analysis of Petri nets," in *Design, Automation and Test in Europe*, 1998, pp. 790–795.

[107] E. Pastor, J. Cortadella, and M. A. Peña, "Structural Methods Applied to the Symbolic Analysis of Petri Nets," in *International Conference on Application and Theory of Petri Nets*, 1999, pp. 26–45.

[108] A. Semenov and A. Yakovlev, "Combining Partial Orders and Symbolic Traversal for Efficient Verification of Asynchronous Circuits," in *Design Automation Conference*, 1995, pp. 567–573.

[109] A. Miner and G. Ciardo, "Efficient Reachability Set Generation and Storage Using Decision Diagrams," in *Application and Theory of Petri Nets*. Springer Berlin Heidelberg, 1999, pp. 6–25. [Online]. Available: http://www.springerlink.com/index/M9F5N8017DX8XULE.pdf

[110] A. Srinivasan, T. Ham, S. Malik, and R. K. Brayton, "Algorithms for Discrete Function Manipulation," in *IEEE International Conference on Computer-Aided Design*. IEEE, 1990, pp. 92–95.

[111] G. Ciardo, G. Luettgen, and R. Siminiceanu, "Efficient Symbolic State-Space Construction for Asynchronous Systems," in *Application and Theory of Petri Nets*. Springer Berlin Heidelberg, 2000, pp. 103–122. [Online]. Available: http://hdl.handle.net/2060/20000011947

[112] K. Strehl and L. Thiele, "Interval Diagram Techniques for Symbolic Model Checking of Petri nets," in *Design, Automation and Test in Europe*. IEEE, 1999, pp. 756–757.

[113] A. Tovchigrechko, "Model Checking using Interval Decision Diagrams," PhD, BTU Cottbus, 2008.

[114] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang, "Spectral Transforms for Large Boolean Functions with Applications to Technology Mapping," in *Design Automation, 1993. 30th Conference on*, 1993, pp. 54–60. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1600192

[115] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Algebraic Decision Diagrams and their Applications," *Proceedings of 1993 International Conference on CoCmputer Aided Design (ICCAD)*, pp. 188–191, 1993. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=580054

[116] M. Fujita, P. McGeer, and J.-Y. Yang, "Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation," *Formal Methods in System Design*, vol. 169, pp. 149–169, 1997. [Online]. Available: http://link.springer.com/article/10.1023/A:1008647823331

[117] F. Wang and M. Kwiatkowska, "An MTBDD-Based Implementation of Forward Reachability for Probabilistic Timed Automata," in *Automated Technology for Verification and Analysis*, D. A. Peled and Y.-K. Tsay, Eds. Springer Berlin Heidelberg, 2005, pp. 385–399. [Online]. Available: http://link.springer.com/chapter/10.1007/11562948_29

[118] F. Ciesinski, C. Baier, M. Größ er, and D. Parker, "Generating compact MTBDD-representations from probmela specifications," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, K. Havelund, R. Majumdar, and J. Palsberg, Eds., vol. 5156 LNCS. Los Angeles, CA, USA: Springer Berlin Heidelberg, 2008, pp. 60–76. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-540-85114-1_7

[119] B. Bollig and I. Wegener, "Improving the Variable Ordering of OBDDs Is NP-complete," *IEEE Transactions on Computers*, vol. 45, no. 9, pp. 993–1002, 1996. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=537122

[120] H. Fujii, G. Ootomo, and C. Hori, "Interleaving Based Variable Ordering Methods for Ordered Binary Decision Diagrams," in *International Conference on Computer Aided Design*. Santa Clara, CA, USA: IEEE Comput. Soc. Press, 1993, pp. 38–41. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=580028

[121] E. Felt, G. York, R. Brayton, and A. Sangiovanni-Vincentelli, "Dynamic Variable Reordering for BDD Minimization," in *Design Automation Conference*. Hamburg, Germany: IEEE Comput. Soc, 1993, pp. 130–135. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=410627

[122] R. Ebendt, W. Günther, and R. Drechsler, "Combining Ordered Best-First Search With Branch and Bound for Exact BDD Minimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 10, pp. 1515–1529, 2005. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1512370

[123] S. J. Friedman and K. J. Supowit, "Finding the Optimal Variable Ordering for Binary Decision Diagrams," in *Proceedings of the 24th ACM/IEEE Design Automation Conference*, A. O'Neill and D. Thomas, Eds. ACM New York, 1987, pp. 348–356.

[124] R. Drechsler, N. Drechsler, and W. Gunther, "Fast Exact Minimization of BDDs," in *Design and Automation Conference*, M. J. Irwin, Ed. San Francisco, CA, USA: ACM New York, 1998, pp. 200–205. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=724466

[125] R. Ebendt, G. Fey, and R. Drechsler, *Advanced BDD Optimization*. Springer, 2005.

