Verification of Infinite-State Systems and Machine Learning

Igor Khmelnitsky

université
PARIS-SACLAY

Verification of Infinite-State Systems and Machine Learning

Thèse de doctorat de l'Université Paris-Saclay

École doctorale n° 580, Sciences et Technologies de l'Information et de la Communication (STIC)
Spécialité de doctorat: Informatique
Unité de recherche: Université Paris-Saclay, CNRS, ENS Paris-Saclay, Laboratoire Méthodes Formelles, 91190, Gif-sur-Yvette, France.
Référent: : ENS Paris-Saclay

Thèse présentée et soutenue à l'ENS Paris-Saclay, le 27.01.22, par

Igor KHMELNITSKY

Composition du jury:

Patricia Bouyer-Decitre Directrice de recherche, LMF, CNRS, ENS Paris-Saclay, Université Paris-Saclay, France.	Présidente
Javier Esparza Professeur, Technical University of Munich, Germany.	Rapporteur
Ranko Lazic Professeur, DIMAP, Department of Computer Science, University of Warwick, England.	Rapporteur
Dana Fisman Professeure, Ben-Gurion University, Israel.	Examinatrice
Laure Petrucci Professeure, LIPN, CNRS UMR 7030, Université Sorbonne Paris Nord, France.	Examinatrice
Pierre-Alain Reynier Professeur, LIS, CNRS, Aix Marseille Université, Université de Toulon, France.	Examinateur
Alain Finkel Professeur, LMF, CNRS, ENS Paris-Saclay, IUF, Université Paris-Saclay, France.	Codirecteur
Serge Haddad Professeur, LMF, CNRS, ENS Paris-Saclay, Inria, Université Paris-Saclay, France.	Codirecteur

Thèse de doctorat

NNT: 2022UPASG001

Abstract

This thesis consists of three parts. The first one is devoted to the verification of Petri nets, the second one to the verification of recursive Petri nets which extend Petri nets, and the final one aims at combining active learning and verification.

A Petri net can be analyzed by computing and studying its Clover, that is, the canonical representation of the downward over approximation of its reachability set. Using the Karp-Miller algorithm one can compute the Clover, but this algorithm is very inefficient and moreover, its original proof of correctness is not satisfying. Many variations of the original Karp-Miller algorithm computing the clover exist, but some are incomplete, others introduced an unknown supplementary memory size (possibly Ackermannian) and proofs are often heavy. Our first contribution is the design of a complete algorithm in such a way that we can theoretically bound the additional memory requirements. The key idea of this algorithm is the introduction of a new concept, called acceleration. More precisely, using accelerations, we were able:

1. to simplify the proof of correctness of the Karp-Miller algorithm;
2. to present the first simple modification of the original but incomplete Minimal Coverability Tree algorithm;
3. to prove that the supplementary memory needed by our algorithm is elementary (2-EXPSPACE);
4. to implement a prototype MinCov, showing experimentally that it is the most efficient one compared to other tools computing the Clover.

In the early two-thousands, Recursive Petri nets (RPN) have been introduced in order to model distributed planning of multiagent systems for which counters and recursivity were necessary. Although RPN strictly extend Petri nets and context-free grammars, most of the usual problems (reachability, termination, etc.) were shown to be decidable. For almost all other models extending Petri nets and context-free grammars, the complexity of coverability and termination are unknown or strictly larger than EXPSPACE. In contrast, we establish here that for RPN, the coverability, termination, boundedness and finiteness problems are EXPSPACE-complete as for Petri net. While having a great expressive power,

RPN suffer several modeling limitations. We introduce Dynamic Recursive Petri nets (DRPN) which address these issues, extending the expressiveness of RPN. This model generalizes almost all previous known models, which extend the Petri net and keep the coverability problem decidable. Thus, we establish that the coverability problem is decidable for DRPN.

For active learning and formal methods, our work focuses on Angluin's L^* algorithm. Angluin's algorithm learns the minimal deterministic finite automaton (DFA) of a regular language using membership and equivalence queries. Its probabilistic approximately correct (PAC) version substitutes an equivalence query by a set of random membership queries. Thus, it can be applied to any kind of device and may be viewed as synthesizing an automaton from observations of the device. We are interested in how the PAC version behaves for devices which are obtained from a DFA by introducing some noise. More precisely, we study whether the algorithm reduces the noise, producing a DFA closer to the original one than the noisy device. We found that the reduction of the noise strongly depends on the type of noise and its amount. Moreover, we use this algorithm to develop a property-directed approach for verification of recurrent neural networks (RNNs). It learns a DFA as a surrogate model from a given RNN, which is then analyzed using model checking as a verification technique. We show that this not only allows us to discover small counterexamples fast, but also to generalize them by pumping towards faulty flows, hinting at the underlying error in the RNN.

Résumé

Cette thèse est découpée en trois parties. La première est consacrée à la vérification des réseaux de Petri, la seconde à la vérification des réseaux de Petri récursifs qui étendent les réseaux de Petri et la dernière vise à combiner l'apprentissage actif et la vérification.

Un réseau de Petri peut être analysé en calculant et étudiant son *Clover*, la représentation canonique de la sur-approximation vers le bas de son ensemble d'accessibilité. A l'aide de l'algorithme de Karp-Miller on peut calculer le Clover, mais cet algorithme est très inefficace et de plus sa preuve originelle de correction n'est pas satisfaisante. Il y a de nombreuses variantes de cet algorithme mais certaines sont incomplètes et d'autres requièrent une mémoire additionnelle de taille potentiellement ackermannienne. Enfin les preuves de correction sont souvent intriquées. Notre première contribution est la conception d'un algorithme complet incluant une borne théorique sur la taille de la mémoire additionelle. L'idée clef de cet algorithme est l'introduction d'un nouveau concept, appelé accélération. Plus précisément à l'aide des accélérations, nous avons pu:

1. simplifier la preuve de correction de l'algorithme de Karp-Miller;
2. de présenter la première modification simple de l'algorithme incomplet de construction du "Minimal Coverability Tree";
3. de prouver que la mémoire supplémentaire requise par notre algorithme est élémentaire (2-EXPSPACE);
4. de développer un prototype MinCov et de montrer expérimentalement qu'il est l'outil le plus efficace parmi ceux qui calculent le Clover.

Au début des années 2000, les réseaux de Petri récursifs (RPN) ont été introduits en vue de la modélisation et de l'analyse de la planification distribuée de systèmes multi-agents pour lesquels la présence de compteurs et la recursivité sont nécessaires. Bien que les RPN étendent strictement les réseaux de Petri et les grammaires algébriques, la plupart des problèmes usuels (accessibilité, terminaison, etc.) restent décidables. Pour presque tous les modèles incluant les réseaux de Petri et les grammaires algébriques, la complexité des problèmes de la couverture et de la terminaison est inconnue ou strictement plus grande que EXPSPACE. Ici,

nous établissons que les problèmes de couverture, terminaison, caractère borné et finitude des RPN sont EXPSPACE-complets comme ceux des réseaux de Petri. Bien qu'ayant un grand pouvoir d'expression, les RPN souffrent de plusieurs limitations en terme de modélisation. Aussi nous introduisons les réseaux de Petri récursifs dynamiques (DRPN) qui répondent à ces limitations et par conséquent étendent le pouvoir d'expression des RPN. Les DRPN généralisent presque toutes les extensions de réseaux de Petri pour lesquelles le problème de couverture est décidable. Nous démontrons alors que le problème de couverture reste décidable pour les DRPN.

Dans la troisième partie, notre travail se concentre sur l'algorithme L* d'Angluin. Cet algorithme apprend l'automate fini déterministe (DFA) minimal d'un langage régulier à l'aide de questions d'appartenance et d'équivalence de langages. Sa version probabilistiquement approximativement correcte (PAC) remplace une question d'équivalence par un ensemble de questions aléatoires d'appartenance. Nous avons étudié comment la PAC version se comporte pour des machines qui sont obtenues en "bruitant" un DFA et si l'algorithme réduit le bruit. Nous établissons que la réduction du bruit dépend fortement de la nature du bruit et de sa quantité. De plus, nous utilisons cet algorithme pour développer une approche de vérification des réseaux neuronaux récurrents (RNN). Un DFA est appris comme une abstraction d'un RNN puis analysé à l'aide de techniques de "model checking". Nous établissons deux avantages de cette approche : lorsque la propriété n'est pas vérifiée les contre-exemples exhibés sont de petite taille et susceptibles d'être généralisés en un patron d'erreur mettant en évidence la nature de la faute du RNN.

Acknowledgments

Acknowledgments Here

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Verification

Verification, the act of proving or disproving the correctness of software and hardware, is an essential part of development and research in the academia and in the industry. Widespread engineering techniques such as testing may be useful, but cannot rule out completely the existence of bugs. This poses grave dangers in critical systems, such as medical systems, electric grids, etc. For this reason formal verification was introduced and is now widely adopted by companies such as Intel, IBM, Facebook, Google, etc. Formal verification allows proving the correctness of software or hardware using mathematical tools. One of the approaches for formal verification is model checking, where we model the underlying system in terms of a mathematical model, and verify its properties using the mathematical features of the model.

Model-checking has been very successful in the verification of finite-state systems, see e.g., [18]. However, assuming that systems have finitely many states is very restrictive. Therefore, the study of infinite-state systems has gained popularity. With their additional capabilities, infinite-state systems have an increased complexity, which made the study of infinite-state system a very active one. The infinite-state space comes from a huge variety of sources, such as systems with variables that belong to the natural or even real numbers, recursive systems which may lead to infinite stacks, parameterized systems which can describe infinitely many finite systems depending on the initial state. Many abstract models have been introduced in order to model these infinite-state systems, e.g., WSTS [63], timed automata [112], pushdown automata [105]...

Cover of Petri nets

In recent years, multiple advancements led to rapid growth of concurrent and distributed systems. These types of systems are even harder to verify using the usual engineering solutions, since their bugs tend to occur with low probability and may remain undetected for long stretches of time.

Petri nets are a model specifically designed as a modeling language for the description of concurrent and distributed systems. Petri nets do not only have a very intuitive graphical representation, but they also have formal semantics which give rise to many analysis techniques. Petri nets are used in a wide variety of domains, for example:

- *computer science*: verification of multithreaded programs [86], communication protocols [79], hardware designs [159], concurrent object-oriented programming [3], etc;
- *biology*: modeling and analysis of molecular networks [18];
- *business process management*: modeling notations and workflow management systems [154];
- *process mining*: creating process models as abstract representations of event logs [153].

The most sought after property to verify for Petri nets is the reachability problem: given an initial configuration of the system, is there a sequence of actions that leads to a target configuration? In other words, can a given system, with some initial conditions, reach a bug/desired-state. Unfortunately, it has been shown that this property is theoretically very hard to verify (it is Ackermannian-complete [104, 41]). However, some techniques exist for *bounded* Petri nets (finite-state systems) which most often work in practice, e.g., partial-order reductions, symmetry-based reductions, etc; and some techniques exist for sub-problems, e.g., backward algorithm and coverability trees for the so-called coverability problem. Unfortunately, these techniques are not always well translated to the reachability problem on *unbounded* Petri nets (infinite-state systems) which need their own techniques.

One of the most studied property in unbounded Petri nets is coverability: given an initial configuration of the system, is there a sequence of actions that leads to a configuration *larger* than the target configuration? It has been studied for several reasons: (1) many properties like mutual exclusion, safety, LTL model checking with atomic predicates on transitions, or control-state reachability, reduce to coverability, (2) the coverability problem is EXPSPACE-complete, and (3) there exist efficient prototypes and numerous case studies. To solve the coverability problem, there are backward and forward algorithms. But these algorithms do not address relevant problems like the repeated coverability problem, the boundedness

problem and regularity of the traces. However, these problems are EXPSPACE-complete [45, 24] and are also decidable using the (possibly Ackermannian) Karp and Miller algorithm [85] that computes a finite tree labeled by a set of *ω-markings* $C \subseteq \mathbb{N}_\omega^P$ (where $\mathbb{N}_\omega$ is the set of naturals enlarged with an upper bound ω and P is the set of places) such that the reachability set and the finite set C have the same downward closure in $\mathbb{N}^P$. Thus, a marking $\mathbf{m}$ is coverable if there exists some $\mathbf{m}' \geq \mathbf{m}$ with $\mathbf{m}' \in C$. Hence, C can be seen as *one* among all the possible finite representations of the infinite downward closure of the reachability set. This set C allows, for instance, to solve multiple instances of coverability in linear time w.r.t. the size of C avoiding to call many times a costly algorithm. Informally, the Karp and Miller algorithm builds a reachability tree but, in order to ensure termination, substitutes by ω some finite components of a marking of a vertex when some marking of an ancestor is smaller. Unfortunately, C may contain comparable markings, where for coverability properties only the maximal elements are important. The set of maximal elements of C can be defined independently of the Karp and Miller algorithm and was called the *minimal coverability set* in [56] and abbreviated as the *Clover* in the more general framework of Well Structured Transition Systems (WSTS) [58].

Due to the fact that the Karp and Miller algorithm keeps in memory not only the maximal elements, its space requirement may be huge. To address this issue, Alain Finkel designed a new algorithm, the Minimal Coverability Tree algorithm (`MCT`) [56]. This algorithm modifies the Karp and Miller algorithm in such a way that at each step of the algorithm, the set of ω-markings labeling vertices is an antichain. Unfortunately, `MCT` algorithm possesses a subtle bug which makes it generate, sometimes, an under-approximation of the Clover. A counter example for the algorithm can be found in [57, 65]. There were a few attempts (e.g, [152], [135], [64]) to fix this algorithm by keeping more information during the execution. However, the size of this extra memory was not bounded and may even be non-primitive recursive (as compared to `MCT`). Thus, fixing the `MCT` algorithm or designing a new Clover generating algorithm, without keeping a huge amount of extra information, is an important goal. Which leads us to our first problem:

Question 1

Is there an algorithm computing the Clover which is conservative in its memory usage?

Recursive Petri nets

Petri net is a useful formalism for analysis of concurrent programs for many reasons, however they cannot model several widely used patterns in concurrent sys-

tems, in particular recursive features. Therefore, an important research direction consists of extending Petri nets to support new modeling features while still preserving decidability of properties checking.

Such extensions may be partitioned between those whose states are still markings in N^p, and the other ones. One of the simplest extension consists of adding two inhibitor arcs, with which one can simulate a Minsky machine [5], and therefore yield undecidability of most of the verification problems. However, adding a single inhibitor arc preserves the decidability of the reachability, coverability, and boundedness problems [133, 29, 28]. When adding more than three reset arcs to a Petri net, the coverability problem becomes Ackermannian-complete [143] and boundedness undecidable [51]. In ν-Petri nets, the tokens are colored where colors are picked in an infinite domain: their coverability problem is double-Ackermann time complete [101]. In Petri nets with a stack, the reachability problem may be reduced to the coverability problem and both are at least not primitive recursive [40, 98] while their decidability status is still unknown [98]. In branching vector addition systems with states (BVASS) a state is a set of threads with associated markings. A thread either fires a transition as in Petri nets or forks, transferring a part of its marking to the new thread. For BVASS, the reachability problem is also not primitive recursive [100] and its decidability is still an open problem while the coverability and the boundedness problems are 2-EXPTIME-complete [46]. The analysis of subclasses of Petri nets with a stack is an active field of research [13, 113, 43, 165]. However, for none of the above extensions, the coverability and termination problems belong to EXPSPACE.

The *Recursive Petri net* (RPN) model has been introduced to model distributed planning of multiagent systems for which counters and recursivity were necessary for specifying resources and delegation of subtasks [52]. Roughly speaking, a state in an RPN consists of a directed rooted tree of threads, where each thread has a marking with which it plays a token game. The thread can fire three types of transition: (1) *elementary transition*, changing its own marking, (2) *abstract transition*, consuming tokens and creating a new child thread with an initial marking depending only on the fired transition, and (3) *cut transition*, pruning the subtree rooted in the thread firing it and producing tokens in its parent.

While RPNs extend Petri nets and context-free grammars as was shown in [71], the reachability, boundedness and termination remain decidable [71, 72]. This is shown by reducing these properties to reachability problems of Petri nets, so the corresponding algorithms are not primitive recursive. LTL model checking is undecidable for RPN but becomes decidable for the subclass of sequential RPN [73]. The question of the decidability of the coverability problem on RPN was not investigated, partly because no order was designed for its states. There are many orders which can be defined on the states of RPN, but one would want this order

to be at least congruent with the transitions. Which brings us to the following question:

Question 2

Is there a 'good' order on the states of an RPN? If so, is the coverability problem decidable?

While having a great expressive power, RPN suffer from two main limitations:

1. RPNs do not include more general features for transitions like reset arcs, transfer arcs, etc;
2. The initial marking associated to the recursive "call" only depends on the calling transition and not on the current marking of the caller.

Which brings us to our third question:

Question 3

Does there exist a model extending RPN, which: (1) has greater transition features, (2) has a dynamic thread creation, and (3) keeps some interesting properties of the RPN decidable?

Active learning and verification

One of the first goals of this PhD thesis was to connect machine learning and verification of infinite-state systems. In this regard, we had the great luck to join the brand-new project LeaRNNify. The aim of this project is to bring together two different kinds of algorithmic learning, namely grammatical inference and learning of neural networks. This subject was not exactly the research we had initially planned, which was the use of machine learning for verification, but the opportunity to work with leading people from the verification community on verifying recurrent neural networks (RNN) was one we could not pass. That is why one can spot some gap between the previous subjects and the ones we will introduce below.

The problem of learning a language from its finite samples of strings by discovering the corresponding grammar is known as grammar inference, whose significance was initially stated in [145] and an overview of its very first results can be found in [21]. There are generally two types of algorithms for learning deterministic finite automaton (DFA) of a regular language, so-called online and offline algorithms. *Offline algorithms* are given a set of words that are accepted by the automaton (positive examples) and a set of unaccepted words (negative examples), and they

return a minimal automaton which rejects the negative examples and accepts the positive ones. See for example the algorithm from [22]. *Online algorithms* on the other hand can ask further queries whether a word is accepted or unaccepted by the desired automaton. This way the algorithm can ask for relevant examples helping it construct an automaton more faithfully to the desired automaton. A well-known example for an online algorithm is the Angluin L^* algorithm [10].

Angluin's algorithm learns a minimal DFA of a regular language in the presence of a minimally adequate teacher. A *minimally adequate teacher* is a teacher (given a language) which is capable of answering two types of queries, namely *membership* and *equivalence* queries. The algorithm uses these two types of queries to generate the DFA representing the language. There are many implementations and optimizations to Angluin's algorithm, such as the ones in [88, 81]. Two of the disadvantages of this algorithm is that (1) the equivalence query is very expensive computationally, and (2) one wants to learn from a language represented by a model which might not have an equivalence query. One of the ways to face this problem is the probabilistic approximately correct (PAC) version of Angluin's algorithm [10]. This version substitutes the equivalence query by a large enough set of random membership queries. Thus, it can be applied to any kind of device and may be viewed as synthesizing an automaton abstracting the behavior of the device based on observations.

Recurrent neural networks (RNNs) are a state-of-the-art tool to represent and learn sequence-based models. They have applications in time-series prediction, sentiment analysis, and many more. In particular, they are increasingly used in safety-critical applications and act, for example, as controllers in cyber-physical systems [4]. Therefore, as the safty of these RNN is important, there is a growing need for formal verification.

An interesting usage of Angluin's algorithm is extracting a DFA from an RNN approximating its language. More generally in recent years there have been many works developing this type of techniques, i.e., extracting from RNNs state-based formalism such as finite automata, e.g., [14, 115, 114, 122, 123, 156]. These types of extraction are useful for understanding and analyzing RNN, and can be used for verification. One way of using these extractions for verification is the following technique: Given an RNN and some specification it should fulfill, one first extracts a state-based formalism, such as finite automaton, which approximates the black box, then check whether this approximation fulfills the specification. Unfortunately, this kind of technique suffers from two problems: (1) RNN systems can be huge, and therefore the extraction can be a very lengthy process, (2) the algorithm might find a counter example for the approximation on which the RNN and the approximation disagree. This brings us to our the fourth problem:

Question 4

Is there a way to use abstraction for the verification of RNN, avoiding long execution time, and making sure that if the algorithm terminates with a counter example to the abstraction then it is a counter example for the RNN?

Another issue with this type of verification is that RNN are inherently noisy, i.e., some of their output is wrong. This is due to the fact that they are statistically learned. But Angluin's algorithm, as most other learning algorithms in the literature, assumes the correctness of the training data, including the example data such as attributes as well as classification results. However, as seen above, sometimes the noise-free datasets are not available. Some research has been previously carried on the effects of noise on learning, such as: [11, 87, 130]. This made us ask:

Question 5

What does Angluin's PAC Algorithm generate when given a noisy model? Is there noise to which it is robust?

Regular languages are widely used due to their simplicity, but unfortunately they are very restrictive. For example, the family of languages of balanced parentheses is not regular, but it is a subset of context-free languages. Context-free languages are a well-known language family which is used in a large variety of applications, such as compilers and processing natural languages. Therefore, designing an Angluin type of algorithm, learning a context-free language, can give us for example a better approximation of RNN. Unfortunately, learning context-free language seems to be currently out of reach, but we can maybe learn a subclass of context-free languages which is larger (than regular languages). Therefore, we are faced with the following question:

Question 6

Does there exist a model including regular languages and balanced parentheses for which we have an extraction algorithm?

1.1 Summary of contribution

Cover of Petri nets

In Chapter 3, we introduce the concept of *abstraction* as an ω-transition (transitions that may create or demand ω tokens) that mimics the effect of an infinite family of firing sequences of markings w.r.t. coverability. As a consequence, adding abstractions to the net does not modify its coverability set. Moreover, the classical Karp and Miller *acceleration* can be formalized as an abstraction whose incidence on places is either ω or null. The set of accelerations of a net is upward closed, and well-ordered. Hence, there exists a finite subset of minimal accelerations, and we show that the size of all minimal accelerations is bounded by a double exponential using a recent result from [103]. Using our new definition of acceleration, we give a short proof for the correctness of the Karp and Miller algorithm.

Despite the current opinion that *"The flaw is intricate and we do not see an easy way to get rid of it... Thus, from our point of view, fixing the bug of the MCT algorithm seems to be a difficult task"* [65], in Chapter 4 we describe our *simple* modification of MCT which makes it correct. It mainly consists in memorizing discovered accelerations and using them as ordinary transitions. Contrary to *all* existing minimal coverability set algorithms that use an *unknown additional memory* that could be non-primitive recursive, we show that the additional memory required for accelerations is at most doubly exponential. This provides an answer to Question 1.

In Chapter 5, we describe in detail how we developed an optimized prototype of our algorithm. Comparing the prototype performance against other tools producing the Clover on benchmarks either from the literature or random ones, have confirmed that our algorithm requires significantly less memory than the other algorithms and is close to the fastest tool w.r.t. the execution time. Moreover, we show how to combine our algorithm with continuous over approximation techniques from [26], resulting in superior tool solving coverability.

Recursive Petri nets

In Chapter 8, we begin by introducing a quasi-order on states of RPN compatible with the firing rule and establish that it is not a well quasi-order. Moreover, we show that there cannot exist a transition-preserving compatible well quasi-order, preventing us to use the framework of Well Structured Transition Systems to prove that coverability is decidable. We show that the RPN languages are *quite close* to recursively enumerable languages, since the closure under homomorphism and intersection with a regular language is the family of recursively enumerable languages. More precisely, we show that RPN coverability (as reachability) languages

strictly include the union of context-free languages and Petri net coverability languages. Moreover, we prove that RPN coverability languages and reachability languages of Petri nets are incomparable. We prove that RPN coverability languages are a strict subclass of RPN reachability languages. In addition, we establish that the family of RPN languages is closed under union, homomorphism, concatenation and Kleene star, but neither under intersection with a regular language nor under complementation. This answers the first half of Question 2, by showing that the newly defined quasi-order has interesting qualities.

From a complexity point of view, we show that, as for Petri nets, coverability, termination, boundedness, and finiteness are EXPSPACE-complete, answering the second half of Question 2. Thus, the increase of expressive power does not entail a corresponding increase in complexity. In order to solve the coverability problem, we design a set of reductions, reducing the coverability problem for RPN to the coverability problem in Petri net. In order to solve the termination problem for RPN, we consider two cases for an infinite sequence, depending on (informally speaking) whether the depth of the trees corresponding to states are bounded or not along the sequence. For the unbounded case, we introduce the abstract graph that expresses the ability to create threads from some initial state. The decidability of the finiteness and boundedness problems are also mainly based on this abstract graph.

In Chapter 9, we introduce Dynamic Recursive Petri nets (DRPN) (motivated by the limitation of RPN), a model extending RPN with generalized transition features and dynamic thread creation. We show that the family of coverability languages of DRPN are strictly more expressive than the family of coverability languages of RPN. We achieve this in two ways, first using the generalized transition features, and second using the dynamic thread creation, showing that both extensions are "true extensions" for RPN. Finally, we prove that the coverability problem is still decidable for DRPN, using a type of saturation algorithm, allowing us to reduce the problem to the one for WSTS, which answers Question 3.

Active Learning and Verification

In Chapter 10, we describe 3 types of algorithms which verify that a given RNN satisfies some given specifications:

- **Statistical Model Checking (SMC).** This algorithm generates a large (statistically significant) finite set of words and checks whether the RNN satisfies the given specification on this set.
- **Automaton Abstraction and Model Checking (AAMC).** This is the algorithm we described above. It generates a DFA which approximates the RNN, and then checks whether it satisfies the given specification.

- **Property-Directed Verification (PDV).** Similarly to AAMC, it generates a DFA approximating the RNN. But instead of performing the generation of the RNN and then the model checking, we intertwined them together. This algorithm was inspired by black-box checking from [126].

The PDV algorithm is our answer to Question 4, where its superiority can be seen by its performance on the experiments we ran in the end of Chapter 10. We compare these algorithms according to a set of benchmarks coming from a variety of sources. We find that the quickest algorithm is the PDV. On average it is 4.5 times faster than SMC and 20 times faster than AAMC. Moreover, we show that algorithms generating a DFA have two other advantages on SMC. First, both AAMC and PDV find counter examples which are smaller than SMC. Second, in case we find a mistake we can use the generated DFA in order to find a "faulty flow", which allows us to generate more counter examples and to "visualize" the "reason" the problem exists.

In Chapter 11, we investigate the robustness of Angluin's algorithm to noisy models (answering Question 5). To this end, we introduce three types of noise applied to DFA, producing three types of random languages:

1. **Noisy output.** A random language produced by reversing the word acceptance of the DFA with a small probability,
2. **Noisy input.** A random language produced by, with a small probability, replacing each letter of a word by one chosen uniformly from the alphabet and then checking whether the DFA accepts it,
3. **Counter DFA.** A language which combines the status of a word w.r.t. the DFA and its status w.r.t. a randomly generated counter automaton.

Our experiments consisted of first generating several hundreds of random DFA, to which we applied these noises, and then computing the statistical distance from the original DFA. We were investigating several questions, where the important ones were: (1) what is the threshold in terms of distance between the original and noisy language above which the algorithm produces a device that is no more "similar to" the original DFA? (2) What is the impact of the nature of noise on the robustness of Angluin's algorithm? Our experiments show that for the first two types of noises (noisy output and noisy input) there is a threshold from which Angluin's algorithm is able to extract a DFA "similar to" the original one. However, for the third type of noise (counter DFA), there does not exist a threshold from which the original DFA is "recoverable". These results directed us to conjecture that Angluin's algorithm is more robust to noise which is less structured. To this end we prove that the random languages that are produced by the first noise and second noise (under a slight condition on the original DFA), are almost surely not recursively enumerable.

In Chapter 12, we present the family of *visibly pushdown languages* which is a subset of the family of context-free languages. These languages were originally defined as the languages of visibly pushdown automaton introduced by [7, 8], but can equivalently be seen as the languages of visibly pushdown grammars. Moreover, this family includes the family of balanced parentheses languages. For this family we present: (1) an active learning algorithm for visibly pushdown grammars (answering Question 6), and (2) its applicability for learning surrogate models of RNN trained on context-free languages. Our learning algorithm makes use of the proximity of visibly pushdown languages and regular tree languages, and builds on an existing learning algorithm for regular tree languages. Equivalence tests between a given RNN and a hypothesis grammar rely on a mixture of A* search and random sampling. An evaluation of our approach on a set of RNNs from the literature shows good preliminary results.

1.2 Organization of the thesis

- **Preliminaries** (Chapter 2). We give the basics of: WSTS, Petri nets, formal languages, and Angluin's algorithm, used in this thesis.

Part I - Cover of Petri Nets

- **Covering and Abstractions** (Chapter 3). We define a new construction called abstractions, which commodifies the accelerations. Using this, we show a new proof for the Karp and Miller algorithm and improve it.
- **Computing the Clover Efficiently** (Chapter 4). We answer Question 1, by fixing the `MCT` algorithm. We achieve that by using the commodified accelerations. This gives a new algorithm `MinCov`, which uses only EXPSPACE supplementary memory.
- **The Tool MinCov** (Chapter 5). We discuss the implantation details of `MinCov`. We show small optimization done to it, such as a smart choice of exploration order, fast data structures, etc. Moreover, we show how one can use `MinCov` to solve the coverability problem quickly. Finally, we compare its performance to other tools computing the Clover and coverability.

Part II - Recursive Petri net

- **Extending Petri nets** (Chapter 6). We discuss some limitations of Petri nets, such as their inability to do zero tests, or simulate a stack. In the second section we describe some models extending Petri nets solving the problems described.

- **Recursive Petri Nets** (Chapter 7). We begin by introducing RPNs, their syntax and their semantics. We continue by reviewing their usage and previous results.
- **Expressiveness and Decision problems in RPN** (Chapter 8). The chapter where we answer Question 2. We define an order on the states of an RPN. We show that the coverability language defined by this order includes context free and Petri net coverability languages. Finally, using a Rackoff type technique we show that coverability, termination, and boundedness problems are EXPSPACE-complete.
- **Dynamic Recursive Petri Nets** (Chapter 9). The chapter where we answer Question 3, defining a new model, the Dynamic Recursive Petri Nets. We show that it extends RPN and Affine nets. Moreover, we show that its coverability languages strictly include that of RPN coverability languages. We finish this chapter by showing that coverability is decidable.

Part III - Active Learning and Verification

- **Property Directed Verification** (Chapter 10). We begin by presenting the three types of verification algorithms for RNN. We then compare them on a set of benchmarks.
- **Noisy DFA** (Chapter 11). The chapter where we study how different kinds of noise effect Angluin's algorithm, answering Question 5. We review several hundreds of experiments performed to this end. In the last section, we show some theoretical results to strengthen our conclusions on the experimental data.
- **VPA** (Chapter 12). The chapter where we introduce the family of visibly pushdown languages. For this family of languages, we present an active learning algorithm (answering Question 6), and test its efficiency on extracting visibly pushdown languages from a set of RNN.

Chapter 2

Preliminaries

2.1 Well Structured Transition Systems

Well-structured transition systems (WSTS) [63] are a family of infinite state systems for which many verification problems are decidable. These decidability results depend on a well-quasi-ordering between states, which is also compatible with transitions. This family extends many well-known models such as Petri nets, post self modifying nets, timed automata, lossy systems...
This section is strongly based on the paper [63] and all the missing proofs, details and in depth explanations can be found there.

2.1.1 Well-Quasi-Order

Given a set X, we call a relation $\leq$ on it an *order* if it is:

Reflexive $\forall x \in X \; x \leq x$;
Transitive $\forall x, y, z \in X$ if $x \leq y$ and $y \leq z$ then $x \leq z$;
Antisymmetric $\forall x, y \in X$ if $x \leq y$ and $y \leq x$ then $x = y$;

If we drop the antisymmetry requirement we get a *quasi-order* (qo). Let $x < y$ denote $x \leq y \not\leq x$. We call an order $(X, \leq)$ *well-founded* if there is no infinite strictly decreasing sequence, i.e., $(x_i)_{i=0}^{\infty} \subseteq X$ such that:

$$x_1 > x_2 > x_3 > \dots$$

An order is a *well-quasi-order* (wqo) if for any sequence $(x_i)_{i=0}^{\infty} \subseteq X$ there exist $i < j$ such that $x_i \leq x_j$. If the order is well-founded and ordered (i.e., not qo) we call it *well-ordered*. For example of a well-order, we have the usual order $\leq$ on $\mathbb{N}^d$, defined as follows, given $v, v' \in \mathbb{N}^d$ we say that $v \leq v'$ if and only if $v(i) \leq v'(i)$ for all $i \leq d$.

Given a qo $(X, \leq)$, we call a set $U \subseteq X$ *upward (resp. downward) closed* if for any $x \leq y$ (resp. $x \geq y$), where $x \in U$ and $y \in X$ then $y \in U$. Give an $x \in X$, denote the upward closer of x by $\uparrow x = \{y \in X \mid y \geq x\}$, and for a set $B \subseteq X$ denote $\uparrow B = \bigcup_{x \in B} \uparrow x$, respectively for downward closure we denote $\downarrow x$ and $\downarrow B$. Given an upward closed set U, we call a subset $B \subseteq U$ a basis of U if $\uparrow B = U$. We call a basis $\uparrow B = U$ *minimal basis* if for any other basis $\uparrow B' = U$ we have $|B| \leq |B'|$. A well-known result [75] gives us that if the qo is a wqo if and only if any upward closed set has a finite basis. For example, the minimal basis of the upward closed set $\{v \in \mathbb{N}^3 \mid v(0) \geq 2 \text{ or } v(1) \geq 1\}$ is $\{(2,0,0),(0,1,0)\}$. As a consequence of this result, one gets that:

Lemma 2.1.1. *Given a wqo $(X, \leq)$, for any increasing of sequence upward closed sets $U_1 \subseteq U_2 \subseteq U_3 \subseteq \cdots$, there exists an $n \in \mathbb{N}$ such that $U_n = U_{n+1} = U_{n+2} = \cdots$.*

An *antichain* E is a set for which : $\forall x \neq y \in E \ \neg(x \leq y \vee y \leq x)$. For an example, the set:

$$\{(n,0),(n-1,1),\ldots,(0,n)\} \subseteq \mathbb{N}^2$$

is an antichain on length $n+1$ in $(\mathbb{N}^2, \leq)$. X is *FAC* if all of its antichains are finite. A set $E \subseteq X$ is *directed* if E is nonempty and for all $x, y \in E$ there exists $z \in E$ such that $x \leq z$ and $y \leq z$. An *ideal* is a directed downward closed set. A well-known characterization is that a set is FAC if and only if it is equal to a finite union of ideals. A proof of this result can be found in [27]. Given a set $E \subseteq X$, there may exist several finite families of ideals whose union is equal to E. Among all these finite families, one can choose the unique set of maximal ideals (by inclusion): this set is therefore canonically associated with E. An alternative definition for $(X, \leq)$ to be wqo is if it is well-founded and FAC.

2.1.2 Well structured transition systems

A *transition system* is a tuple $TS = \langle S, \rightarrow \rangle$, where S is a set of elements we call *states* and $\rightarrow \subseteq S \times S$ is a set of elements we call *transitions*. Denote by $\xrightarrow{*}$ the reflexive and transitive closure of the relation $\rightarrow$. For a state $s \in S$, denote by $Succ_{TS}(s) = \{s' \in S \mid s \rightarrow s'\}$ the set of immediate successors and $Pred_{TS}(s) = \{s' \in S \mid s' \rightarrow s\}$ the set of immediate predecessors. Given an qo $\leq$, we that it is *compatible* with TS, if for any $s_1 \leq t_1$ and $s_1 \rightarrow s_2$ there exists a state $t_2 \in S$ such that $t_1 \xrightarrow{*} t_2$ and $s_2 \leq t_2$ (see Figure 2.1). Moreover, denote by $Succ^*_{TS}(s) = \{s' \in S \mid s \xrightarrow{*} s'\}$ and $Pred^*_{TS}(s) = \{s' \in S \mid s' \xrightarrow{*} s\}$.
A *Well Structured Transition System* (WSTS), $\langle S, \rightarrow, \leq \rangle$, is a transition system equipped with $\leq$ a qo on S where $\leq$ is a wqo, and $\leq$ is compatible.

We call the WSTS *effective* if the WSTS has a set of state the relation and ordering are finitely encoded, there exists a Turing machine to decide the transition

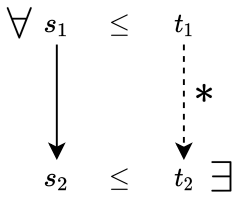


Figure 2.1: Compatible relation.

relation, the order relation and whether an element is in the states. For more details, see [27]. From now on, we consider that all WSTS are effective.

For any WSTS W and a state $s \in S$, we define:

1. The *reachability set* $Reach(W, s)$ is defined by:

$$Reach(W, s) = \{s' \in S \mid s \xrightarrow{*} s'\} (= Succ^*_W(s))$$

2. The *coverability set* $Cover(W, s)$ is defined by:

$$Cover(W, s) = \downarrow Reach(W, s)$$

A fundamental question one can ask about WSTS (and transition system in general) is the reachability problem:

Definition 2.1.2 (Reachability problem)**.** Given a WSTS $W = \langle S, \rightarrow, \leq \rangle$, an initial state $s_0 \in S$, and a target state $s \in S$. Does s belong to $Reach(W, s_0)$?

Unfortunately, reachability in general is undecidable for WSTS, this result is due to WSTS being an extension of rest-net (defined and discussed in Section 6.2), and rest net having undecidable reachability. Another problem is the one which uses the $Cover$ instead of the $Reach$:

Definition 2.1.3 (Coverability problem)**.** Given a WSTS $W = \langle S, \rightarrow, \leq \rangle$, an initial state $s_0 \in S$, and a target state $s \in S$. Does s belong to $Cover(W, s_0)$?

Note that, $s \in Cover(W, s_o)$ if and only if $s_0 \in Pred^*_W(\uparrow s)$. Therefore, if we had an algorithm computing $Pred^*_W(\uparrow s)$ we would be able to solve coverability. We say that a WSTS has *effective pred-basis* if there exists an algorithm such that for any $s \in S$ it computes $pb(s)$ a finite basis of $\uparrow Pred_W(\uparrow s)$. We extend the notation of pb to finite sets, given $C \subseteq S$, then $pb(C)$ is a finite basis of $\uparrow Pred_W(\uparrow C)$. Note that using the algorithm for $pb(c)$, we can get an algorithm deciding $pb(C)$. From now on, we consider that all the WSTS have an effective pred-basis. An algorithm computing $Pred^*_W(\uparrow s)$ exists for WSTS, the *backward coverability* algorithm [1],

Algorithm 1: Backward coverability algorithm

Input: A WSTS W, and a state $s \in S$
Output: A finite basis of $Pred^*_W(\uparrow s)$

```
1 C ← {s};
2 repeat oldC ← C; C ← C ∪ pb(C) until ↑C =↑ oldC;
3 return C
```

see Algorithm 1. First, we note that the Algorithm 1 terminates since in every iteration the set $\uparrow oldC \subseteq \uparrow C$, and from Lemma 2.1.1 we know that this sequence stabilizes. When it terminates $\uparrow C = \bigcup_{n=0}^{m} \uparrow Pred^n_W(\uparrow s)$ for some $m \in \mathbb{N}$. On one hand, note that the set $Pred^*_W(\uparrow s)$ is upward closed, hence

$$C = \bigcup_{n=0}^{m} \uparrow Pred^n_W(\uparrow s) = \bigcup_{n=0}^{m} Pred^n_W(\uparrow s) \subseteq Pred^*_W(\uparrow s)$$

i.e., Algorithm 1 is consistent. On the other, because of the stabilization we have that for some finite $m \in \mathbb{N}$:

$$Pred^*_W(\uparrow s) = \bigcup_{n=0}^{\infty} Pred^n_W(\uparrow s) = \bigcup_{n=0}^{m} Pred^n_W(\uparrow s) \subseteq C$$

i.e., Algorithm 1 is complete.

Proposition 2.1.4 ([63]). *Given a WSTS and a state s, the Algorithm 1 terminates and returns a finite basis of $Pred^*_W(\uparrow s)$.*

Finally, since $s \in Cover(W, s_0)$ if and only if $s_0 \in Pred^*_W(\uparrow s)$ we have that:

Theorem 2.1.5. *Coverability is decidable for WSTS.*

2.2 Petri nets

We define the Petri nets by using the *backward incidence* matrix **Pre** and the *incidence* matrix **C**, as compared to the usual way of using **Pre** and the *forward incidence matrix* **Post**. The connection between these two definitions is that $\mathbf{Post} = \mathbf{C} + \mathbf{Pre}$.

Definition 2.2.1. A *Petri net* is a tuple $\mathcal{N} = \langle P, T, \mathbf{Pre}, \mathbf{C} \rangle$ where:

- P is a finite set of *places*;
- T is a finite set of *transitions* with $P \cap T = \emptyset$;

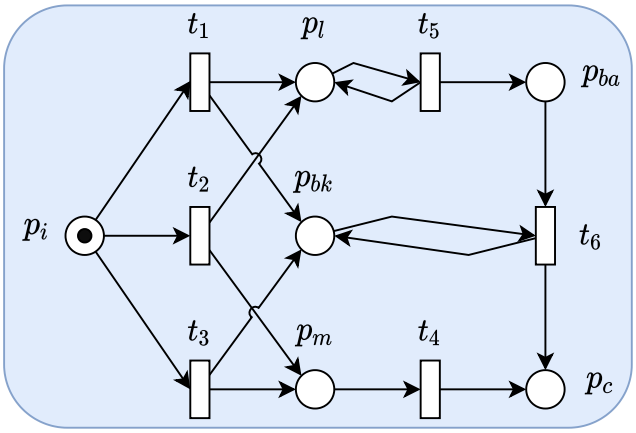


Figure 2.2: The marked Petri net $\mathcal{N}_1$.

- $\mathbf{Pre} \in \mathbb{N}^{P \times T}$ is the *backward incidence matrix*;
- $\mathbf{C} \in \mathbb{Z}^{P \times T}$ is the *incidence matrix* for which the following holds:
  For all $p \in P$ and $t \in T$, $\mathbf{C}(p,t) + \mathbf{Pre}(p,t) \geq 0$.

A *marked Petri net* $(\mathcal{N}, \mathbf{m}_0)$ is a Petri net $\mathcal{N}$ initialized with a *marking* $\mathbf{m}_0 \in \mathbb{N}^P$.

One can see in Figure 2.2, a graphical representation of a Petri net $\mathcal{N}_1$, where places are denoted by circles, transitions by squares and the arrows represent the backward incidence and incidence matrix. So in Figure 2.2 the Petri net $\mathcal{N}_1$ has a set of 6 places $P = \{p_i, p_\ell, p_{bk}, p_m, p_{ba}, p_c\}$, a set of 5 transitions $T = \{t_1, t_2, t_3, t_4, t_5\}$ and its backward and incidence matrices are:

$$\mathbf{Pre} = \begin{array}{c} \\ p_i \\ p_\ell \\ p_{bk} \\ p_m \\ p_{ba} \\ p_c \end{array} \begin{array}{c} \begin{array}{cccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array} ; \mathbf{C} = \begin{array}{c} \\ p_i \\ p_\ell \\ p_{bk} \\ p_m \\ p_{ba} \\ p_c \end{array} \begin{array}{c} \begin{array}{cccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \end{array}.$$

Moreover, we can also represent graphically the marking of a Petri net, by a number of small tokens inside the places. For example, the marking of the Petri net $\mathcal{N}_1$ in Figure 2.2 is p_i.