[126] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *Tools and Algorithms for Construction and Analysis of Systems*,

no. 97. Springer Berlin Heidelberg, 1999, pp. 193–207. [Online]. Available: http://www.springerlink.com/index/VF286K9MQ0JP05DH.pdf

[127] S. Ogata, T. Tsuchiya, and T. Kikuno, "SAT-Based Verification of Safe Petri Nets," in *Automated Technology for Verification and Analysis*. Springer-Verlag Berlin Heidelberg, 2004, pp. 79–92.

[128] K. Heljanko, "Bounded Reachability Checking with Process Semantics," in *International Conference on Concurrency Theory*. Springer-Verlag Berlin Heidelberg, 2001, pp. 218–232.

[129] E. Best and R. Devillers, "Sequential and Concurrent Behaviour in Petri net Theory," pp. 87–136, 1987.

[130] S. Melzer and S. Römer, "Deadlock Checking Using Net Unfoldings," in *Computer Aided Verification*, vol. 1254. Springer Berlin Heidelberg, 1997, pp. 352 – 363. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.52.3517

[131] V. Khomenko and M. Koutny, "LP Deadlock Checking Using Partial Order Dependencies," in *CONCUR*. Springer-Verlag Berlin Heidelberg, 2000, pp. 410–425. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.25.8272&rep=rep1&type=pdfhttp://link.springer.com/chapter/10.1007/3-540-44618-4_4

[132] ——, "Verification of bounded Petri nets using Integer Programming," *Formal Methods in System Design*, vol. 30, no. 2, pp. 143–176, 2007.

[133] J. Esparza and S. Melzer, "Verification of Safety Properties Using Integer Programming: Beyond the State Equation," *Formal Methods in System Design*, vol. 16, no. 2, pp. 159–189, 2000.

[134] J. Desel, "Basic Linear Algebraic Techniques for Place/Transition Nets," in *Lectures on Petri Nets I: Basic Models*. Springer Berlin Heidelberg, 1998, pp. 257–308.

[135] M. Silva, E. Terue, and J. M. Colom, "Linear Algebraic and Linear Programming Techniques for the Analysis of Place/Transition net Systems," *Lectures on Petri Nets I: Basic Models*, vol. 1491, pp. 309–373, 1998. [Online]. Available: papers2://publication/uuid/48B69E34-629D-4ACC-864D-FF15690206B0

[136] J. Desel and W. Reisig, "Place/Transition Petri Nets," in *Lectures on Petri Nets I: Basic Models*, 1988, vol. 1491, pp. 122–173.

[137] J. Engelfriet and G. Rozenberg, "Elementary Net Systems," in *Lectures on Petri Nets I: Basic Models*, ser. Lecture Notes in Computer Science, G. Rozenberg

and W. Reisig, Eds. Springer Berlin Heidelberg, 1998, vol. 1491, pp. 12–121. [Online]. Available: http://link.springer.com/10.1007/3-540-65306-6

[138] P. Thiagarajan, "Elementary Net Systems," *Petri Nets: Central Models and Their Properties*, vol. 254, pp. 26–59, 1987. [Online]. Available: http://link.springer.com/chapter/10.1007/BFb0046835

[139] J. L. Peterson, "Petri Nets," *ACM Computing Surveys*, vol. 9, no. 3, pp. 223–252, Sep. 1977. [Online]. Available: http://portal.acm.org/citation.cfm?doid=356698.356702

[140] P. Sobociński, "Nets, Relations and Linking Diagrams," in *Algebra and Coalgebra in Computer Science*, R. Heckel and S. Milius, Eds. Springer Berlin Heidelberg, 2013, pp. 282–298. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-40206-7_21

[141] L. Bernardinello and F. De Cindio, "A Survey of Basic Net Models and Modular Net Classes," *Advances in Petri Nets 1992*, vol. 609, pp. 304–351, 1992. [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-55610-9_177

[142] S. Christensen and N. D. Hansen, "Coloured Petri Nets Extended With Place Capacities, Test Arcs and Inhibitor Arcs," in *Application and Theory of Petri Nets*, M. A. Marsan, Ed., no. 5. Springer Berlin Heidelberg, 1993, pp. 186–205. [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-56863-8_47

[143] P. Selinger, "A survey of graphical languages for monoidal categories," *New Structures for Physics*, pp. 1–63, Aug. 2009. [Online]. Available: http://arxiv.org/abs/0908.3347

[144] S. Lack, "Composing PROPs," *Theory and Applications of Categories*, vol. 13, no. 9, pp. 147–163, 2004.

[145] J. C. Corbett, "Evaluating Deadlock Detection Methods for Concurrent Software," *IEEE Transactions on Software Engineering*, vol. 22, no. 3, pp. 161–180, Mar. 1996. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=489078

[146] J. Rathke, P. Sobociński, and O. Stephens, "Decomposing Petri nets," *arXiv:1304.3121*, 2013. [Online]. Available: http://arxiv.org/abs/1304.3121

[147] P. A. Abdulla, S. P. Iyer, and A. Nylén, "SAT-Solving the Coverability Problem for Petri Nets," *Formal Methods in System Design*, vol. 24, no. 1, pp. 25–43, Jan. 2004. [Online]. Available: http://link.springer.com/10.1023/B:FORM.0000004786.30007.f8

[148] T. Coquand, "Pattern Matching with Dependent Types," in *Workshop on Types for Proofs and Programs*, 1992, pp. 66–80.