Given an alphabet Σ we define, as usual Σ^*, the *Kleene star* of Σ. The column vector of the matrix $\mathbf{Pre}$ (resp. $\mathbf{C}$) indexed by $t \in T$ is denoted by $\mathbf{Pre}(t)$ (resp. $\mathbf{C}(t)$). A transition $t \in T$ is *fireable* from a marking $\mathbf{m} \in \mathbb{N}^P$ if $\mathbf{m} \geq \mathbf{Pre}(t)$. When t is fired from a marking $\mathbf{m}$, its *firing* leads to a marking $\mathbf{m}' \stackrel{\text{def}}{=} \mathbf{m} + \mathbf{C}(t)$, which

is denoted by $\mathbf{m} \xrightarrow{t} \mathbf{m}'$. We extend the firing rule to a sequence of firings $\sigma \in T^*$ recursively according to its length, as follows: The empty sequence ε is always fireable and does not change the marking. The sequence $\sigma = t\sigma'$, with $t \in T$ and $\sigma' \in T^*$ is fireable from $\mathbf{m}$ if $\mathbf{m} \xrightarrow{t} \mathbf{m}'$ and σ' is fireable from $\mathbf{m}'$. The firing of σ from $\mathbf{m}$ leads to a marking $\mathbf{m}''$ reached by σ' from $\mathbf{m}'$. We denote this firing by $\mathbf{m} \xrightarrow{\sigma} \mathbf{m}''$.

Petri nets with P places are WSTS, where the states are $\mathbb{N}^P$, the transitions are as described above, and the order defined on the states is the (also defined in the previous section) wqo on the vectors in $\mathbb{N}^P$. Both the reachability and coverability problems are decidable for Petri nets. Moreover, the reachability problem was recently shown to be Ackermannian-complete [104, 41] (i.e., $\mathbb{F}_\omega$) and the coverability problem is EXPSPACE-complete [132].

Since the coverability set is downward closed and $(\mathbb{N}^P, \leq)$ is FAC, it can be expressed as a finite union of ideals. The ideals of $\mathbb{N}^P$ can be elegantly defined as follows. We first extend the natural numbers and integers: $\mathbb{N}_\omega = \mathbb{N} \cup \{\omega\}$ and $\mathbb{Z}_\omega = \mathbb{Z} \cup \{\omega\}$. Then we extend the order relation and the addition operation to $\mathbb{Z}_\omega$: For all $n \in \mathbb{Z}$, $\omega > n$ and for all $n \in \mathbb{Z}_\omega$, $n + \omega = \omega + n = \omega$. $\mathbb{N}_\omega^P$, with this extended order, is still well-ordered, and its elements are called *ω-markings*. There is a one to one correspondence between ideals of $\mathbb{N}^P$ and ω-markings. Let $\mathbf{m} \in \mathbb{N}_\omega^P$. Denote by $[\![\mathbf{m}]\!]$ the set:

$$[\![\mathbf{m}]\!] = \{\mathbf{m}' \in \mathbb{N}^P \mid \mathbf{m}' \leq \mathbf{m}\}$$

$[\![\mathbf{m}]\!]$ is an ideal of $\mathbb{N}^P$ (and any ideal can be represented as such). By the definitions and properties stated above, we are able to formally define the *Clover* of a Petri net.

Definition 2.2.2. Let $(\mathcal{N}, \mathbf{m}_0)$ be a marked PN. Then $Clover(\mathcal{N}, \mathbf{m}_0) \subseteq \mathbb{N}_\omega^P$ is the (finite) set of maximal (for inclusion) ideals such that :

$$Cover(\mathcal{N}, \mathbf{m}_0) = \bigcup_{\mathbf{m} \in Clover(\mathcal{N}, \mathbf{m}_0)} [\![\mathbf{m}]\!]$$

Example 2.2.3. The marked Petri net $\mathcal{N}_1$ in Figure 2.2 is unbounded. Its $Clover(\mathcal{N}_1, p_i)$ is the set of four elements:

$$\{p_i, p_{bk} + p_m, p_l + p_m + \omega p_{ba}, p_l + p_{bk} + \omega p_{ba} + \omega p_c\}$$

For example, the marking $p_l + p_{bk} + \alpha p_{ba} + \beta p_c$ is reachable by the sequence $t_1 t_5^{\alpha+\beta} t_6^{\beta}$ and therefore covered.

2.3 Language theory

Given an alphabet Σ, a language (of finite words) on this alphabet is a subset $\mathcal{L} \subseteq \Sigma^*$. We denote by ε the empty word. Note that, in Part III we change the notation of the empty word to λ due to conflicts with the notation for PAC learning. For a language $\mathcal{L}$ we denote its complement by $\overline{\mathcal{L}} = \{w \in \Sigma^* \mid w \notin L\}$. For two languages $\mathcal{L}_1, \mathcal{L}_2 \subseteq \Sigma^*$, we let $\mathcal{L}_1 \setminus \mathcal{L}_2 = \mathcal{L}_1 \cap \overline{\mathcal{L}_2}$. The symmetric difference of $\mathcal{L}_1$ and $\mathcal{L}_2$ is defined as $\mathcal{L}_1 \oplus \mathcal{L}_2 = (\mathcal{L}_1 \setminus \mathcal{L}_2) \cup (\mathcal{L}_2 \setminus \mathcal{L}_1)$.

A fundamental approach in the research of abstract model is to consider them as language generators.

2.3.1 Regular languages

Let us start with the family of *regular languages* on the alphabet Σ. This family is defined inductively with the following rules:

1. $\emptyset$ is a regular language;
2. For all $a \in \Sigma$, the language $\{a\}$ is a regular language;
3. Given a regular language $\mathcal{L}$, the Kleene star of this language, i.e., $\mathcal{L}^*$, is a regular language;
4. Given two regular languages $\mathcal{L}, \mathcal{L}'$, the union, i.e., $\mathcal{L} \cup \mathcal{L}'$, and the concatenation, i.e., $\mathcal{L} \cdot \mathcal{L}'$, are regular languages.

For example, the language $\{(ab)^n \mid n \in \mathbb{N}\}$ is regular, since it is equal to $(\{a\} \cdot \{b\})^*$, which is the application of the above rules (in order) 2, 4, and then 3. Another example is $\{\varepsilon\}$ where ε is the empty word, since it can be expressed as $\emptyset^*$.

We now define a *deterministic finite automaton* (DFA) which are closely related to regular languages:

Definition 2.3.1 (DFA)**.** A (complete) deterministic finite automaton (DFA) is a 5-tuple $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F \rangle$ such that:

1. Q is a finite set of states;
2. Σ is a finite alphabet;
3. $\delta : Q \times \Sigma \to Q$ is a transition function;
4. $q_0 \in Q$ is an initial state;
5. $F \subseteq Q$ is a finite set of final states.

The transition function δ is inductively extended to words as follows: $\delta(q, \varepsilon) = q$ and $\delta(q, wa) = \delta(\delta(q, w), a)$. We say that a word w is *accepted* by $\mathcal{A}$ if $\delta(q_0, w) \in F$. The language generated by a DFA $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, is the set of all accepted words.

Theorem 2.3.2 (Kleene theorem)**.** *The family of regular languages is equal to the family of DFA languages.*

For example one can see in Figure 2.3 the DFA $\mathcal{A}_1$ with the generated language $\mathcal{L}(\mathcal{A}_1) = \{(ab)^n \mid n \in \mathbb{N}\}$. The states are represented by circles, where a state with a double circle is a final state, and a state with an incoming arrow not connected to anything else in the initial state. The transition function is represented by the arrows between the states. Given a DFA $\mathcal{A} = \langle Q, \delta, q_0, F \rangle$, the complement

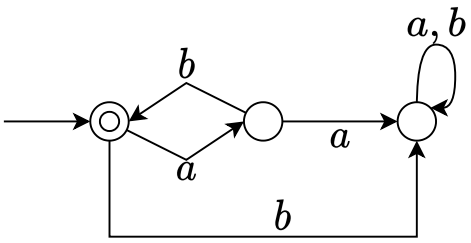


Figure 2.3: The DFA $\mathcal{A}_1$ with the language $\{(ab)^n \mid n \in \mathbb{N}\}$.

DFA $\overline{\mathcal{A}} = (Q, \delta, q_0, Q \setminus F)$, we get $\mathcal{L}(\overline{\mathcal{A}}) = \overline{\mathcal{L}(\mathcal{A})} = \Sigma^* \setminus \mathcal{L}(\mathcal{A})$. Given a DFA $\mathcal{A}$, we call it a *minimal DFA* if for any other DFA $\mathcal{A}'$ with the same language, i.e. $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$, $\mathcal{A}$ has less or equal number of states than $\mathcal{A}'$.

Theorem 2.3.3. *For any regular language $\mathcal{L}$ there exists a unique minimal DFA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}$.*

Given a language $\mathcal{L} \subseteq \Sigma^*$ and $x, y \in \Sigma^*$ we say that $z \in \Sigma^*$ is a *distinguishing word* if exactly one of the two words xz, yz is in $\mathcal{L}$. We say that two words $x, y \in \Sigma^*$ are *Nerode congruent*, i.e., $x \sim_{\mathcal{L}} y$, if and only if they do not have a distinguishing word.

Theorem 2.3.4 (Myhill–Nerode theorem)**.** *The language $\mathcal{L}$ is regular if and only if Nerode congruence has a finite number of equivalence classes. Moreover, the number of classes is equal to the number of states of the minimal DFA $\mathcal{A}$ such that $\mathcal{L} = \mathcal{L}(\mathcal{A})$.*

2.3.2 Context-free languages

Another well-known family of languages is the family of languages generated by *Context-Free Grammars* (CFG).

Definition 2.3.5 (CFG)**.** A context-free grammar, is a 4-tuple $\langle V, \Sigma, R, S \rangle$ such that:

1. V is a finite alphabet, the *non-terminal* variables;
2. Σ is a finite alphabet, the *terminals* variables;
3. R is a subset of $V \times (V \cup \Sigma)^*$, the *rewrite relation*;
4. $S \in V$ is a starting symbol.

The language $\mathcal{L}(G) \subseteq \Sigma^*$ of a grammar G is defined using a global rewrite relation $\Rightarrow \subseteq (\Sigma \cup V)^* \times (\Sigma \cup V)^*$ defined by $uAv \Rightarrow uwv$ for all rewrite action (A, w) and $u, v \in (\Sigma \cup V)^*$. With this, we let $\mathcal{L}(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$ where $\Rightarrow^*$ denotes the reflexive transitive closure of the binary relation $\Rightarrow$. We call a language $\mathcal{L}$ *context-free* if there exists a CFG G over Σ such that $\mathcal{L} = \mathcal{L}(G)$. The family of context-free languages is strictly greater than the family of regular languages.

Theorem 2.3.6. *Any regular language $\mathcal{L}$ is a context-free language.*

The language of palindromes on $\Sigma = \{a, b\}$ is a non-regular context-free language. The CFG generating it is $\Sigma = \{a, b\}$, $V = \{S\}$, S is the starting symbol, and it has three rewrite actions:

$$S \to aSa; \quad S \to bSb; \quad S \to \varepsilon.$$

The family of Dyck languages (i.e., the family of balanced parentheses) is a family of non-regular context-free languages. The CFG of a Dyck language of order n (the number of different parentheses) on the alphabet $\Sigma = \{p_i, q_i\}_{i=1}^n$ is $V = \{S\}$, S is the starting symbol, and it has $n + 1$ rewrite actions:

$$\text{For all } i \leq n \; S \to p_i S q_i \, S; \quad S \to \varepsilon.$$

Given the alphabets Σ_1 and Σ_2 and a function $h : \Sigma_1^* \mapsto \Sigma_2^*$, we call this function h a *homomorphism* if
$h(ww') = h(w)h(w')$ for all $w, w' \in \Sigma_1^*$. An important property of Dyck languages was shown by Chomsky and Schützenberger. They showed that any context-free language can be 'represented' by a Dyck language and a regular language:

Theorem 2.3.7 (Chomsky–Schützenberger representation)**.** *Let $\mathcal{L}$ be a context-free language on the alphabet Σ, then there exist a Dyck language $\mathcal{L}_D$, a regular language $\mathcal{L}_R$ both on the same alphabet Σ', and a homomorphism $h : \Sigma' \mapsto \Sigma$, such that $\mathcal{L} = h(\mathcal{L}_D \cap \mathcal{L}_R)$.*

Just like in the case of regular languages, there is a device generating exactly any context-free language.

Definition 2.3.8 (PDA)**.** A pushdown automaton (PDA) is a 7-tuple $\mathcal{A} = \langle Q, \Sigma, \Gamma, \delta, q_0, Z, F \rangle$ such that:

1. Q is a finite set of states;
2. Σ is a finite alphabet, the *input alphabet*;
3. Γ is a finite alphabet, the *stack alphabet*;
4. δ is a finite subset of $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \times Q \times \Gamma^*$;
5. $q_0 \in Q$ is an initial state;
6. $Z \in \Gamma$ is an initial stack symbol;
7. $F \subseteq Q$ is a finite set of final states.

The set of configurations of $\mathcal{A}$ is $Cf(\mathcal{A}) = Q \times \Sigma^* \times \Gamma^*$. The step relation $\vdash$ on $Cf(\mathcal{A}) \times Cf(\mathcal{A})$ is defined as follows. For all $(p, a, A, q, v) \in \delta$ and all $(q, aw, Au) \in Cf(\mathcal{A})$, $(q, aw, Au) \vdash (q', w, vu)$. Then the language of $\mathcal{A}$ is defined by: $\mathcal{L}(\mathcal{A}) = \{w \mid \exists (q, u) \in F \times \Gamma^* \ (q_0, w, Z) \vdash^* (q, \varepsilon, u)\}$ and we get:

Proposition 2.3.9. *The family of context-free languages is equal to the family of languages accepted by pushdown automata.*

For example, in Figure 2.4 one can see the PDA $\mathcal{A}_2$ for the language of palindromes on two letters.

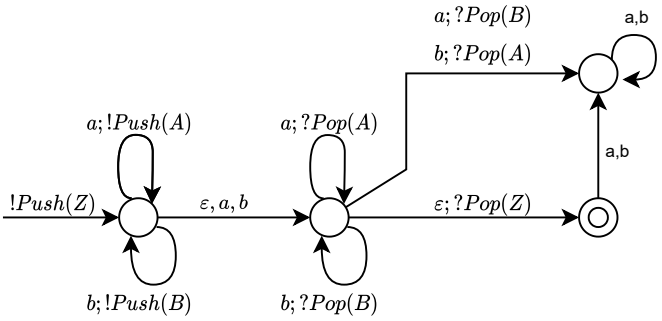


Figure 2.4: A PDA whose language is the language of palindromes on two letters.

2.3.3 Recursively enumerable languages

Definition 2.3.10. A deterministic Turing machine is a 7-tuple $TM = \langle Q, q_0, q_f, \Gamma, \flat, \Sigma, \delta \rangle$ such that:

1. Q is a finite set of states;
2. $q_0 \in Q$ is an initial state;
3. $F \subseteq Q$ are final states;
4. Γ, *tape alphabet*;
5. $\flat \in \Gamma$ is the *blank symbol*;

6. $\Sigma \subseteq \Gamma$ with $\flat \notin \Sigma$, is the *input alphabet*;
7. δ is a partial function from $Q \times \Gamma \mapsto Q \times \Gamma \setminus \{\flat\} \times \{-1, 1\}$.

The set of configurations of TM is $Cf(TM) = \{(q, w, n) \mid q \in Q, w \in (\Gamma \setminus \{\flat\})^*, 0 < n \leq |w| + 1\}$. The step relation $\vdash$ on $Cf(TM) \times Cf(TM)$ is defined as follows.

- For all $(q, w, |w|+1) \in Cf(TM)$ and $(q', a, d) = \delta(q, \flat)$ such that $|w|+1+d > 0$,
  $(q, w, |w| + 1) \vdash (q', wa, |w| + 1 + d)$;
- For all $(q, w, n) \in Cf(TM)$ with $n \leq |w|$ and $(q', a, d) = \delta(q, w[n])$ such that $n + d > 0$,
  $(q, w, n) \vdash (q', w[1, n - 1]aw[n + 1, |w|], n + d)$.

Then the language of TM is defined by: $\mathcal{L}(\mathcal{A}) = \{w \mid \exists (q, u, n) \in Cf(TM), q \in F, (q_0, w, 1) \vdash^* (q, u, n)\}$.

We say that $\mathcal{L}$ is a *recursively enumerable* language if there exists a Turing machine TM such that $\mathcal{L} = \mathcal{L}(TM)$. $\mathcal{L}$ is *recursive* if $\mathcal{L}$ is a recursively enumerable and $\overline{\mathcal{L}}$ is a recursively enumerable.

This family of recursive languages strictly includes the family of context-free languages.

Theorem 2.3.11. *Any context-free language $\mathcal{L}$ is a recursive language.*

$\mathcal{L} = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ is a non context-free recursive language.

2.3.4 Petri net languages

All the results and definitions of this subsection come from the Peterson book [127].

Using a similar procedure generating regular languages from finite state machines, one can define a Petri net languages. Let $(\mathcal{N}, \mathbf{m})$ be a marked Petri net, S_f a finite set of final markings, and we equip any transition t with a label $\lambda : T \mapsto \Sigma \cup \{\varepsilon\}$ where Σ is a finite alphabet and ε is the empty word. The labeling is extended to transition sequences in the usual way. Once S_s and λ are specified we can define two languages of the marked Petri net:

Reachability languages. All the words labeling sequences reaching some state of S_f:

$$\mathcal{L}_R(\mathcal{N}, \mathbf{m}, \lambda, S_f) = \{\lambda(\sigma) | \mathbf{m} \xrightarrow{\sigma} \mathbf{m}' \in S_f\}$$

Coverability languages. All the words labeling sequences covering some state of S_f:

$$\mathcal{L}_C(\mathcal{N}, \mathbf{m}, \lambda, S_f) = \{\lambda(\sigma) | \mathbf{m} \xrightarrow{\sigma} \mathbf{m}'' \geq \mathbf{m}' \in S_f\}$$

Let us for now focus on the reachability languages, the larger family:

Proposition 2.3.12 ([128, 68]). *The family of Petri net reachability languages strictly includes the family of coverability languages.*

The language $\mathcal{L}_2 = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ is a Petri net reachability language. Indeed, $L_2 = Lan_R(\mathcal{N}_2, p_i, \lambda, \{p_f\}\})$ where $\mathcal{N}_2$ is the Petri net depicted in Figure 2.5, and the labeling is denoted by (red) letters above the transitions. Reachability lan-

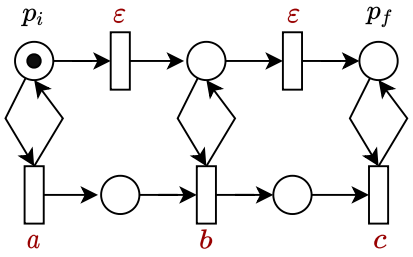


Figure 2.5: The Petri net $\mathcal{N}_2$ for the language $\mathcal{L}_2 = \{a^n b^n c^n \mid n \in \mathbb{N}\}$.

guages have some important closure properties:

Proposition 2.3.13. *([127]) The family of Petri net reachability languages is closed under the following operations: union, intersection, Kleene star, concatenation, and homomorphism.*

Let us describe the relations between the family of reachability languages and the previously defined ones.

Proposition 2.3.14. *([127]) The family of regular languages is strictly included in the family of reachability languages of Petri nets.*

The context-free languages and the reachability languages are incomparable, where for example it was shown that:

Lemma 2.3.15. *([127]) The language of palindromes on two letters is not a reachability language for any Petri net.*

Moreover, the language $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ is not a context-free language, hence:

Proposition 2.3.16. *The family of context-free languages and the family of Petri net reachability languages are incomparable.*

Since reachability for Petri nets is decidable, and context-free languages are recursive, we have:

Proposition 2.3.17. *The family of reachability languages of Petri nets is strictly included in the family of recursive languages.*

2.3.5 L* — algorithm

Angluin introduced L*, a classical instance of a learning algorithm in the presence of a minimally adequate teacher (MAT) [10]. Given any regular language $\mathcal{L} \subseteq \Sigma^*$, the algorithm L* eventually outputs the unique minimal DFA $\mathcal{H}$ such that $\mathcal{L}(\mathcal{H}) = \mathcal{L}$. The crux is that, while Σ is given, $\mathcal{L}$ is a priori unknown and can only be accessed through *membership queries (MQ)* and *equivalence queries (EQ)*:

(MQ) $w \in \mathcal{L}$? for a given word $w \in \Sigma^*$. Thus, the answer is either yes or no.

(EQ) $\mathcal{L}(\mathcal{H}) = \mathcal{L}$? for a given DFA $\mathcal{H}$. Again, the answer is either yes or no. If the answer is no, one also gets a counterexample word from the symmetric difference $\mathcal{L}(\mathcal{H}) \oplus L$.

Essentially, L* asks MQs until it considers that it has a consistent data set to come up with a hypothesis DFA $\mathcal{H}$ (Line 8), which then undergoes an EQ (Line 9). If the latter succeeds, then the algorithm stops. Otherwise, the counterexample and possibly more membership queries are used to refine the hypothesis. The algorithm

Algorithm 2: L*- algorithm.

Input: An oracle providing a membership query $MQ : \Sigma^* \mapsto \{T, F\}$, and a equivalence query $EQ : \textbf{DFA} \mapsto \Sigma^* \times \{T, F\}$
Data: A_T, A_F the minimal automata with $\mathcal{L}(A_T) = \Sigma^*$ and $\mathcal{L}(A_F) = \emptyset$
Output: A finite automaton $\mathcal{H}$

```
1  if MQ(ε) then
2  |  H ← A_T;
3  else
4  |  H ← A_F;
5  end
6  Counter, Equal ← EQ(H);
7  while not Equal do
8  |  H ← Refine(H, MQ, Counter);
9  |  Counter, Equal ← EQ(H);
10 end
11 return H
```

provides the following guarantee: If MQs and EQs are answered according to a given regular language $\mathcal{L} \subseteq \Sigma^*$, then the algorithm eventually outputs, after polynomially many steps, the unique minimal DFA $\mathcal{H}$ such that $\mathcal{L}(\mathcal{H}) = \mathcal{L}$. The heart of this algorithm lies in procedure of the refinement of the current hypothesis Line 8 according to a counter example and membership queries. There are several ways to implement this process. In the original paper [10] Angluin maintains

an observation table which can represent DFA and can be refined by a counter example. In this work we mainly use the process developed by Kearns et al. in [88]. Their main idea is to use the Nerode congruence and Theorem 2.3.4. Denote by $\mathcal{L}$ the regular language we are attempting to learn. They maintain two sets:

1. State access words S. Representatives of equivalence classes of the Nerode congruence.
2. Distinguishing words D. A set of distinguishing words, such that, for any $s, s' \in S$, there exists a distinguishing word $d \in D$.

These two sets are stored in a *binary decision tree* $\mathcal{T}$, where S are the leaves and D are the inner vertices. Given a word w, the decision made in each inner vertex is answering whether $wd \in \mathcal{L}$. Therefore, for a given tree $\mathcal{T}$ we can assign for any word w a state $s \in S$, denote it by $\text{Sift}(\mathcal{T}, w)$. Figure 2.6, illustrates a run of the algorithm learning the language $\mathcal{L}_2 = \{(ab)^n \mid n \in \mathbb{N}\}$ and presents two trees $\mathcal{T}$, and $\mathcal{T}'$, for which $\text{Sift}(\mathcal{T}, aa) = a$ and $\text{Sift}(\mathcal{T}', aa) = aa$. An invariant property of

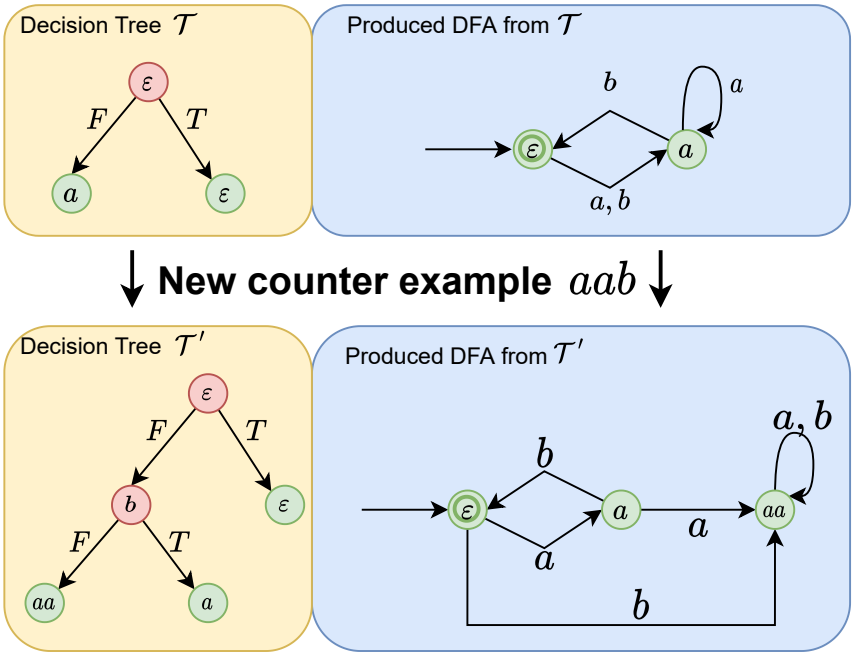


Figure 2.6: Partial run of the Kearns et al. L^* algorithm for the language $\{(ab)^n \mid n \in \mathbb{N}\}$.

the decision trees, which are built during the algorithm, is that two words $w \sim_{\mathcal{L}} w'$ are assigned to the same state by the decision trees. Assuming that the language

is not universal or empty, there exists a word w such that $w \in \mathcal{L}$ if and only if $\varepsilon \notin \mathcal{L}$. Therefore, the initial decision tree consists of the root ε with two children w and ε (see $\mathcal{T}$ in Figure 2.6).

To create a hypothesis DFA from these trees, one takes the set S to be the states of the DFA. The transition function δ is defined on each $(s, a) \in S \times \Sigma$ to be the state $\text{Sift}(\mathcal{T}, sa) = s'$. The initial state is the state ε and the final state are all the states on the right side of the tree. See for example the trees and their DFA in Figure 2.6.

Now, assume we performed an equivalence check and received a counter example w. It means that there is an equivalence class we did not discover yet, more precisely there exists a $s \in S$ which represents more than one class of $\sim_{\mathcal{L}}$. Therefore, the algorithm splits one of the previously known states (i.e., leaf of the tree) and adds a new distinguishing word (inner node) to separate these new states. This is done by taking the counter example w and checking the smallest prefix w_{pre} for which $s = \text{Sift}(\mathcal{T}, w_{pre}) \neq \delta(q_0, w_{pre}) = s'$ in the current hypothesis DFA and Tree $\mathcal{T}$. Let w_d be the distinguishing word between s' and s, and denote by $w_{pre} = w'_{pre}a$. We change the leaf s' in to an inner vertex with word $w_d a$, add it two new leafs s and w'_{pre}. In Figure 2.6, the counter-example aab, splits the node a into aa and a and add the distinguishing word $b = \varepsilon b$.

Therefore, every round, we add a single state which does not belong to any other equivalence class of the original language. When all classes have been discovered the algorithm terminates and this happens in finite time, since there are finitely many class in a regular language by Theorem 2.3.4. Moreover, since we have representatives of the classes and distinguishing words, the current DFA is the minimal DFA.

An issue may rise if one wants to use this algorithm to learn a regular language from a black box, where an equivalence query cannot be implemented. In this case we can use a probabilistic version of Angluin's algorithm. In Angluin's original paper [10], she designs a type of *probably approximately correct* (PAC) version of her algorithm, where the equivalence query is replaced with a large set of random membership queries. The size of this set, $StEq$, depends on three variables $0 < \varepsilon, \delta < 1$ and the number of equivalence queries already performed n:

$$StEq(n, \varepsilon, \delta) = \frac{1}{\varepsilon}\left(\log(\frac{1}{\delta}) + \log(2)(n+1)\right).$$

By replacing the equivalency queries with random $StEq(n, \varepsilon, \delta)$ membership queries we get the PAC version of the algorithm, for which we get the following guarantee:

Theorem 2.3.18 ([10]). *Let P be a probability distribution on Σ^*, $\mathcal{L}$ a regular language, and $0 < \varepsilon, \delta < 1$. The PAC version of Angluin's algorithm terminates*

outputting a DFA $\mathcal{A}$, such that with probability greater than $1-\delta$:

$$\sum_{w \in \mathcal{L} \oplus \mathcal{L}(\mathcal{A})} P(w) \leq \varepsilon$$

2.4 Markov Chains

The following is a short summary of some results and definitions for Markov chains. We add it here as it is an integral part of a proof in Chapter 11.

Definition 2.4.1. A sequence $\{X_n\}_{n=0}^{\infty}$ of random variables is called a *Markov Chain* (MC) on the state space S if it satisfies, $\forall n \in \mathbb{N}, \{x_j\}_{j=0}^{n+1} \subseteq S$:

$$P\left(X_{n+1} = x_{n+1} | X_0 = x_0, X_1 = x_1, ..., X_n = x_n\right) = P\left(X_{n+1} = x_{n+1} | X_n = x_n\right)$$

We call a MC time homogeneous if for all $n \in \mathbb{N}$, $P\left(X_{n+1} = x_{n+1} \mid X_n = x_n\right) = P\left(X_1 = y \mid X_0 = x\right)$. We call it finite if $|S| < \infty$. From now on we assume that all *MCs are time homogeneous and finite.* Let the transition function be the stochastic matrix $\mathcal{P}$, such that $\mathcal{P}(x,y) = P(X_{n+1} = y \mid X_n = x)$. To examine the behavior of a MC we need to know how to "start" the chain at X_0. Let the *initial distribution* be $\pi_0\left(x\right) = P\left(X_0 = x\right)$ a probability measure on S. Denote by:

$$P_{\pi_0}\left(X_0 = x_0, ..., X_n = x_n\right) = \pi_0\left(x_0\right) \mathcal{P}\left(x_0, x_1\right) \cdots \mathcal{P}\left(x_{n-1}, x_n\right)$$

For a state $x \in S$, we denote by π_x initial distribution such that $\pi_0(x) = 1$. Given $x \in S$ we call the random value $T_y = \inf\left\{n \geq 0 | X_n \in A\right\}$ the *hitting time* of y. Denote by $\rho(x,y) = P_{\pi_x}(T_y < \infty)$ the probability of reaching y from x in finite time. Given two states $x, y \in S$, we say that x leads to y, i.e. $x \to y$, if $\rho_{x,y} > 0$. We call S *irreducible* if $\forall x, y \in S$, $x \to y$ and $y \to x$.

We call the measure π on S an invariant probability for the MC if $\pi\mathcal{P} = \pi$.

Lemma 2.4.2. *Given an irreducible MC, then the invariant measure exists and it is unique.*

For $x \in S$ satisfying $\mathcal{P}^n\left(x,x\right) > 0$ for some $n > 0$, call $d_x = gcd\left(\left\{n > 1 | \mathcal{P}^n\left(x,x\right) > 0\right\}\right)$ is the period of x.

Lemma 2.4.3. *If the MC is irreducible, then $\forall x, y \in S$, $d_x = d_y$.*

The Period of the MC is then defined to be the period of all states. We call a irreducible MC aperiodic if it is of period one.

Theorem 2.4.4. *Given a finite, irreducible, and aperiodic, MC with invariant distribution π, then:*

$$\forall x, y \in S \ \lim_{n \to \infty} \mathcal{P}^n\left(x,y\right) = \pi\left(y\right)$$

Given a period $d > 1$, one can divide the states into d equivalence classes $S = S_0 \cup S_1 \cdots S_{d-1}$, such that $\forall i < d$ and $\forall j \neq (i+1) \bmod d$ one has:

$$\forall x \in S_i \,, \forall y \in S_j \quad \mathcal{P}(x, y) = 0$$

Moreover, the MC on the states S_i with the probability distribution $\mathcal{P}^n$ is an irreducible MC.

Part I

Cover of Petri Nets

Chapter 3

Cover and Commodification of Accelerations

Recall that the *Cover* (also denoted as coverability set) of a Petri net with an initial marking is the downward closure (for the usual order on integer vectors) of the set of reachable markings. An effective finite representation of the cover makes it possible to decide several problems, such as: Can a given marking be covered by a reachable marking (the coverability problem)? Is the set of reachable markings finite? Which places are unbounded?

In 1969, Karp and Miller showed that a finite representation of the coverability set of Petri nets and vector addition systems is computable by an algorithm (the `K-M` algorithm) constructing a finite tree [85] whose finite set of vertex labels (ω-markings) C represents the cover. Specifically, the downward closure (in $\mathbb{N}^P$) of C, coincides with the cover.

The original proof of `K-M` algorithm is incomplete as Hack had already noted in 1974 [68]. Motivated by the lack of complete and certified proof, Yamamoto et al. wrote a formal COQ proof of the correctness of `K-M` algorithm [160].

In this chapter we give a simple and elegant proof of the `K-M` algorithm based on three new concepts: *abstraction*, *acceleration* and *sequence of exploration*. In particular, we transform the accelerations of the `K-M` algorithm to first-class citizens (i.e., commodification of the accelerations) instead of using them implicitly. Then we propose an "accelerated" variant of the `K-M` algorithm with an expected gain in execution time.

An abstraction is an ω-transition (i.e., a generalized transition), where (1) its backward incidence and incidence with respect to a place can be equal to ω (i.e., belonging to $\mathbb{N}_\omega$) and (2) which has an infinite family of transition sequences "justifying" the introduction of the ω's. We show that their firing from a ω-marking whose associated ideal is included in Cover leads to a ω-marking whose associated ideal is also included in Cover. We then prove that the concatenation of abstrac-

tions is still an abstraction. An acceleration is an abstraction whose incidence with respect to each place is either zero or ω. We establish that any abstraction can be transformed into an acceleration by substituting the strictly positive components of the incidence by ω and requiring ω tokens for the strictly negative components of the incidence.

The proof of the K-M algorithm becomes rather simple, with the addition of ghost variables (i.e., variables without effect on the execution of the algorithm). As usual, the proof of the termination is based on the well order of $\mathbb{N}_\omega^P$. The proof of consistency is an almost immediate consequence of the properties of abstractions and accelerations. The proof of completeness is based on the notion of exploration sequences, detailed below.

We then deepen our study of accelerations. The set of accelerations provided with a natural order is a well order, i.e., every upward closed set can be represented by a finite set of minimal elements. We show that the integer coefficients of the minimal accelerations are bounded by $B(e, d)$, which is polynomial in the size of the incidence matrices e and doubly exponential in the number of places d. We also show how to transform (*truncate*) any acceleration into an acceleration whose integer coefficients are bounded by $B(e, d)$. We then propose an accelerated version of the algorithm of K-M with an expected gain in the execution time. The general principle is as follows: when you discover an acceleration, you truncate it and memorize it. Then at each step of the algorithm, the marking of the current node is increased by firing the accelerations that can be fired. Due to the truncation of the accelerations, our accelerated version of the K-M algorithm requires a minimal additional cost in memory (2−EXP), compared to the general cost memory of the K-M algorithm which is non-primitive recursive. In addition, the proof of the correctness of our accelerated variant is immediately deduced from our original proof.

Organization. In Section 3.1 we introduce and study abstractions and accelerations of a Petri net. We then establish the proof of the K-M algorithm in Section 3.2. In Section 3.3 we describe an improved version of the K-M algorithm.

Based on. This chapter is mainly based on our work in [61].

3.1 Covering and Abstractions

Let $\mathcal{N} = \langle P, T, \mathbf{Pre}, \mathbf{C} \rangle$ be a Petri net. In order to introduce abstractions and accelerations, we generalize the transitions to take into account place markings with ω tokens.

Definition 3.1.1. Let P be a set of places. An *ω-transition* $\mathbf{a}$ is defined by two vectors:

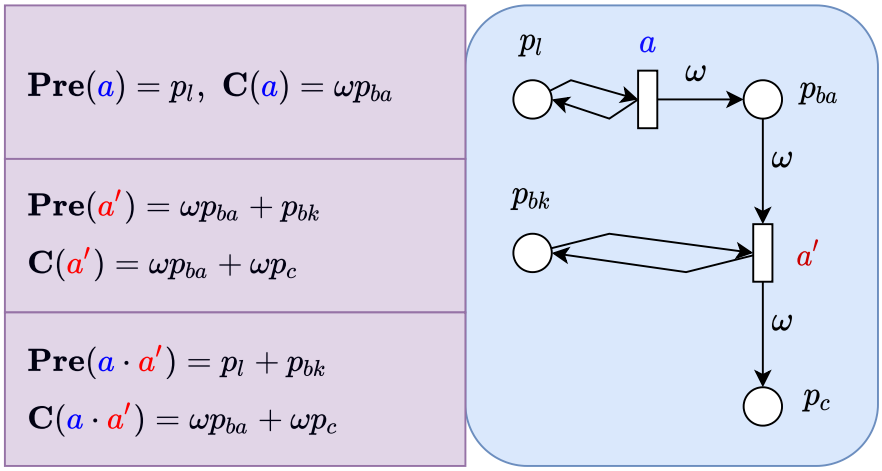


Figure 3.1: Concatenation of abstractions

- $\mathbf{Pre}(\mathbf{a}) \in \mathbb{N}_\omega^P$ its *backward incidence*;
- $\mathbf{C}(\mathbf{a}) \in \mathbb{Z}_\omega^P$ its *incidence* with $\mathbf{Pre}(\mathbf{a}) + \mathbf{C}(\mathbf{a}) \geq 0$.

For simplicity, we denote $\mathbf{Pre}(\mathbf{a})(p)$ (resp. $\mathbf{C}(\mathbf{a})(p)$) by $\mathbf{Pre}(p, \mathbf{a})$ (resp. $\mathbf{C}(p, \mathbf{a})$). An ω-transition $\mathbf{a}$ is fireable from an ω-marking $\mathbf{m} \in \mathbb{N}_\omega^P$ if $\mathbf{m} \geq \mathbf{Pre}(\mathbf{a})$. When $\mathbf{a}$ is fired from an ω-marking $\mathbf{m}$, it leads to an ω-marking $\mathbf{m}' \stackrel{\text{def}}{=} \mathbf{m} + \mathbf{C}(\mathbf{a})$, which we denote $\mathbf{m} \xrightarrow{\mathbf{a}} \mathbf{m}'$. Note that if $\mathbf{Pre}(p, \mathbf{a}) = \omega$ then whatever the value of $\mathbf{C}(p, \mathbf{a})$ is, one has $\mathbf{m}'(p) = \omega$. So without loss of generality, we suppose that for all ω-transition $\mathbf{a}$, $\mathbf{Pre}(p, \mathbf{a}) = \omega$ implies $\mathbf{C}(p, \mathbf{a}) = \omega$.

In order to define abstractions, we define the incidence of a sequence of ω-transitions σ by recurrence over its length. Like previously, we introduce $\mathbf{Pre}(p, \sigma) \stackrel{\text{def}}{=} \mathbf{Pre}(\sigma)(p)$ and $\mathbf{C}(p, \sigma) \stackrel{\text{def}}{=} \mathbf{C}(\sigma)(p)$. The base case corresponds to the definition of an ω-transition. Let $\sigma = t\sigma'$, with t an ω-transition and σ' a sequence of ω-transitions, then:

- $\mathbf{C}(\sigma) = \mathbf{C}(t) + \mathbf{C}(\sigma')$;
- For all $p \in P$
  - if $\mathbf{C}(p, t) = \omega$ then $\mathbf{Pre}(p, \sigma) = \mathbf{Pre}(p, t)$;
  - else $\mathbf{Pre}(p, \sigma) = \max(\mathbf{Pre}(p, t), \mathbf{Pre}(p, \sigma') - \mathbf{C}(p, t))$.

The sequence σ is fireable from $\mathbf{m}$ if and only if $\mathbf{m} \geq \mathbf{Pre}(\sigma)$. In this case, $\mathbf{m} \xrightarrow{\sigma} \mathbf{m} + \mathbf{C}(\sigma)$. For an example, see Figure 3.1, where we have the $\mathbf{Pre}$ and $\mathbf{C}$ of two omega transitions $\mathbf{a}$ and $\mathbf{a}'$ and their concatenation $\mathbf{a} \cdot \mathbf{a}'$.

An *abstraction* of a PN is an ω-transition which concisely reflects the behavior of the net from the point of view of coverability (see Proposition 3.1.3). We note that a transition t of a PN is by construction (with $\sigma_n = t$) an abstraction.

Definition 3.1.2. Let $\mathcal{N} = \langle P, T, \mathbf{Pre}, \mathbf{C}\rangle$ be a PN and $\mathbf{a}$ be an ω-transition.

The ω-transition $\mathbf{a}$ is an *abstraction* if for all $n \geq 0$, there exists $\sigma_n \in T^*$ such that for all $p \in P$ with $\mathbf{Pre}(p, \mathbf{a}) \in \mathbb{N}$, we have:

1. $\mathbf{Pre}(p, \sigma_n) \leq \mathbf{Pre}(p, \mathbf{a})$;
2. if $\mathbf{C}(p, \mathbf{a}) \in \mathbb{Z}$ then $\mathbf{C}(p, \sigma_n) \geq \mathbf{C}(p, \mathbf{a})$;
3. if $\mathbf{C}(p, \mathbf{a}) = \omega$ then $\mathbf{C}(p, \sigma_n) \geq n$.

The following proposition justifies the interest of abstractions.

Proposition 3.1.3. *Let $(\mathcal{N}, \mathbf{m}_0)$ be a marked PN, $\mathbf{a}$ be an abstraction and $\mathbf{m}$ be an ω-marking such that: $[\![\mathbf{m}]\!] \subseteq Cover(\mathcal{N}, \mathbf{m}_0)$ and $\mathbf{m} \xrightarrow{\mathbf{a}} \mathbf{m}'$. Then $[\![\mathbf{m}']\!] \subseteq Cover(\mathcal{N}, \mathbf{m}_0)$.*

Proof. Let $\mathbf{m}^* \in [\![\mathbf{m}']\!]$. Denote $n = \max(\mathbf{m}^*(p) \mid \mathbf{m}'(p) = \omega)$ and
$\ell = \max(\mathbf{Pre}(p, \sigma_n), n - \mathbf{C}(p, \sigma_n) \mid \mathbf{m}(p) = \omega)$.
Define $\mathbf{m}^\sharp \in [\![\mathbf{m}]\!]$ by:

- If $\mathbf{m}(p) < \omega$ then $\mathbf{m}^\sharp(p) = \mathbf{m}(p)$;
- Otherwise $\mathbf{m}^\sharp(p) = \ell$.

Let us check that σ_n is fireable from $\mathbf{m}^\sharp$. For any $p \in P$,

- If $\mathbf{m}(p) < \omega$ then $\mathbf{m}^\sharp(p) = \mathbf{m}(p) \geq \mathbf{Pre}(p, \mathbf{a}) \geq \mathbf{Pre}(p, \sigma_n)$;
- Otherwise $\mathbf{m}^\sharp(p) = \ell \geq \mathbf{Pre}(p, \sigma_n)$.

Let us check that $\mathbf{m}^\sharp + \mathbf{C}(\sigma_n) \geq \mathbf{m}^*$. For any $p \in P$,

- If $\mathbf{m}(p) < \omega$ and $\mathbf{C}(p, \mathbf{a}) < \omega$ then
  $\mathbf{m}^\sharp(p) + \mathbf{C}(p, \sigma_n) \geq \mathbf{m}(p) + \mathbf{C}(p, \mathbf{a}) = \mathbf{m}'(p) \geq \mathbf{m}^*(p)$;
- If $\mathbf{m}(p) < \omega$ and $\mathbf{C}(p, \mathbf{a}) = \omega$ then
  $\mathbf{m}^\sharp(p) + \mathbf{C}(p, \sigma_n) \geq \mathbf{C}(p, \sigma_n) \geq n \geq \mathbf{m}^*(p)$;
- If $\mathbf{m}(p) = \omega$ then $\mathbf{m}^\sharp(p) + \mathbf{C}(p, \sigma_n) \geq n - \mathbf{C}(p, \sigma_n) + \mathbf{C}(p, \sigma_n) = n \geq \mathbf{m}^*(p)$.