[149] A. Abel and T. Altenkirch, "A Predicative Analysis of Structural Recursion," *Journal of Functional Programming*, vol. 12, no. 01, Jan. 2002. [Online]. Available: http://www.journals.cambridge.org/abstract_S0956796801004191

[150] R. Milner, "A Theory of Type Polymorphism in Programming," *Journal of Computer and System Sciences*, vol. 17, no. 3, pp. 348–375, 1978. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022000078900144

[151] R. Hinze, "Theoretical Pearl: Church Numerals, Twice!" *Journal of Functional Programming*, vol. 15, no. 1, pp. 1–13, Jan. 2005. [Online]. Available: http://www.journals.cambridge.org/abstract_S0956796804005313

[152] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*. ACM New York, 1977, pp. 238–252. [Online]. Available: http://dl.acm.org/citation.cfm?id=512973

[153] B. Pierce, *Types and Programming Languages*, 1st ed. The MIT Press, 2002.

[154] P. Katis, N. Sabadini, and R. F. C. Walters, "Compositional Minimization in Span(Graph): Some Examples," in *Proceedings of the Workshop of the COMETA Project on Computational Metamodels*, 2004, pp. 181–197.

[155] S. Peyton Jones, "The Haskell 98 language and libraries: The revised report," *Journal of Functional Programming*, vol. 13, no. 1, pp. 0–255, Jan 2003.

[156] E. F. Moore, "Gedanken-experiments on Sequential Machines," *Automata studies*, vol. 34, pp. 129–153, 1956.

[157] J. E. Hopcroft, "An N Log N Algorithm for Minimizing States in a Finite Automaton," in *International Symposium on The Theory of Machines and Computations*. AP, 1971, pp. 189–196.

[158] J. a. Brzozowski, "Canonical Regular Expressions and Minimal State Graphs for Definite Events," in *Mathematical Theory of Automata*, 1962, pp. 529—-561. [Online]. Available: http://www.diku.dk/hjemmesider/ansatte/henglein/papers/brzozowski1962.pdf

[159] H. Björklund and W. Martens, "The Tractability Frontier for NFA Minimization," *Journal of Computer and System Sciences*, vol. 78, no. 1, pp. 198–210, Jan. 2011. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0022000011000456

[160] G. Gramlich and G. Schnitger, "Minimizing NFAs and Regular Expressions," *Journal of Computer and System Sciences*, vol. 73, no. 6, pp. 908–923, Sep. 2007. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0022000006001735

[161] R. Mayr and L. Clemente, “Advanced Automata Minimization,” in *Symposium on Principles of Programming Languages*. Rome, Italy: ACM, 2013, pp. 63–74. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2429069.2429079

[162] A. Arnold, A. Dicky, and M. Nivat, “A Note about Minimal Non-Deterministic Automata,” *Bulletin of the EATCS*, vol. 47, pp. 166–169, 1992.

[163] F. Bonchi and D. Pous, “Checking NFA Equivalence with Bisimulations up to Congruence,” in *Symposium on Principles of Programming Languages*. Rome, Italy: ACM, 2013, pp. 457–468.

[164] K. Schmidt, “LoLA: A Low Level Analyser,” in *Application and Theory of Petri Nets 2000*, ser. Lecture Notes in Computer Science, M. Nielsen and D. Simpson, Eds., vol. 1825. Berlin, Heidelberg: Springer Berlin Heidelberg, Jun. 2000, pp. 465–474.

[165] C. Rodríguez and S. Schwoon, “Cunf: A Tool for Unfolding and Verifying Petri Nets with Read Arcs,” in *Automated Technology for Verification and Analysis*, 2013, pp. 492–495. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-319-02444-8_42

[166] M. Heiner, C. Rohr, and M. Schwarick, “MARCIE - Model checking and reachability analysis done efficiently,” in *Application and Theory of Petri Nets*, 2013, pp. 389–399.

[167] F. Kordon, H. Garavel, L.-M. Hillah, F. Hulin-Hubard, A. Linard, M. Beccuti, S. Evangelista, A. Hamez, N. Lohmann, E. L. And, E. Paviot-Adet, C. Rodriguez, C. Rohr, and J. Srba, “HTML results from the Model Checking Contest @ Petri Nets (2014 edition),” 2014. [Online]. Available: http://mcc.lip6.fr/2014

[168] M. Koutny and V. Khomenko, “Linear Programming Deadlock Checking Using Partial Order Dependencies,” Newcastle University, Tech. Rep., 2000. [Online]. Available: http://www.ncl.ac.uk/computing/research/publication/160655

[169] K. G. Larsen, “A Context Dependent Equivalence Between Processes,” *Theoretical Computer Science*, vol. 49, pp. 185–215, 1987.