□

A simple way to build new abstractions consists in concatenating them.

Proposition 3.1.4. *Let $\mathcal{N} = \langle P, T, \mathbf{Pre}, \mathbf{C}\rangle$ be a PN and σ a sequence of abstractions. Then the ω-transition $\mathbf{a}$ defined by $\mathbf{Pre}(\mathbf{a}) = \mathbf{Pre}(\sigma)$ and $\mathbf{C}(\mathbf{a}) = \mathbf{C}(\sigma)$ is an abstraction.*

Proof. We show this result by recurrence on the length of σ. The base case is immediate. Let $\sigma = b\sigma'$ and (by the recursion hypothesis) let $\{\sigma'_n\}_{n\in\mathbb{N}}$ a family of sequences of transitions associated with σ'. Let $\{\sigma_{n,b}\}_{n\in\mathbb{N}}$ a family of sequences of transitions associated with b. Fix $n \in \mathbb{N}$.
Denote by $n' = \max(n, \max(n - \mathbf{C}(p,b) \mid \mathbf{C}(p,b) < \omega = \mathbf{C}(p,\sigma')))$.
Denote $\ell = \max(\mathbf{Pre}(p,\sigma'_{n'}), n - \mathbf{C}(p,\sigma'_{n'}) \mid \mathbf{Pre}(p,b) < \omega = \mathbf{C}(p,b))$.
Let us show that $\sigma_{\ell,b}\sigma'_{n'}$ satisfies the three conditions of Definition 3.1.2.

First recall, that we set the ω-transitions in such a way that for any $p \in P$, if $\mathbf{Pre}(p,\mathbf{a}) = \omega$ then $\mathbf{C}(p,\mathbf{a}) = \omega$. Hence, we are only interested in places such that $\mathbf{Pre}(p,\mathbf{a}) < \omega$. Let $p \in P$, $\mathbf{Pre}(p,\mathbf{a}) < \omega$ if and only if
(1) $\mathbf{Pre}(p,b) < \omega$ and $\mathbf{C}(p,b) = \omega$, or
(2) $\mathbf{Pre}(p,b) < \omega$, $\mathbf{C}(p,b) < \omega$ and $\mathbf{Pre}(p,\sigma') < \omega$.

1. **Case $\mathbf{Pre}(p,b) < \omega$ and $\mathbf{C}(p,b) = \omega$.**
   Therefore $\mathbf{Pre}(p,\mathbf{a}) = \mathbf{Pre}(p,b)$ and $\mathbf{C}(p,\mathbf{a}) = \omega$.
   We thus have $\mathbf{Pre}(\sigma_{\ell,b}) \leq \mathbf{Pre}(p,b) = \mathbf{Pre}(p,\mathbf{a})$.
   Moreover, $\mathbf{Pre}(p,\sigma_{\ell,b}) + \mathbf{C}(p,\sigma_{\ell,b}) \geq \mathbf{C}(p,\sigma_{\ell,b}) \geq \ell \geq \mathbf{Pre}(p,\sigma'_{n'})$ showing condition *ii.* holds.
   Finally, $\mathbf{C}(p,\sigma_{\ell,b}) + \mathbf{C}(p,\sigma'_{n'}) \geq \ell + \mathbf{C}(p,\sigma'_{n'}) \geq n - \mathbf{C}(p,\sigma'_{n'}) + \mathbf{C}(p,\sigma'_{n'}) \geq n$ fulfilling condition *iii.*
2. **Case $\mathbf{Pre}(p,b) < \omega$, $\mathbf{C}(p,b) < \omega$ and $\mathbf{Pre}(p,\sigma') < \omega$.**
   Therefore, $\mathbf{Pre}(p,\mathbf{a}) = \max(\mathbf{Pre}(p,b), \mathbf{Pre}(p,\sigma') - \mathbf{C}(p,b))$ and:

$$\begin{aligned}\mathbf{Pre}(p,\sigma_{\ell,b}\sigma'_{n'}) &= \max(\mathbf{Pre}(p,\sigma_{\ell,b}), \mathbf{Pre}(p,\sigma'_{n'}) - \mathbf{C}(p,\sigma_{\ell,b})) \\ &\leq \max(\mathbf{Pre}(p,b), \mathbf{Pre}(p,\sigma') - \mathbf{C}(p,b)) = \mathbf{Pre}(p,\mathbf{a})\end{aligned}$$

   showing condition *ii.* holds. In order to show that condition *ii.* holds, there are two subcases to be considered:

   - $\mathbf{C}(p,\sigma') < \omega$. So, $\mathbf{C}(p,\mathbf{a}) = \mathbf{C}(p,b) + \mathbf{C}(p,\sigma')$
     and $\mathbf{C}(p,\sigma_{\ell,b}\sigma'_n) = \mathbf{C}(p,\sigma_{\ell,b}) + \mathbf{C}(p,\sigma'_n) \geq \mathbf{C}(p,b) + \mathbf{C}(p,\sigma') = \mathbf{C}(p,\mathbf{a})$.
   - $\mathbf{C}(p,\sigma') = \omega$. So, $\mathbf{C}(p,\mathbf{a}) = \omega$
     and $\mathbf{C}(p,\sigma_{\ell,b}\sigma'_{n'}) = \mathbf{C}(p,\sigma_{\ell,b}) + \mathbf{C}(p,\sigma'_{n'}) \geq \mathbf{C}(p,b) + n - \mathbf{C}(p,b) = n$.

Therefore, $\mathbf{a}$ is an abstraction. □

We now introduce the concept underlying the construction of Karp and Miller.

Definition 3.1.5. Let $\mathcal{N} = \langle P, T, \mathbf{Pre}, \mathbf{C}\rangle$ be a PN. An *acceleration* is an abstraction $\mathbf{a}$ such that $\mathbf{C}(\mathbf{a}) \in \{0,\omega\}^P$.

The following proposition provides a way of obtaining acceleration from any abstraction.

Proposition 3.1.6. *Let $\mathcal{N} = \langle P, T, \mathbf{Pre}, \mathbf{C}\rangle$ be a PN and $\mathbf{a}$ be an abstraction. Define $\mathbf{a}'$ an ω-transition by for all $p \in P$*

- *If $\mathbf{C}(p,\mathbf{a}) < 0$ then $\mathbf{Pre}(p,\mathbf{a}') = \mathbf{C}(p,\mathbf{a}') = \omega$;*
- *If $\mathbf{C}(p,\mathbf{a}) = 0$ then $\mathbf{Pre}(p,\mathbf{a}') = \mathbf{Pre}(p,\mathbf{a})$ and $\mathbf{C}(p,\mathbf{a}') = 0$;*
- *If $\mathbf{C}(p,\mathbf{a}) > 0$ then $\mathbf{Pre}(p,\mathbf{a}') = \mathbf{Pre}(p,\mathbf{a})$ and $\mathbf{C}(p,\mathbf{a}') = \omega$.*

Then $\mathbf{a}'$ is an acceleration.

Proof. Consider $\{\sigma_n\}_{n\in\mathbb{N}}$ a family associated with the abstraction $\mathbf{a}$. We now show that the family $\{\sigma_n^n\}_{n\in\mathbb{N}}$ satisfies the conditions of Definition 3.1.2 for $\mathbf{a}'$. For all $n \in \mathbb{N}$:

- Let $p \in P$ such that $\mathbf{Pre}(p,\mathbf{a}') < \omega$.
  This implies that $\mathbf{C}(p,\mathbf{a}) \geq 0$ and that $\mathbf{Pre}(p,\mathbf{a}') = \mathbf{Pre}(p,\mathbf{a})$.
  Since, $\mathbf{C}(p,\sigma_n) \geq \mathbf{C}(p,\mathbf{a}) \geq 0$, one has
  $\mathbf{Pre}(p,\sigma_n^n) = \mathbf{Pre}(p,\sigma_n) \leq \mathbf{Pre}(p,\mathbf{a}) = \mathbf{Pre}(p,\mathbf{a}')$;
- Let $p \in P$ such that $\mathbf{C}(p,\mathbf{a}') = 0$. We get that $0 = \mathbf{C}(p,\mathbf{a}) \leq \mathbf{C}(\sigma_n)$.
  Therefore, $0 \leq n\mathbf{C}(\sigma_n) = \mathbf{C}(\sigma_n^n)$;
- Let $p \in P$ such that $\mathbf{Pre}(p,\mathbf{a}') < \omega$ and $\mathbf{C}(p,\mathbf{a}') = \omega$. This implies that $\mathbf{C}(p,\mathbf{a}) > 0$. So $1 \leq \mathbf{C}(p,\mathbf{a}) \leq \mathbf{C}(\sigma_n)$. Therefore, $n \leq n\mathbf{C}(\sigma_n) = \mathbf{C}(\sigma_n^n)$.

□

3.2 Karp and Miller's algorithm

Algorithm 3 is the K-M algorithm enlarged with 'ghost' variables (i.e., having no influence on the behavior of the algorithm) Acc and δ which will simplify the proof. Let us briefly describe this algorithm. It maintains a directed tree $Tr = (V, E, \lambda, \delta)$ whose vertices (V) are labeled by an ω-marking (i.e., $\lambda : V \mapsto \mathbb{N}_\omega^P$) and the edges ($E$) are labeled by a sequence of ω-transitions belonging to $T\mathsf{Acc}^*$ (i.e., $\delta : E \mapsto T\mathsf{Acc}^*$). We extend δ to a mapping from $E^* \mapsto (T\mathsf{Acc}^*)^*$ in the usual way. K-M maintains a subset of the vertices (Front) which are still to be explored. In order to shorten the description of the algorithm, we introduced $\mathsf{Anc}(u)$ the set of ancestors of u (excluding u).

As long as Front is not empty, the algorithm chooses a vertex $u \in$ Front. Then there are three cases. Note that, by "in our version" we mean the introduction of the new ghost variables.

- The marking of u is less than or equal to that of an ancestor u': then u is removed from Front and V and the edge entering u is removed.

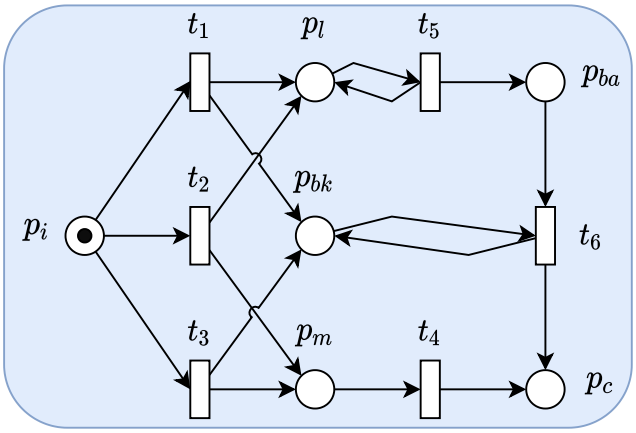


Figure 3.2: An unbounded Petri net.

- The marking of u is greater than that of an ancestor u' and for at least one place p, $\lambda(u')(p) < \lambda(u)(p) < \omega$. For all such places p, we substitute to its marking the value ω. *In our version*, we also define an ω-transition $\mathbf{a}$ by (1) defining it as the sequence of ω-transitions which labels the path from u' to u, (2) applying the transformation of Proposition 3.1.6, and (3) concatenating it with the sequence labeling the incoming edge of u.
- Otherwise, the algorithm determine the fireable transitions and fire them to create the children of u which are inserted in Front. The vertex u is removed from Front. *In our version*, the incoming edge of a new vertex is labeled by the transition that has been fired.

When Front is empty, the algorithm ends.

Example 3.2.1. The Figure 3.3 illustrates the tree of Karp and Miller corresponding to the PN in Figure 3.2. Let us describe its construction during the development of the leftmost branch. From the initial marking, one fires t_1 which leads to $p_l + p_{bk}$, incomparable with $\mathbf{m}_0$. The exploration continues from this marking. The only fireable transition is t_5 whose firing leads to the marking $p_l + p_{bk} + p_{ba}$. An acceleration $\mathbf{a}_1$ is discovered with $\mathbf{Pre}(\mathbf{a}_1) = p_l$ and $\mathbf{C}(\mathbf{a}_1) = \omega p_{ba}$. The current marking is then modified accordingly to the firing of $\mathbf{a}_1$. This vertex is examined again during a subsequent iteration. There is no more acceleration possible. Consequently, one continues the exploration: t_5 and t_6 are fireable. The vertex associated with the firing of t_5 has an identical marking: it will therefore be removed. The vertex associated with firing of t_6 gives rise to a new acceleration: $\mathbf{a}_2$ with $\mathbf{Pre}(\mathbf{a}_2) = p_{bk} + \omega p_{ba}$ and $\mathbf{C}(\mathbf{a}_2) = \omega p_c + \omega p_{ba}$. Since from the last marking only t_5 and t_6 are fireable and lead to the same marking, the exploration of the

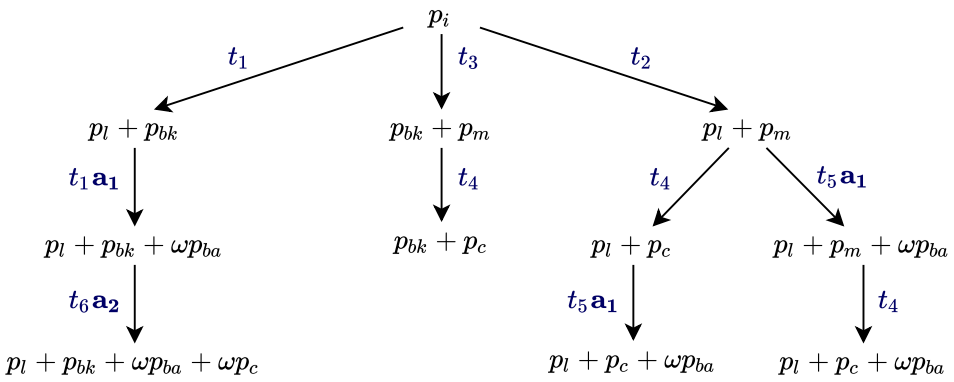


Figure 3.3: Karp and Miller's tree.

branch is stopped. Note that $\mathbf{a}_1$ is re-discovered twice during the construction. The algorithm computed a set of 11 ω-markings labeling the 11 nodes. This set a is finite basis of the Cover.

We will now show the correctness of the K-M algorithm, namely:

- It terminates;
- It is consistent: $\bigcup_{v \in V} [\![\lambda(v)]\!] \subseteq Cover(\mathcal{N}, \mathbf{m}_0)$;
- It is complete: $Cover(\mathcal{N}, \mathbf{m}_0) \subseteq \bigcup_{v \in V} [\![\lambda(v)]\!]$.

The termination is based on the fact that $\mathbb{N}_\omega^P$ is well-ordered.

Proposition 3.2.2 (termination)**.** *The K-M algorithm (Algorithm 3) terminates.*

Proof. By contradiction, suppose the K-M algorithm does not terminate. A vertex of the tree can only be chosen in the loop at most $|P|+1$ times. In fact, it remains in Front only if it has been accelerated, which implies that the associated marking has, at least, one more component which is equal to ω.
Consequently, the algorithm builds a tree with infinitely many vertices. Each vertex has at most $|T|$ children. By application of König's lemma, this tree has an infinite branch.
Let $\mathbf{m}_0, \mathbf{m}_1, \ldots$ be the markings associated with the vertices of this branch. Recall that, $\mathbb{N}_\omega^P$ is well-ordered. Therefore, we can extract an increasing infinite subsequence $\mathbf{m}_{\alpha(0)} \leq \mathbf{m}_{\alpha(1)} \leq \cdots$. There cannot be an equality between two consecutive vertices because then the second vertex would have been removed, hence we have a strictly increasing sequence. We therefore deduce that an acceleration has been detected between any two consecutive vertices. Thus, each marking has, at least, one more component which is equal to ω than its predecessor. Therefore, this sequence contains at most $|P| + 1$ elements which contradicts the hypothesis. □

Algorithm 3: The Karp and Miller algorithm

```
KarpMiller(𝒩, m_0)
Input: A marked PN (𝒩, m_0)
Data: V a set of vertices; E ⊆ V × V; Front ⊆ V; λ : V → ℕ_ω^p;
      δ : E → TAcc*;
Tr = (V, E, λ, δ) a labeled tree; Acc a set of ω-transitions;
u, u', u'' vertices; a an ω-transition with non negative incidence;
Output: A labeled tree Tr = (V, E, λ, δ)
1  V ← {r}; E ← ∅; Front ← {r}; λ(r) ← m_0; Acc ← ∅;
2  while Front ≠ ∅ do
3      Choose u ∈ Front
4      if ∃u' ∈ Anc(u) s.t. λ(u') ≥ λ(u) then
5          Front ← Front \ {u} ; V ← V \ {u} ; E ← E \ V × {u} // λ(u) is
            covered
6      else if ∃u' ∈ Anc(u) s.t. λ(u') < λ(u) ∧ ∃p λ(u')(p) < λ(u)(p) < ω
        then
           // An acceleration is found between u and its ancestors
              u'
7          Let γ ∈ E* be the path from u' to u in Tr
8          a ← NewAcceleration()
9          foreach p ∈ P do
10             if λ(u')(p) < λ(u)(p) then λ(u)(p) ← ω
11             if C(p, δ(γ)) < 0 then Pre(p, a) ← ω; C(p, a) ← ω
12             if C(p, δ(γ)) = 0 then Pre(p, a) ← Pre(p, δ(γ)); C(p, a) ← 0
13             if C(p, δ(γ)) > 0 then Pre(p, a) ← Pre(p, δ(γ)); C(p, a) ← ω
14         end
15         Let (u'', u) be the incoming arc of u in Tr
16         δ((u'', u)) ← δ((u'', u)) · a; Acc ← Acc ∪ {a}
17     else
18         Front ← Front \ {u}
19         foreach t ∈ T s.t. λ(u) ≥ Pre(t) do
               // Adding the children of u
20             u' ← NewNode(); V ← V ∪ {u'}; Front ← Front ∪ {u'};
                E ← E ∪ {(u, u')}
21             λ(u') ← λ(u) + C(t); δ((u, u')) ← t
22         end
23     end
24 end
25 return Tr
```

The following lemma illustrates the advantage of the introduction of ghost variables.

Lemma 3.2.3. *For every edge $(u, v) \in E$, we have $\lambda(u) \xrightarrow{\delta(u,v)} \lambda(v)$.*

Proof. There are two cases to be considered:

• **The creation of (u, v).** This happens during the construction of the successors of u. Consequently, there exists a transition $t \in T$ such that, $\lambda(u) \xrightarrow{t} \lambda(v)$ and this transition labels the edge (u, v) .

• **The modification of $\lambda(v)$.** This happens when the algorithm discovers an acceleration $\mathbf{a}$ between an v and an ancestor u' of v. We denote by $\mathbf{m}^-$ the marking associated with v before its update and $\mathbf{m}^+$ the marking associated with v after its update. By induction, the sequence of ω-transitions along the path from u' to v is fireable from $\lambda(u')$, hence also from $\mathbf{m}^-$. This sequence has the same precondition that $\mathbf{a}$ has, except possibly on places p where $\mathbf{m}^-(p) = \omega$. So $\mathbf{a}$ is fireable from $\mathbf{m}^-$ and by construction $\mathbf{m}^- \xrightarrow{\mathbf{a}} \mathbf{m}^+$. □

The following lemma is based on the preservation of abstractions by concatenation and the construction of accelerations from abstractions.

Lemma 3.2.4. *Every ω-transition $\mathbf{a} \in \mathsf{Acc}$ is an acceleration.*

Proof. The proof is done by induction according to the order of insertion in Acc. Let $\mathbf{a} \in \mathsf{Acc}$ be an ω-transition. Let us denote by σ the sequence corresponding to the path in the directed tree (created during the run of `K-M`) which led to the creation of $\mathbf{a}$. By the induction hypothesis, σ is a sequence of abstractions. From Proposition 3.1.4, σ is an abstraction (as a concatenation of abstractions). The construction of $\mathbf{a}$ from σ corresponds to Proposition 3.1.6. Therefore, $\mathbf{a}$ is an acceleration. □

The consistency of the algorithm is now a consequence of the previous lemmas.

Proposition 3.2.5 (consistency)**.** *For all $v \in V$, $[\![\lambda(v)]\!] \subseteq Cover(\mathcal{N}, \mathbf{m}_0)$.*

Proof. The proof is done by induction on the length of the path from r to u. The marking associated with the root r is $\mathbf{m}_0$. Hence $[\![\mathbf{m}_0]\!] \subseteq Cover(\mathcal{N}, \mathbf{m}_0)$. Denote by u the parent of v. By the induction hypothesis $[\![\lambda(u)]\!] \subseteq Cover(\mathcal{N}, \mathbf{m}_0)$. By Lemma 3.2.3, $\lambda(u) \xrightarrow{\delta(u,v)} \lambda(v)$. By Lemma 3.2.4, $\delta(u, v)$ is a sequence of abstractions. By Proposition 3.1.4, $\delta(u, v)$ is an abstraction. By Proposition 3.1.3, $[\![\lambda(v)]\!] \subseteq Cover(\mathcal{N}, \mathbf{m}_0)$.

□

In order to ease the proof of completeness, we introduce the notion of sequence of exploration, related to the coverability tree and its construction.

Definition 3.2.6. A sequence of transitions $\mathbf{m} \xrightarrow{\sigma} \mathbf{m}'$ is a *sequence of exploration* of Tr if there exists $v \in \mathsf{Front}$ with $\lambda(v) = \mathbf{m}$ and for all markings $\mathbf{m}''$ visited by the sequence σ and all $v \in V \setminus \mathsf{Front}$, we have : $\mathbf{m}'' \not\leq \lambda(v)$.

For example in Figure 3.4 the sequence σ is an exploration sequence. The

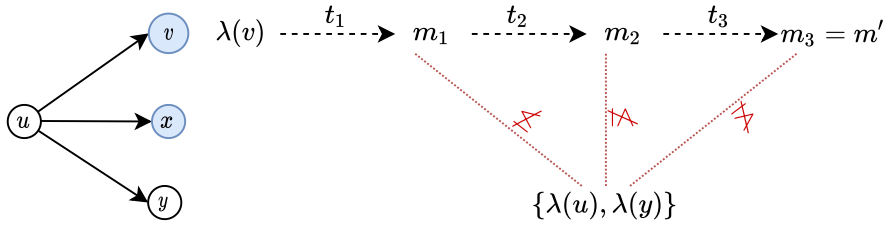


Figure 3.4: Exploration sequence to $\mathbf{m}'$ where $\mathsf{Front} = \{v, x\}$.

definition of exploration sequence brings us to the following definition:

Definition 3.2.7. Let $\mathbf{m}$ be a marking. Then $\mathbf{m}$ is *quasi-covered* if:

- Either there exists $v \in V \setminus \mathsf{Front}$ such that $\mathbf{m} \in [\![\lambda(v)]\!]$;
- Or there exists a sequence of exploration $\mathbf{m}_1 \xrightarrow{\sigma} \mathbf{m}_2 \geq \mathbf{m}$.

We now show that all coverable markings are quasi-covered during the run of the algorithm.

Lemma 3.2.8. *For all $\mathbf{m} \in Cover(\mathcal{N}, \mathbf{m}_0)$ is quasi-covered at the start of each iteration (Line 2) of the main while loop (lines 2 - 26).*

Proof. We establish this result by induction on the number of iterations of the while loop already performed.

• Let the base case be the first time we reach the beginning of the main loop. Since the algorithm is initialized by $V = \mathsf{Front} = \{r\}$ and $\lambda(r) = \mathbf{m}_0$ and for all $\mathbf{m} \in Cover(\mathcal{N}, \mathbf{m}_0)$, there exists a sequence $\mathbf{m}_0 \xrightarrow{\sigma} \mathbf{m}_2 \geq \mathbf{m}$. We get that Assertion 2 holds for the base case.

• Consider the start of an iteration of the loop. Pick $\mathbf{m} \in Cover(\mathcal{N}, \mathbf{m}_0)$. If $\mathbf{m}$ satisfies Assertion 1, it satisfies it until the termination of the algorithm.

Suppose that $\mathbf{m}$ satisfies Assertion 2. Let us denote the sequence of exploration $\mathbf{m}_1 \xrightarrow{\sigma} \mathbf{m}_2 \geq \mathbf{m}$ with $w \in \mathsf{Front}$ where $\lambda(w) = \mathbf{m}_1$. Let us consider the different alternatives.

◦ $\exists u' \in \mathsf{Anc}(u)$ such that $\lambda(u') \geq \lambda(u)$. This implies that $u \neq w$ and the sequence of exploration stays valid.

$\circ$ $\lambda(u)$ is accelerated.

If $u \neq w$, the sequence of exploration is still valid.

If $u = w$ then, since $\lambda(u)$ has been increased, σ is fireable from $\lambda(u)$. Each marking visited is greater than or equal to the corresponding marking of the exploration sequence. So this new sequence is a sequence of exploration that covers $\mathbf{m}$.

$\circ$ u is removed from Front and its children are computed. There are now two subcases to be considered.

- If for all marking $\mathbf{m}'$ visited by σ, $\mathbf{m}' \not\leq \lambda(u)$, the sequence of exploration is still valid.
- Otherwise, consider $\mathbf{m}'$ the last marking visited such that $\mathbf{m}' \leq \lambda(u)$ and the suffix of the sequence $\mathbf{m}' \xrightarrow{\sigma'} \mathbf{m}_2$.
  If $\sigma' = \varepsilon$ then $\mathbf{m} \leq \mathbf{m}_2 = \mathbf{m}' \leq \lambda(u)$. Therefore, Assertion 1 holds for $\mathbf{m}$.
  Otherwise $\mathbf{m}' \xrightarrow{t} \mathbf{m}'' \xrightarrow{\sigma''} \mathbf{m}_2$. Since $\mathbf{m}' \leq \lambda(u)$, u has a child $v \in$ Front such that $\lambda(u) \xrightarrow{t} \lambda(v) \geq \mathbf{m}''$. Therefore, $\lambda(v) \xrightarrow{\sigma''} \mathbf{m}^* \geq \mathbf{m}$ for some $\mathbf{m}^*$ and considering the choice of $\mathbf{m}'$ this sequence is a sequence of exploration.

□

Using the above, we can conclude the completeness of `K-M`:

Proposition 3.2.9 (completeness)**.** *Upon termination of Algorithm 3,*

$$Cover(\mathcal{N}, \mathbf{m}_0) \subseteq \bigcup_{v \in V} [\![\lambda(v)]\!]$$

.

Proof. When Algorithm 3 terminates, Front is empty. The completeness is a consequence of Lemma 3.2.8. □

3.3 An improvement of the algorithm

In order to present an improvement of the algorithm, we deepen the study of accelerations. First, we equip the ω-transitions with an order related to their precondition and incidence.

Definition 3.3.1. Let P be a set of places and let $\mathbf{a}$ and $\mathbf{a}'$ be two ω-transitions. We define the order between them to be:

$$\mathbf{a} \preceq \mathbf{a}' \text{ if and only if } \mathbf{Pre}(\mathbf{a}) \leq \mathbf{Pre}(\mathbf{a}') \wedge \mathbf{C}(\mathbf{a}') \leq \mathbf{C}(\mathbf{a})$$

In other words, $\mathbf{a} \preceq \mathbf{a}'$ if and only if for any ω-marking $\mathbf{m}$ such that $\mathbf{a}'$ is fireable from, then $\mathbf{a}$ is also fireable and firing it leads to an ω-marking greater than or equal to the one reached by $\mathbf{a}'$.

Proposition 3.3.2. *Let $\mathcal{N}$ be a PN. Then the set of abstractions of $\mathcal{N}$ is upward closed. Similarly, the set of accelerations is upward closed in the set of ω-transitions with incidence in $\{0,\omega\}^P$ (also ordered by $\preceq$).*

Proof. Let $\mathbf{a}$ be an abstraction and $\mathbf{a}'$ be an ω-transition such that $\mathbf{a}' \geq \mathbf{a}$. Let $\{\sigma_n\}_{n\in\mathbb{N}}$ be the family of sequences associated with $\mathbf{a}$. Let $n_0 = \max(\mathbf{C}(p,\mathbf{a}') \mid \mathbf{C}(p,\mathbf{a}') \in \mathbb{N})$ where, by convention, $\max(\emptyset) = 0$. We will show that the family $\{\sigma_{\max(n,n_0)}\}_{n\in\mathbb{N}}$ can be associated with $\mathbf{a}'$. Pick p such that $\mathbf{Pre}(p,\mathbf{a}') \in \mathbb{N}$, which implies that $\mathbf{Pre}(p,\mathbf{a}) \in \mathbb{N}$. We get:

- $\mathbf{Pre}(p,\sigma_{\max(n,n_0)}) \leq \mathbf{Pre}(p,\mathbf{a}) \leq \mathbf{Pre}(p,\mathbf{a}')$;
- If $\mathbf{C}(p,\mathbf{a}') \in \mathbb{Z}$ and $\mathbf{C}(p,\mathbf{a}) \in \mathbb{Z}$ then $\mathbf{C}(p,\sigma_{\max(n,n_0)}) \geq \mathbf{C}(p,\mathbf{a}) \geq \mathbf{C}(p,\mathbf{a}')$;
- If $\mathbf{C}(p,\mathbf{a}') \in \mathbb{Z}$ and $\mathbf{C}(p,\mathbf{a}) = \omega$ then $\mathbf{C}(p,\sigma_{\max(n,n_0)}) \geq n_0 \geq \mathbf{C}(p,\mathbf{a}')$;
- If $\mathbf{C}(p,\mathbf{a}') = \omega$ then $\mathbf{C}(p,\mathbf{a}) = \omega$ and $\mathbf{C}(p,\sigma_{\max(n,n_0)}) \geq n$.

The above also applies to accelerations, which finishes the proof. □

Next, we show that the set of accelerations is well-ordered.

Proposition 3.3.3. *Let $\mathcal{N}$ be a PN. Then the set of accelerations of $\mathcal{N}$ with $\preceq$ is well-ordered.*

Proof. The set of accelerations is a subset of $\mathbb{N}^P \times \{0,\omega\}^P$ with the order obtained by the Cartesian product of $(\mathbb{N},\leq)$ and $(\{0,\omega\},\leq)$ (Definition 3.3.1). These sets are well-ordered and since the Cartesian product preserves this property, the proposition follows. □

Let us observe that the set of accelerations is not empty, since it contains the acceleration $\mathbf{a}$ defined by $\mathbf{Pre}(\mathbf{a}) = \mathbf{C}(\mathbf{a}) = 0$ whose associated family $\{\sigma_n\}$ is defined by: for all n, $\sigma_n = \varepsilon$. Since the set of accelerations with the order $\preceq$ is well-ordered, it is equal to the upper closure of the finite set of *minimal* accelerations (minimal according to the order $\preceq$). We now study the maximal size of these minimal accelerations. Given a net, we denote by $d = |P|$ and $e = \max_{p,t}(\max(\mathbf{Pre}(p,t), \mathbf{Pre}(p,t) + \mathbf{C}(p,t))$.

We will use the following result by Jérôme Leroux [103] which gives a bound to the length of the shortest transition sequences which connects two mutually reachable markings $\mathbf{m}_1$ and $\mathbf{m}_2$.

Theorem 3.3.4. *(Theorem 2, [103]) Let $\mathcal{N}$ be a PN, $\mathbf{m}_1, \mathbf{m}_2$ be markings, and σ_1, σ_2 be sequences such that $\mathbf{m}_1 \xrightarrow{\sigma_1} \mathbf{m}_2 \xrightarrow{\sigma_2} \mathbf{m}_1$. Then there exist σ_1', σ_2' such that $\mathbf{m}_1 \xrightarrow{\sigma_1'} \mathbf{m}_2 \xrightarrow{\sigma_2'} \mathbf{m}_1$ where:*

$$|\sigma_1'\sigma_2'| \leq ||\mathbf{m}_1 - \mathbf{m}_2||_\infty (3de)^{(d+1)^{2d+4}}$$

We deduce an upper bound on the size of minimal accelerations. Let $\mathbf{v} \in \mathbb{N}_\omega^P$. We denote by

$$||\mathbf{v}||_\infty = \max(\mathbf{v}(p) \mid \mathbf{v}(p) \in \mathbb{N})$$

Proposition 3.3.5. *Let $\mathcal{N}$ be a PN and let $\mathbf{a}$ be a minimal acceleration. Then*

$$||\mathbf{Pre}(\mathbf{a})||_\infty \leq e(3de)^{(d+1)^{2d+4}}$$

Proof. We consider the net $\mathcal{N}' = \langle P', T', \mathbf{Pre}', \mathbf{C}' \rangle$ obtained from $\mathcal{N}$ by removing the set of places $\{p \mid \mathbf{Pre}(p, \mathbf{a}) = \omega\}$ and adding the set of transitions $T_1 = \{t_p \mid p \in P'\}$ with $\mathbf{Pre}(t_p) = p$ and $\mathbf{C}(t_p) = -p$.
We denote $P_1 = \{p \mid \mathbf{Pre}(p, \mathbf{a}) < \omega = \mathbf{C}(p, \mathbf{a})\}$. Let $\mathbf{m}_1$ the marking obtained by restricting $\mathbf{Pre}(\mathbf{a})$ to P' and $\mathbf{m}_2 = \mathbf{m}_1 + \sum_{p \in P_1} p$. Observe that $d' \leq d$ and that $e' = e$.
Let $\{\sigma_n\}_{n \in \mathbb{N}}$ be the family of sequences associated with $\mathbf{a}$.
Consider $n^* = ||\mathbf{Pre}(\mathbf{a})||_\infty + 1$. Then σ_{n^*} is fireable in $\mathcal{N}'$ from $\mathbf{m}_1$ and its firing reaches a marking that covers $\mathbf{m}_2$. By concatenating transitions of T_1, we obtain a firing sequence in $\mathcal{N}'$ such that $\mathbf{m}_1 \xrightarrow{\sigma_1} \mathbf{m}_2$. By the same process, we obtain a sequence $\mathbf{m}_2 \xrightarrow{\sigma_2} \mathbf{m}_1$.
Applying Theorem 3.3.4, there exists a sequence σ_1' with $\mathbf{m}_1 \xrightarrow{\sigma_1'} \mathbf{m}_2$ and $|\sigma_1'| \leq (3de)^{(d+1)^{2d+4}}$ since $||\mathbf{m}_1 - \mathbf{m}_2||_\infty = 1$. By deleting transitions of T_1 in σ_1', we obtain a sequence $\sigma_1'' \in T^*$ with $\mathbf{m}_1 \xrightarrow{\sigma_1''} \mathbf{m}_2' \geq \mathbf{m}_2$ and $|\sigma_1''| \leq (3de)^{(d+1)^{2d+4}}$.
The ω-transition $\mathbf{a}'$, defined by:

- $\mathbf{Pre}(p, \mathbf{a}') = \mathbf{Pre}(p, \sigma_1'')$ for all $p \in P'$;
- $\mathbf{Pre}(p, \mathbf{a}') = \omega$ for all $p \in P \setminus P'$;
- and $\mathbf{C}(\mathbf{a}') = \mathbf{C}(\mathbf{a})$.

is an acceleration with associated family $\{\sigma_1''^n\}_{n \in \mathbb{N}}$.
By definition of $\mathbf{m}_1$, $\mathbf{a}' \leq \mathbf{a}$. Since $\mathbf{a}$ is minimal, $\mathbf{a}' = \mathbf{a}$.
Since $|\sigma_1''| \leq (3de)^{(d+1)^{2d+4}}$, $||\mathbf{Pre}(\mathbf{a})||_\infty = ||\mathbf{Pre}(\mathbf{a}')||_\infty \leq e(3de)^{(d+1)^{2d+4}}$. □

Using previous results, we show that if we get a "too big" minimal acceleration, we can truncate it.

Proposition 3.3.6. *Let $\mathcal{N}$ be a PN and $\mathbf{a}$ be an acceleration.*
Then the ω-transition $trunc(\mathbf{a})$ is an acceleration, where $trunc(\mathbf{a})$ is defined by:

- $\mathbf{C}(trunc(\mathbf{a})) = \mathbf{C}(\mathbf{a})$;
- *for all p such that $\mathbf{Pre}(p, \mathbf{a}) \neq \omega$, we have* $\mathbf{Pre}(p, trunc(\mathbf{a})) = \min(\mathbf{Pre}(p, \mathbf{a}), e(3de)^{(d+1)^{2d+4}})$;
- *for all p such that $\mathbf{Pre}(p, \mathbf{a}) = \omega$, we have $\mathbf{Pre}(p, trunc(\mathbf{a})) = \omega$.*

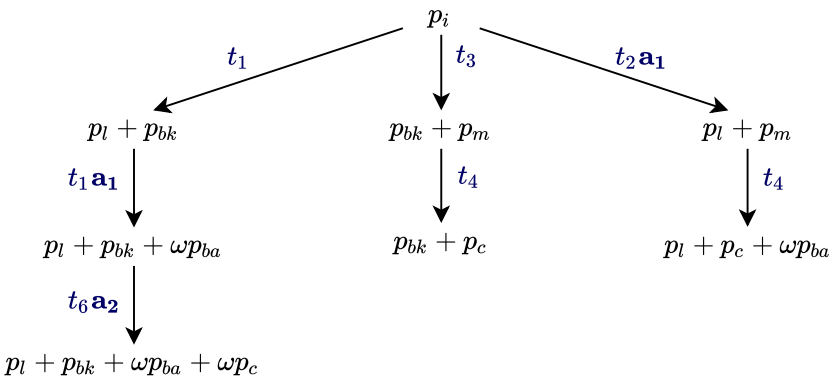


Figure 3.5: An accelerated K-M

Proof. Let $\mathbf{a}' \leq \mathbf{a}$, be a minimal acceleration. For all p such that $\mathbf{Pre}(p, \mathbf{a}) \neq \omega$, $\mathbf{Pre}(p, \mathbf{a}') \leq e(3de)^{(d+1)^{2d+4}}$. Hence $\mathbf{a}' \leq trunc(\mathbf{a})$. Since the set of accelerations is upward closed, we deduce that $trunc(\mathbf{a})$ is an acceleration. □

We are now able to describe the improvement made to the construction of Karp and Miller (see Algorithm 4). First when one discovers an acceleration, one truncates it before inserting it into Acc (line 17). Then, when a vertex of the Front is selected, one first tries to apply the accelerations of Acc to increase its marking (lines 4-6).

Example 3.3.7. Figure 3.5 shows the accelerated K-M tree corresponding to the net of Figure 3.2. When the vertex obtained by firing t_2 from the initial marking is examined, the algorithm evaluates whether firing $\mathbf{a}_1$ or $\mathbf{a}_2$ (both discovered in the left branch) is possible. The acceleration $\mathbf{a}_1$ is fireable and therefore fired.

The proof of termination is unchanged, while the proofs of consistency and completeness require only very minor modifications to integrate the case of the application of accelerations. Below we give an intuition on what needs to change without a full proof. This is due to the fact that we present an even more sophisticated algorithm in the next chapter. The proof of consistency remains valid since the $\omega-$transition $\mathbf{a}$, which is truncated before being added to Acc, is still an acceleration due to Proposition 3.3.6. For completeness one needs to include in the induction step of the proof of Lemma 3.2.8, the case of the algorithm using a previously discovered acceleration.

In practice, applying the memorized accelerations decreases the size of the tree. Furthermore, the cost of memorizing accelerations is largely compensated by this decrease. For the benchmarks showing this claim see Chapter 5

Algorithm 4: An acceleration of the Karp and Miller algorithm

`KarpMillerImproved`$(\mathcal{N}, \mathbf{m}_0)$
Input: A marked PN $(\mathcal{N}, \mathbf{m}_0)$
Data: V set of vertices; $E \subseteq V \times V$; $\mathsf{Front} \subseteq V$; $\lambda : V \to \mathbb{N}^p_\omega$; $\delta : E \to T\mathsf{Acc}^*$;
$Tr = (V, E, \lambda, \delta)$ a labeled tree; Acc a set of ω-transitions;
u, u', u'' vertices; $\mathbf{a}$ an acceleration;
Output: A labeled tree $Tr = (V, E, \lambda, \delta)$

1 $V \leftarrow \{r\}$; $E \leftarrow \emptyset$; $\mathsf{Front} \leftarrow \{r\}$; $\lambda(r) \leftarrow \mathbf{m}_0$; $\mathsf{Acc} \leftarrow \emptyset$;
2 **while** $Front \neq \emptyset$ **do**
3 Choose $u \in \mathsf{Front}$ and let u'' be the predecessor of u
4 **foreach** $\mathbf{a} \in \mathsf{Acc}$ *s.t.* $\lambda(u) \xrightarrow{\mathbf{a}} \lambda(u) + \mathbf{C}(\mathbf{a}) > \lambda(u)$ **do**
5 $\lambda(u) \leftarrow \lambda(u) + \mathbf{C}(\mathbf{a})$; $\delta((u'', u)) \leftarrow \delta((u'', u))\mathbf{a}$
6 **end**
7 **if** $\exists u' \in \mathsf{Anc}(u)$ *s.t.* $\lambda(u') \geq \lambda(u)$ **then**
8 $\mathsf{Front} \leftarrow \mathsf{Front} \setminus \{u\}$; $V \leftarrow V \setminus \{u\}$; $E \leftarrow E \setminus V \times \{u\}$ // $\lambda(u)$ is covered
9 **else if** $\exists u' \in \mathsf{Anc}(u)$ *s.t.* $\lambda(u') < \lambda(u) \wedge \exists p\ \lambda(u')(p) < \lambda(u)(p) < \omega$ **then**
  // An acceleration is found between u and an ancestors of u
10 Let $\gamma \in E^*$ The path from u' to u in Tr
11 $\mathbf{a} \leftarrow$ `NewAcceleration`()
12 **foreach** $p \in P$ **do**
13 **if** $\mathbf{C}(p, \delta(\gamma)) < 0$ **then** $\mathbf{Pre}(p, \mathbf{a}) \leftarrow \omega$; $\mathbf{C}(p, \mathbf{a}) \leftarrow \omega$
14 **if** $\mathbf{C}(p, \delta(\gamma)) = 0$ **then** $\mathbf{Pre}(p, \mathbf{a}) \leftarrow \mathbf{Pre}(p, \delta(\gamma))$; $\mathbf{C}(p, \mathbf{a}) \leftarrow 0$
15 **if** $\mathbf{C}(p, \delta(\gamma)) > 0$ **then** $\mathbf{Pre}(p, \mathbf{a}) \leftarrow \mathbf{Pre}(p, \delta(\gamma))$; $\mathbf{C}(p, \mathbf{a}) \leftarrow \omega$; $\lambda(u)(p) \leftarrow \omega$
16 **end**
17 $\mathbf{a} \leftarrow trunc(\mathbf{a})$
18 $\delta((u'', u)) \leftarrow \delta((u'', u)) \cdot \mathbf{a}$; $\mathsf{Acc} \leftarrow \mathsf{Acc} \cup \{\mathbf{a}\}$
19 **else**
20 $\mathsf{Front} \leftarrow \mathsf{Front} \setminus \{u\}$
21 **foreach** $t \in T$ *s.t.* $\lambda(u) \geq \mathbf{Pre}(t)$ **do**
  // Adding the children of u
22 $u' \leftarrow$ `NewNode`(); $V \leftarrow V \cup \{u'\}$; $\mathsf{Front} \leftarrow \mathsf{Front} \cup \{u'\}$; $E \leftarrow E \cup \{(u, u')\}$
23 $\lambda(u') \leftarrow \lambda(u) + \mathbf{C}(t)$; $\delta((u, u')) \leftarrow t$
24 **end**
25 **end**
26 **end**
27 **return** Tr

From a theoretical point of view, there is at most a doubly exponential number of accelerations each of exponential size: that is to say an additional doubly exponential memory complexity. Recall, that the size of a cover tree is in the worst case non-primitive recursive. Therefore, even without decrease of the size of the tree, the increase of memory size is negligible. Moreover, if the memory space is a strong constraint then it is enough to keep a subset of the accelerations since the proof of the modified algorithm is valid for any subset of accelerations.

Chapter 4

Computing the Clover Efficiently

As shown in the previous chapter, the K-M algorithm computes a finite tree labeled by a finite set of ω-*markings* $C \subseteq \mathbb{N}_\omega^P$ such that C's downward closure restricted to $\mathbb{N}^P$ is the Cover. Therefore, C is *one* among all the possible finite representations of the Cover. Moreover, it is not necessarily minimal in the number of elements because it may contain comparable ω-markings and thus contain redundant items.

Clover, the canonical representation of the cover. However, the set of maximal elements of C is unique and minimal (in size). This set can be defined independently of the K-M algorithm and was called the *minimal coverability set (MCS)* in [56] and abbreviated as the *Clover* in the more general framework of Well Structured Transition Systems (WSTS) [58]. Given the Clover, one can answer coverability questions without rerunning one of the classical coverability algorithms each time. One only needs to compare the desired marking to be covered with the markings in the Clover, which takes time proportional to the size of the Clover. Clover also makes it possible to answer a wider variety of questions beyond coverability and finiteness problems. Let us illustrate this point with the following question on parameterized coverability: Is the marking $(n, 2, 5, n)$ coverable for all $n \geq 0$? This property holds if and only if the ω-marking $(\omega, 2, 5, \omega)$ is smaller than another ω-marking in the Clover, which can be tested in time proportional to the size of the Clover. Let us remark that it can be reduced to the place boundedness problem (noticed by Grégoire Sutre and transmitted to us in a private communication). The question which arises is whether one can find an efficient algorithm to compute the Clover of a marked Petri net.

The minimal coverability tree algorithm. So in [56] the author computes the Clover by modifying the K-M algorithm in such a way that at each step of the algorithm, the set of ω-markings labelling vertices is an antichain. But this aggressive strategy, implemented by the so-called Minimal Coverability Tree algorithm (MCT), contains a subtle bug, it may compute a strict under-approximation of Clover as shown in [57, 65].

Alternative minimal coverability set algorithms. Since the discovery of this bug, three algorithms (with variants) [65, 135, 129] have been designed for computing the minimal coverability set without building the full Karp-Miller tree. In [65] Geeraerts et al. proposed a minimal coverability set algorithm (called `CovProc`) that is not based on the `K-M` algorithm but uses a similar but restricted introduction of ω's. In [135], Reynier et al. proposed a modification of the MCT, called the Monotone-Pruning algorithm (called `MP`), that keeps but "deactivates" vertices labeled with smaller ω-markings while MCT would have deleted them. Recently, in [134], Reynier et al. simplified their original proof of correctness for `MP` and showed how to extend it for some extensions of Petri nets. In [152], Valmari et al. proposed another algorithm (denoted below as `VH`) for constructing the minimal coverability set without deleting vertices. Their algorithm builds a graph and not a tree as usual. In [129], Piipponen et al. improved this algorithm by designing appropriate data structures and heuristics for exploration strategy that may significantly decrease the size of the graph.

In this chapter, we design an algorithm based on (and fixing) the `MCT` generating the Clover of a given net. Despite the current opinion that *"The flaw is intricate and we do not see an easy way to get rid of it... Thus, from our point of view, fixing the bug of the MCT algorithm seems to be a difficult task"* [65], we have found a *simple* modification of `MCT` which makes it correct. It mainly consists in memorizing discovered accelerations and using them as ordinary transitions as described in Section 3.3. Contrary to *all* existing minimal coverability set algorithms that use an *unknown additional memory* that could be non-primitive recursive, we show, that by Theorem 3.3.4, the additional memory required for accelerations, is at most doubly exponential (Proposition 3.3.6).

Organization. In Section 4.1 we review in more details the minimal coverability set algorithms discussed above. Section 4.2 gives an overview and an example run of our algorithm `MinCov`. In Section 4.3 we prove `MinCov` correctness.

Based on. This chapter is mainly based on our work in [60].

4.1 Clover finding algorithms

Throughout the years, major work has been made on algorithms generating of the Clover. Most of them trying to optimize the `K-M` algorithm. The following is a short summary of the relevant history of this work:

MCT

In 1993, Alain Finkel defined a unique minimal coverability graph for marked Petri nets [56]. This graph allowed them to decide the same problems as the tree which the K-M algorithm produces. Given a marked Petri net, the vertices of its minimal coverability graph are the markings of its Clover. Hence, computing this graph, one computes also the Clover. The algorithm given in [56], from now on denoted here by MCT, for computing the minimal coverability graph is based on a new optimization of the K-M algorithm. The main idea of MCT was to use the K-M exploration but keep only the vertices with incomparable and maximal markings. This was achieved by preforming the following actions: Given v the last vertex discovered and $\lambda(v)$ its marking, then:

- If there exists a vertex u (previously discovered) such that its marking has $\lambda(u) \geq \lambda(v)$, then delete v;
- If there exist a vertex, u an ancestor of v with marking strictly smaller $\lambda(v)$. Accelerate the marking of u (the same as in K-M) and remove the entire subtree rooted in u;
- Finally, if there exist a vertex u (not an ancestor of v) with marking strictly smaller $\lambda(v)$. Remove u and the subtree rooted in u.

Unfortunately the MCT algorithm was shown in [57] to produce sometimes an under-approximation of the Clover. This was shown by designing a Petri net for which, if one picks carefully the order of the vertices explored by MCT it computes a set of ω-markings which downward closure in $\mathbb{N}^P$ is a strict subset of the Cover.

CovProc

In 2007, Geeraerts et al. (a subset of the authors of [57], the paper showing the bug in MCT), designed an algorithm computing the Clover. In contrary to most of other algorithms preforming this task, they do not build a tree (like the one in K-M) but handle sets of pairs of markings. The relation between these pairs is that the second marking can be reached from the first one. This relation is sufficient to find and apply acceleration, like the ones from K-M. In order to exploit monotonicity property of Petri nets, they define a new type of order on these pairs. This order allows them to maintain sets of maximal pairs during the execution.

The authors provide a way to further optimize their algorithm in a form of an oracle. This oracle provides pairs of marking such that the second marking $\mathbf{m}$ is in the Cover and for which its downward closure is closed to transitions i.e.,

$$\downarrow\{\mathbf{m}' \mid \forall t \in T \text{ s.t. } \mathbf{m} \xrightarrow{t} \mathbf{m}'\} \subseteq \downarrow\mathbf{m}.$$

They give an example of such an oracle, which provides them with a good performance boost.

An implementation of `CovProc` with and without this oracle was written in python and compared to `K-M`. They showed that they are much faster than `K-M`. On unbounded Petri nets, the version with the oracle provides a boost to the performance (compared to `CovProc` without the oracle), with speed-ups of up too $100\times$ faster.

MP

In 2011 Reynier et al. of [135] set out to fix the bug in `MCT` by designing a new algorithm *Monotone-Pruning* (`MP`). Their approach for fixing the bug, was to instead of deleting vertices, as done by `MCT`, "disabling" them. A disabled vertex cannot have new children, or be used to discover new accelerations. A disabled vertex is only used to insure completeness.

The algorithm `MP` was implemented in Python and was compared to `K-M` and `CovProc` [135]. They showed that on a set of popular benchmarks, `MP` was $10-30\times$ faster than `CovProc` and on benchmarks that didn't time out for the `K-M` it was $100-300\times$ faster.

Recently [134] Reynier et al. revisited their algorithm. They point out that the proof for completeness provided in [135] is very complicated. Hence, in [134] they devise a new proof for its correctness. This proof 1. is much clearer, and 2. it allows them to give a more generalized version of `MP`. This generalization might be used in some subset of WSTS type of systems.

VH

In 2014 Valmari et al. [152] set out to design an improved version of the `K-M` algorithm, which we will refer to as `VH`. Unlike previous tree algorithms, the authors of [152] did not try to fix the bug with was caused by pruning subtrees. Instead, they focused on making some useful heuristics, smart data structures, etc. Here are a few of these optimizations:

- Written in C.
- **Vertex blocking:** Given a newly discovered vertex with a marking $\mathbf{m}$. If there already exists a marking larger or equal to it, `VH` would not explore its children.
- **Hash table:** In a forward exploration of a Petri net's coverability tree, there might be multiple vertices with the same marking. Not only can it happen, it does happen quite regular because of the nature of transitions in Petri net.

Therefore, instead of checking whether which is bigger or equal to it, they first check for equality using a Hash table.

- **Better acceleration techniques:** Using smart heuristics, they find accelerations faster.

Later on in [129] Piipponen et al. improved `VH`. The authors identified two bottlenecks in `VH`: 1) checking for acceleration, and 2) comparing a given marking to a set of previously discovered marking. Both of these operations are very costly timewise. To solve the first issue, they use a Trajan inspired algorithm. For the second issue, they designed a new data structure inspired by a binary decision diagrams.

Both versions, [152] and [129], were implemented in *C++*. Both tools showed very good performance on the ran benchmarks(where the newer version seem to be 10× faster). Both of the versions were able to compute Clovers to benchmarks which were previously timed out on other tools. However, one have to note that the benchmarks ran in [152] and [129] were only run on their tools.

4.2 Specification and illustration

As discussed in the introduction, to compute the clover of a Petri net, most algorithms build coverability trees (or graphs), which are variants of the Karp and Miller tree with the aim of reducing the peak memory during the execution. The seminal algorithm [56] is characterized by a main difference with the KMT construction: when finding that the marking associated with the current vertex strictly covers the marking of another vertex, it deletes the subtree issued from this vertex, and when the current vertex belonged to the removed subtree it substitutes it to the root of the deleted subtree. This operation drastically reduces the peak memory but as shown in [57] entails incompleteness of the algorithm.

Like the previous algorithms that ensure completeness with deletions, our algorithm also needs additional memory. However, unlike the other algorithms, it memorizes accelerations instead of ω-markings. This approach has two advantages. First, we are able to exhibit a theoretical upper bound on the additional memory which is doubly exponential, while the other algorithms do not have such a bound. Furthermore, accelerations are reused in the construction and thus may even shorten the execution time and peak space w.r.t. the algorithm in [56].

Before we delve into a high-level description of this algorithm, let us present some of the variables, functions, and definitions used by the algorithm. Algorithm 5, denoted from now on as `MinCov` takes as an input a marked net $(\mathcal{N}, \mathbf{m}_{ini})$ and constructs a directed labeled tree $Tr = (V, E, \lambda, \delta)$, and a set Acc of ω-transitions (which by Lemma 4.3.4 are accelerations). Each $v \in V$ is labeled by an ω-marking,

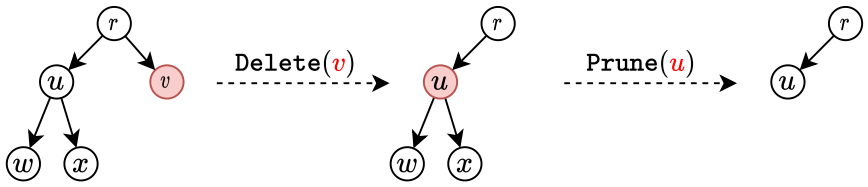


Figure 4.1: Prune and Delete.

$\lambda(v) \in \mathbb{N}_\omega^P$. Since Tr is a directed tree, every vertex $v \in V$, has a predecessor (except the root r) denoted by $prd(v)$ and a set of descendants denoted by $Des(v)$. By convention, $prd(r) = r$. Each edge $e \in E$ is labeled by a firing sequence $\delta(e) \in T_o \cdot \mathsf{Acc}^*$, consisting of an ordinary transition followed by a sequence of accelerations (which by Lemma 4.3.3 fulfills $\lambda(prd(v)) \xrightarrow{\delta(prd(v),v)} \lambda(v)$). In addition, again by Lemma 4.3.3, $\mathbf{m}_0 \xrightarrow{\delta(r,r)} \lambda(r)$. Let $\gamma = e_1 e_2 \dots e_k \in E^*$ be a path in the tree, we denote by $\delta(\gamma) := \delta(e_1)\delta(e_2)\dots\delta(e_k) \in (T \cup \mathsf{Acc})^*$. The subset $\mathsf{Front} \subseteq V$ is the set of vertices 'to be processed'.

`MinCov` may call function `Delete`(v) that removes from V a leaf v of Tr and function `Prune`(v) that removes from V all descendants of $v \in V$ except v itself as illustrated in Figure 4.1.

First `MinCov` does some initialization, and sets the tree Tr to be a single vertex r with marking $\lambda(r) = \mathbf{m}_{ini}$ and $\mathsf{Front} = \{r\}$. Afterwards, the main loop builds the tree, where each iteration consists in processing some vertex in Front as follows.

`MinCov` picks a vertex $u \in \mathsf{Front}$ (line 3). From $\lambda(u)$, `MinCov` fires a sequence $\sigma \in Acc^*$ reaching some $\mathbf{m}_u$ that maximizes the number of ω produced, i.e., $|\{p \in P \mid \lambda(u)(p) \neq \omega \wedge \mathbf{m}_u(p) = \omega\}|$. Thus, in σ, no acceleration occurs twice, and its length is bounded by $|P|$. Then `MinCov` updates $\lambda(u)$ with $\mathbf{m}_u$ (line 5) and the label of the edge incoming to u by concatenating σ. Afterwards, it performs one of the following actions according to the marking $\lambda(u)$:

- **Cleaning** (line 7): If there exists $u' \in V \setminus \mathsf{Front}$ with $\lambda(u') \geq \lambda(u)$. The vertex u is redundant and `MinCov` calls `Delete`(u)
- **Accelerating** (lines 8-16): If there exists u', an ancestor of u with $\lambda(u') < \lambda(u)$. then an acceleration can be computed. The acceleration $\mathbf{a}$ is deduced from the firing sequence labeling the path from u' to u. `MinCov` inserts $\mathbf{a}$ into Acc, calls `Prune`(u') and pushes back u' in Front.
- **Exploring** (lines 18 - 25): Otherwise `MinCov` calls `Prune`(u') followed by `Delete`(u') for all $u' \in V$ with $\lambda(u') < \lambda(u)$ since they are redundant. Afterwards, it removes u from Front and for all fireable transition $t \in T$ from $\lambda(u)$, it creates a new child for u in MCT and inserts it into Front.

4.2.1 A run of the algorithm

Example 4.2.1. Let us describe some iterations of $\texttt{MinCov}(\mathcal{N}, \mathbf{m}_{init})$, where $\mathcal{N}$ is the Petri net in Figure 3.2 and $\mathbf{m}_{init} = \mathbf{p}_{init}$. For each iteration of the main loop below, we present a figure showing Tr before and after the iteration. In the figures, we color in *red* the currently processed vertex and in *blue* the other vertices in Front.

Iteration 2 (Figure 4.2): MinCov picks v_5 from Front and processes it. Since

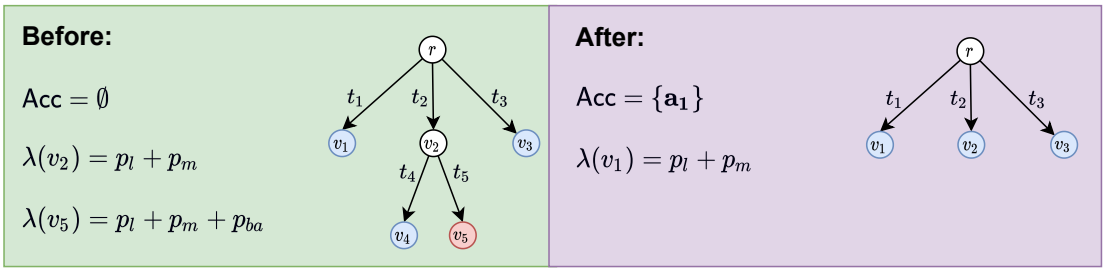


Figure 4.2: Iteration 2 — Finding a new acceleration.

Acc $= \emptyset$, there is no possible acceleration firing (lines 4-5). Next, MinCov discovers an ancestor v_2 of v_5 with a smaller marking. Therefore, it performs an acceleration phase (lines 8-16). MinCov builds a new acceleration $\mathbf{a}_1$ (according to labels of the edges between v_5 and v_2), with $\mathbf{Pre}(\mathbf{a}_1) = \mathbf{p}_l$ and $\mathbf{C}(\mathbf{a}_1) = \omega \cdot \mathbf{p}_{ba}$ and inserts it in Acc. Afterwards, it performs $\texttt{Prune}(v_2)$, and pushes v_2 back to Front.

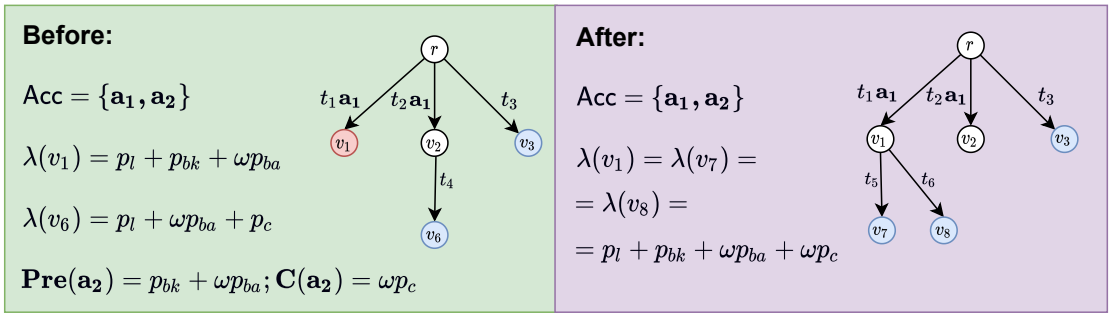


Figure 4.3: Iteration 7 — Side deletion and exploration.

Iteration 7 (Figure 4.3): MinCov picks v_1 from Front and processes it. Since $\mathbf{a}_2$ is fireable from $\lambda(v_1)$, $\lambda(v_1)$ becomes $\lambda(v_1) + \mathbf{C}(\mathbf{a}_2) = \mathbf{p}_l + \mathbf{p}_{bk} + \omega\mathbf{p}_{ba} + \omega\mathbf{p}_c$ (lines 4-5). Since there is no marking greater than $\lambda(v_1)$ in $\lambda(V \setminus \text{Front})$ and no ancestor with a smaller marking, MinCov goes to the exploration phase (lines 18-25).

Looking at all vertices, `MinCov` deletes (the subtree rooted in) v_6 since $\lambda(v_6) < \lambda(v_1)$. Afterwards, it creates two children of v_1 (v_7 and v_8) since t_5 and t_6 are fireable from $\lambda(v_1)$.

Iteration 11 (Figure 4.4): `MinCov` picks v_9 from Front and processes it. Neither $\mathbf{a}_1$, nor $\mathbf{a}_2$ are fireable from $\lambda(v_9)$, so it is unchanged by lines 4-5. Then since $\lambda(v_9) \leq \lambda(v_1)$, `MinCov` performs the cleaning phase by calling `Delete`(v_9).

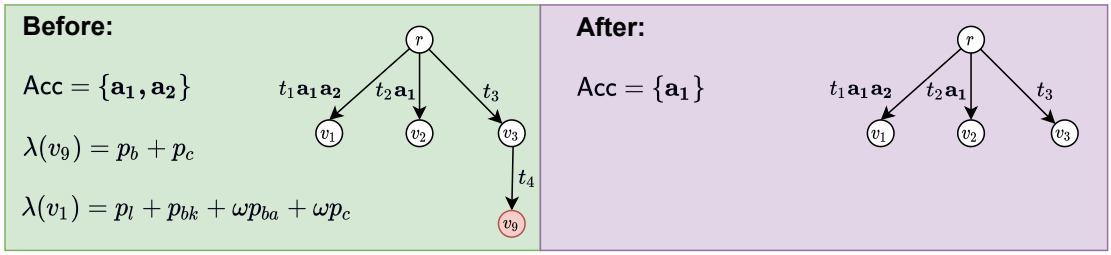


Figure 4.4: Iteration 11 — Clean up.

Iteration 11 is the last iteration of `MinCov` since Front $= \emptyset$. The final tree is the one presented in Figure 4.5. The set of ω-markings decorating its vertices is:

$$\lambda(V) = \{p_i, p_l + p_{bk} + \omega p_{ba} + \omega p_c, p_{bk} + p_m, p_l + p_m + \omega p_{ba}\}$$

which is exactly the Clover set of the marked Petri net in Figure 3.2

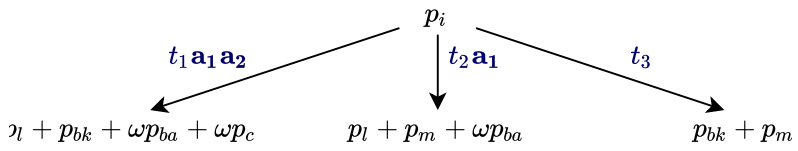


Figure 4.5: The tree generated by `MinCov` in Example 4.2.1

4.3 `MinCov` Correctness Proof

We now establish the correctness of Algorithm 5 by proving the following properties (where for all $W \subseteq V$, $\lambda(W)$ denotes $\bigcup_{v \in W} \lambda(v)$):

- its termination;
- the incomparability of ω-markings associated with vertices in V: $\lambda(V)$ is an antichain;

Algorithm 5: Computing the minimal coverability set

`MinCov`$(\mathcal{N}, \mathbf{m}_{ini})$
Input: A marked Petri net $(\mathcal{N}, \mathbf{m}_{ini})$
Data: V set of vertices; $E \subseteq V \times V$; $\mathsf{Front} \subseteq V$; $\lambda : V \to \mathbb{N}^p_\omega$; $\delta : E \to T_o\mathsf{Acc}^*$; $Tr = (V, E, \lambda, \delta)$ a labeled tree; Acc a set of ω-transitions;
Output: A labeled tree $Tr = (V, E, \lambda, \delta)$

1 $V \leftarrow \{r\}$; $E \leftarrow \emptyset$; $\mathsf{Front} \leftarrow \{r\}$; $\lambda(r) \leftarrow \mathbf{m}_{ini}$; $\mathsf{Acc} \leftarrow \emptyset$; $\delta(r, r) \leftarrow \varepsilon$
2 **while** $Front \neq \emptyset$ **do**
3 `Select` $u \in \mathsf{Front}$
4 Let $\sigma \in \mathsf{Acc}^*$ a maximal fireable sequence of accelerations from $\lambda(u)$
  // Maximal w.r.t. the number of ω's produced
5 $\lambda(u) \leftarrow \lambda(u) + \mathbf{C}(\sigma)$
6 $\delta((prd(u), u)) \leftarrow \delta((prd(u), u)) \cdot \sigma$
7 **if** $\exists u' \in V \setminus \mathsf{Front}$ *s.t.* $\lambda(u') \geq \lambda(u)$ **then** `Delete`(u) // $\lambda(u)$ is covered
8 **else if** $\exists u' \in \mathsf{Anc}(V)$ *s.t.* $\lambda(u) > \lambda(u')$ **then**
    // An acceleration was found between u and one of u's ancestors
9 Let $\gamma \in E^*$ the path from u' to u in Tr
10 $\mathbf{a} \leftarrow$ `NewAcceleration`()
11 **foreach** $p \in P$ **do**
12 **if** $\mathbf{C}(p, \delta(\gamma)) < 0$ **then** $\mathbf{Pre}(p, \mathbf{a}) \leftarrow \omega$; $\mathbf{C}(p, \mathbf{a}) \leftarrow \omega$
13 **if** $\mathbf{C}(p, \delta(\gamma)) = 0$ **then** $\mathbf{Pre}(p, \mathbf{a}) \leftarrow \mathbf{Pre}(p, \delta(\gamma))$; $\mathbf{C}(p, \mathbf{a}) \leftarrow 0$
14 **if** $\mathbf{C}(p, \delta(\gamma)) > 0$ **then** $\mathbf{Pre}(p, \mathbf{a}) \leftarrow \mathbf{Pre}(p, \delta(\gamma))$; $\mathbf{C}(p, \mathbf{a}) \leftarrow \omega$
15 **end**
16 $\mathbf{a} \leftarrow trunc(\mathbf{a})$; $\mathsf{Acc} \leftarrow \mathsf{Acc} \cup \{\mathbf{a}\}$; `Prune`$(u')$; $\mathsf{Front} = \mathsf{Front} \cup \{u'\}$;
17 **else**
18 **for** $u' \in V$ **do**
    // Remove vertices labeled by markings covered by $\lambda(u)$
19 **if** $\lambda(u') < \lambda(u)$ **then** `Prune`(u'); `Delete`(u')
20 **end**
21 $\mathsf{Front} \leftarrow \mathsf{Front} \setminus \{u\}$
22 **foreach** $t \in T \wedge \lambda(u) \geq \mathbf{Pre}(t)$ **do**
    // Add the children of u
23 $u' \leftarrow$ `NewNode`(); $V \leftarrow V \cup \{u'\}$; $\mathsf{Front} \leftarrow \mathsf{Front} \cup \{u'\})$; $E \leftarrow E \cup \{(u, u')\}$
24 $\lambda(u') \leftarrow \lambda(u) + \mathbf{C}(t)$; $\delta((u, u')) \leftarrow t$
25 **end**
26 **end**
27 **end**
28 **return** Tr

- its consistency: $[\![\lambda(V)]\!] \subseteq Cover(\mathcal{N}, \mathbf{m}_0)$;
- its completeness: $Cover(\mathcal{N}, \mathbf{m}_0) \subseteq [\![\lambda(V)]\!]$.

Similarly to the proof of Proposition 3.2.2 we get termination by using the well order of $\mathbb{N}^P_\omega$ and Koenig Lemma.

Proposition 4.3.1. *`MinCov` terminates.*

Proof. Consider the following variation of the algorithm.

Instead of deleting the current vertex when its marking is smaller or equal than the marking of a vertex, one marks it as 'cut' and extract it from Front.

Instead of cutting a subtree when the marking of the current vertex v is greater than the marking of a vertex which is not an ancestor of v, one marks them as 'cut' and extract from Front those who are inside.

Instead of cutting a subtree when the marking of the current vertex v is greater than the marking of a vertex which is an ancestor of v, say v^*, one marks those on the path from v^* to v (except v) as 'accelerated', one marks the other vertices of the subtree as 'cut' and inserts v again in Front with the marking of v^*. All the markings of the subtree in Front are extracted from it.

All the vertices marked as 'cut' or 'accelerated' are ignored for comparisons and discovering accelerations. This alternative algorithm behaves as the original one except that the size of the tree never decreases and so if the algorithm does not terminate, the tree is infinite. Since this tree is finitely branching, due to Koenig Lemma it contains an infinite path. On this infinite path, no vertex can be marked as 'cut' since it would belong to a finite subtree. Observe that the marking labelling the vertex following an accelerated subpath has at least one more ω than the marking of the first vertex of this subpath. So there is an infinite subpath with unmarked vertices in V. But $\mathbb{N}^P_\omega$ is well-ordered, so there should be two vertices v and v', where v' is a descendant of v with $\lambda(v') \geq \lambda(v)$, which contradicts the behavior of the algorithm. □

Since we are going to use recurrence on the number of iterations of the main loop of Algorithm 5, we introduce the following notations: $Tr_n = (V_n, E_n, \lambda_n, \delta_n)$, Front_n, and Acc_n are the values of variables Tr, Front, and Acc at line 2 when n iterations have been executed.

Proposition 4.3.2. *For all $n \in \mathbb{N}$, $\lambda(V_n \setminus \mathsf{Front}_n)$ is an antichain. Thus, on termination, $\lambda(V)$ is an antichain.*

Proof. Let us introduce $V' := V \setminus \mathsf{Front}$ and $V'_n := V_n \setminus \mathsf{Front}_n$. We are going to prove by induction on the number n of iterations of the while-loop that V'_n is an

antichain. MinCov initializes variables V and Front at line 1. So $V_0 = \{r\}$ and $\mathsf{Front}_0 = \{r\}$, therefore, $V'_0 = V_0 \setminus \mathsf{Front}_0 = \emptyset$ is an antichain.
Assume that $V'_n = V_n \setminus \mathsf{Front}_n$ is an antichain. Modifying V'_n can be done by *adding* or *removing* vertices from V_n and *removing* vertices from Front_n while keeping them in V_n. The actions that MinCov may perform in order to modify the sets V and Front are: Delete (lines 7 and 19), Prune (lines 16 and 19), adding vertices to V (line 23), adding vertices to Front (lines 16 and 23), and removing vertices from Front (line 21).
• Both Delete and Prune do not add new vertices to V'. Thus, the antichain feature is preserved.
• MinCov may add vertices to V only at line 23 where it simultaneously adds them to Front and therefore does not add new vertices to V'. Thus, the antichain feature is preserved.
• Adding vertices to Front may only remove vertices from V'_n. Thus, the antichain feature is preserved.
• MinCov can only add a vertex to V' when it removes it from Front while keeping it in V. This is done only at line 21. There the only vertex MinCov may remove (line 21) is the working vertex u. However, if (in the iteration) MinCov reaches line 21 then it did not reach line 7 hence, (1) all markings of $\lambda(V'_n) \subseteq \lambda(V_n)$ are either smaller or incomparable to $\lambda_{n+1}(u)$. Moreover, MinCov has also reached line 18-20, where (2) it performs Delete on all vertices $u' \in V'_n \subseteq V_n$ with $\lambda_n(u') < \lambda_{n+1}(u)$. Let us denote by $V''_n \subseteq V'_n$ the set V' at the end of line 20. Due to (1) and (2), marking $\lambda_{n+1}(u)$ is incomparable to any marking in $\lambda_{n+1}(V''_n)$. Since $V''_n \subseteq V'_n$, $\lambda_{n+1}(V''_n)$ is an antichain. Combining this fact with the incomparability between $\lambda_{n+1}(u)$ and any marking in $\lambda_{n+1}(V''_n)$, we conclude that the set $\lambda_{n+1}(V'_{n+1}) = \lambda_{n+1}(V''_n) \cup \{\lambda_{n+1}(u)\}$ is an antichain. □

In order to establish consistency, we prove that the labelling of vertices and edges is compatible with the firing rule and that *Acc* is a set of accelerations.

Lemma 4.3.3. *For all* $n \in \mathbb{N}$, *for all* $u \in V_n \setminus \{r\}$, $\lambda_n(prd(u)) \xrightarrow{\delta(prd(u),u)} \lambda_n(u)$ *and* $\mathbf{m}_{ini} \xrightarrow{\delta(r,r)} \lambda_n(r)$.

Proof. Let us prove by induction on the number n of iterations of the main loop that, for all $v \in V_n$, the assertions of the lemma hold. Initially, $V_0 = \{r\}$ and $\lambda_0(r) = \mathbf{m}_{ini}$. Since $\mathbf{m}_{ini} \xrightarrow{\varepsilon} \mathbf{m}_{ini} = \lambda_0(r)$ the base case is established.
Assume that the assertions hold for Tr_n. Observe that MinCov may change the labeling function λ and/or add new vertices in exactly two places: at lines 4-6 and at lines 22-25. Therefore, in order to prove the assertion, we show that after each group of lines it still holds.
• After lines 4-6: MinCov computes (1) a maximal fireable sequence $\sigma \in \mathsf{Acc}^*_n$ from $\lambda_n(u)$ (line 4), and updates u's marking to $\mathbf{m}_u = \lambda_n(u) + \mathbf{C}(\sigma)$ (line 5).

Since the assertions hold for Tr_n, (2) if $u \neq r$, $\lambda_n(prd(u)) \xrightarrow{\delta(prd(u),u)} \lambda_n(u)$ else $\mathbf{m}_{ini} \xrightarrow{\delta(r,r)} \lambda_n(r)$. By concatenation, we get $\lambda_n(prd(u)) \xrightarrow{\delta(prd(u),u)\sigma} \mathbf{m}_u$ if $u \neq r$ and otherwise, $\mathbf{m}_{ini} \xrightarrow{\delta(r,r)\sigma} \mathbf{m}_u$ which establishes that the assertions hold after line 6.
• After lines 22-25: The vertices for which λ is updated at these lines are the children of u that are added to the tree. For every fireable transition $t \in T$ from $\lambda(u)$, `MinCov` creates a child v_t for u (lines 22-23). The marking of any child v_t is set to $\mathbf{m}_{n+1}(v) := \mathbf{m}_{n+1}(u) + \mathbf{C}(t)$ (line 24). Therefore, since $\lambda_{n+1}(u) \xrightarrow{t} \lambda_{n+1}(v_t)$, the assertions hold. □

Next, we show that the set Acc always consists of accelerations (which are abstractions).

Lemma 4.3.4. *At any execution point of `MinCov`, Acc is a set of accelerations.*

Proof. At most, one acceleration is added per iteration. Let us prove by induction on the number n of iterations of the main loop that Acc_n is a set of accelerations. Since $\mathsf{Acc}_0 = \emptyset$, the base case is straightforward.
Assume that Acc_n is a set of accelerations and consider Acc_{n+1}. In an iteration, `MinCov` may add an ω-transition $\mathbf{a}$ to Acc. Due to the inductive hypothesis, $\delta(\gamma)$ is a sequence of abstractions where γ is defined at line 9. Consider b, the ω-transition defined by $\mathbf{Pre}(b) = \mathbf{Pre}(\delta(\gamma))$ and $\mathbf{C}(b) = \mathbf{C}(\delta(\gamma))$. Due to Proposition 3.1.4, b is an abstraction. Due to Proposition 3.1.6, the loop of lines 11-15 transforms b into an acceleration $\mathbf{a}$. Due to Proposition 3.3.6, after truncation at line 16, $\mathbf{a}$ is still an acceleration. □

Using the previous two results and Proposition 3.1.4 on concatenation of abstractions, we get consistency:

Proposition 4.3.5. $[\![\lambda(V)]\!] \subseteq Cover(\mathcal{N}, \mathbf{m}_0)$.

Proof. Let $v \in V$. Consider the path $u_0, \ldots, u_k$ of MCT from the root $r = u_0$ to $u_k = v$. Let $\sigma \in (T \cup \mathsf{Acc})^*$ denote $\delta(prd(u_0), u_0) \cdots \delta(prd(u_k), u_k)$. Due to Lemma 4.3.3, $m_0 \xrightarrow{\sigma} \lambda(v)$. Due to Lemma 4.3.4, σ is a sequence of abstractions. Due to Proposition 3.1.4, the ω-transition $\mathbf{a}$ defined by $\mathbf{Pre}(\mathbf{a}) = \mathbf{Pre}(\sigma)$ and $\mathbf{C}(\mathbf{a}) = \mathbf{C}(\sigma)$ is an abstraction. Due to Proposition 3.1.3, $[\![\lambda(v)]\!] \subseteq Cover(\mathcal{N}, \mathbf{m}_0)$. □

The following definitions are related to an arbitrary execution point of `MinCov` and are introduced to establish its completeness. Recall that, while proving the correctness of the Karp and Miller algorithm, we introduced sequences we called

"exploring sequence"(definition 3.2.6). We are again in need for this type of sequences, but since MinCov can fire accelerations directly, we need to extend the definition which would including them:

Definition 4.3.6. Let $\sigma = \sigma_0 t_1 \sigma_1 \dots t_k \sigma_k$ with for all i, $t_i \in T$ and $\sigma_i \in \mathsf{Acc}^*$. Then the firing sequence $\mathbf{m} \xrightarrow{\sigma} \mathbf{m}'$ is an *exploring sequence* if:

- There exists $v \in \mathsf{Front}$ with $\lambda(v) = \mathbf{m}$
- For all $0 \leq i \leq k$, there does not exist $v' \in V \setminus \mathsf{Front}$ with $\mathbf{m} + \mathbf{C}(\sigma_0 t_1 \sigma_1 \dots t_i \sigma_i) \leq \lambda(v')$.

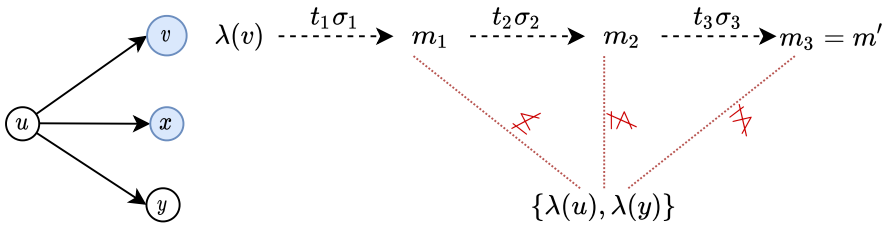


Figure 4.6: Iteration 2 — Finding a new acceleration.

Similarly to Definition 3.2.7 we define quasi-covered. This time using the extended version of exploration sequences:

Definition 4.3.7. Let $\widehat{\mathbf{m}}$ be a marking. Then $\widehat{\mathbf{m}}$ is *quasi-covered* if:

- either there exists $v \in V \setminus \mathsf{Front}$ with $\lambda(v) \geq \widehat{\mathbf{m}}$;
- or there exists an exploring sequence $\mathbf{m} \xrightarrow{\sigma} \mathbf{m}' \geq \widehat{\mathbf{m}}$.

In order to prove completeness of the algorithm, we want to prove that at the beginning of every iteration, any $\mathbf{m} \in \mathit{Cover}(\mathcal{N}, \mathbf{m}_0)$ is quasi-covered. To establish this assertion, we introduce several lemmas showing that this assertion is preserved by some actions of the algorithm with some prerequisites. More precisely, Lemma 4.3.8 corresponds to the deletion of the current vertex, Lemma 4.3.9 to the discovery of an acceleration, Lemma 4.3.10 to the deletion of a subtree whose marking of the root is smaller than the marking of the current vertex and Lemma 4.3.11 to the creation of the children of the current vertex.

Lemma 4.3.8. *Let Tr,* Front *and* Acc *be the values of corresponding variables at some execution point of* MinCov *and $u \in V$ be a leaf in Tr such that the following items hold:*

1. *All* $\mathbf{m} \in Cover(\mathcal{N}, \mathbf{m}_0)$ *are quasi-covered;*
2. $\lambda(V \setminus \mathsf{Front})$ *is an antichain;*
3. *For all* $\mathbf{a} \in \mathsf{Acc}$ *fireable from* $\lambda(u)$, $\lambda(u) = \lambda(u) + \mathbf{C}(\mathbf{a})$*;*
4. *There exists* $v \in V \setminus \{u\}$ *such that* $\lambda(v) \geq \lambda(u)$.

Then all $\mathbf{m} \in Cover(\mathcal{N}, \mathbf{m}_0)$ *are quasi-covered after performing* `Delete`(u).

Proof. Let u be a leaf of Tr. Assume the assertions of the lemma are satisfied. Denote by $Tr' = (V', E', \lambda', \delta')$ and Front', the value of variables Tr and Front after performing `Delete`(u) on Tr.
Let $\mathbf{m}^* \in Cov(\mathcal{N}, \mathbf{m}_{ini})$. Thus $\mathbf{m}^*$ is quasi-covered. By definition, $V' = V \setminus \{u\}$ and $\mathsf{Front}' = \mathsf{Front} \setminus \{u\}$. Moreover, for all $v' \in V'$ we have $\lambda'(v') = \lambda(v')$. We split the proof in two cases:
• There exists $w \in V \setminus \mathsf{Front}$ with $\lambda(w) \geq \mathbf{m}^*$

- If $w \neq u$ then $w \in V' \setminus \mathsf{Front}'$. Since $\lambda'(w) = \lambda(w)$, $\mathbf{m}^*$ is still quasi-covered.
- If $w = u$ then there exists v such that $\lambda'(v) = \lambda(v) \geq \lambda(u)$. Therefore, if $v \notin \mathsf{Front}$ then m^* is still quasi-covered. Otherwise, since $\lambda(V \setminus \mathsf{Front})$ is an antichain, for all $v' \notin \mathsf{Front}$ $\lambda(v') \not\geq \lambda(u)$ implying $\lambda(v') \not\geq \lambda(v)$. Therefore, the sequence $\lambda(v) \xrightarrow{\varepsilon} \lambda(v) \geq \mathbf{m}^*$, is an exploring sequence.

• There is an exploring sequence $\mathbf{m} \xrightarrow{\sigma} \mathbf{m}' \geq \mathbf{m}^*$ with some w such that $\mathbf{m} = \lambda(w)$. Let $\sigma = \sigma_0 t_1 \sigma_1 \cdots t_k \sigma_k$, where $t_i \in T$ and $\sigma_i \in \mathsf{Acc}^*$.

- Assume $w \neq u$. Since (1) for all $0 \leq i \leq k$, there does not exist $v' \in V \setminus \mathsf{Front}$ such that $\lambda(w) + \mathbf{C}(\sigma_0 t_1 \sigma_1 \dots t_i \sigma_i) \leq \lambda(v')$, (2) $V' \setminus \mathsf{Front}' \subseteq V \setminus \mathsf{Front}$, and (3) $\lambda'(w) = \lambda(w)$, $\mathbf{m} \xrightarrow{\sigma} \mathbf{m}'$ is still an exploring sequence.
- Assume $w = u$. Since $\lambda(v) \geq \lambda(u) = \lambda(u) + \mathbf{C}(\sigma_0)$ then $v \in \mathsf{Front}$: otherwise it would contradict the definition of an exploring sequence. Since $\lambda[u](v) = \lambda(v) \geq \lambda(u)$, σ is also fireable from $\lambda[u](v)$. Since $V' \setminus \mathsf{Front}' \subseteq V \setminus \mathsf{Front}$, there does not exist $v' \in V' \setminus \mathsf{Front}'$ such that $\lambda[u](v) + \mathbf{C}(\sigma_0 t_1 \sigma_1 \dots t_i \sigma_i) \leq \lambda[u](v')$ for some i. Therefore, $\lambda'(v) \xrightarrow{\sigma} \lambda'(v) + \mathbf{C}(\sigma) \geq \mathbf{m}^*$ is an exploring sequence.

□

Next we show that, any $\mathbf{m} \in Cover(\mathcal{N}, \mathbf{m}_0)$ is still quasi-covered after finding an acceleration:

Lemma 4.3.9. *Let* Tr, Front *and* Acc *be the values of corresponding variables at some execution point of* `MinCov`*. and* $u \in V$ *such that the following items hold:*

1. *All* $\mathbf{m} \in Cover(\mathcal{N}, \mathbf{m}_0)$ *are quasi-covered;*
2. $\lambda(V \setminus \mathsf{Front})$ *is an antichain;*
3. *For all* $v \in V \setminus \{r\}$, $\lambda(prd(v)) \xrightarrow{\delta(prd(v),v)} \lambda(v)$.

Then all $\mathbf{m} \in Cover(\mathcal{N}, \mathbf{m}_0)$ *are quasi-covered after performing* $\texttt{Prune}(u)$ *and then adding* u *to* Front.

Proof. Let $u \in V$. Denote by $Tr' = (V', E', \lambda', \delta')$, and Front' the value Tr after performing $\texttt{Prune}(u)$ on Tr and adding u to Front. Assume that the assertions of the Lemma hold. If $u \in \mathsf{Front}$ then $Tr' = Tr$ and we are done. So let us assume that $u \notin \mathsf{Front}$. Due to the definition of $\texttt{Prune}$, $V' \setminus \mathsf{Front}' = V \setminus (\mathsf{Front} \cup Des(u))$. Moreover, for all $v \in V'$, $\lambda'(v) = \lambda(v)$.
Let $\mathbf{m}^* \in Cov(\mathcal{N}, \mathbf{m}_{ini})$. Thus $\mathbf{m}^*$ is quasi-covered. We split the proof in two cases:
• There exists $w \in V \setminus \mathsf{Front}$ with $\lambda(w) \geq \mathbf{m}^*$.

- Assume that w is not a descendant of u. Since w is not a descendant of u, $w \in V' \setminus \mathsf{Front}'$. Since $\lambda'(w) = \lambda(w) \geq \mathbf{m}^*$, $\mathbf{m}^*$ is quasi-covered.
- Assume that w is a descendant of u, by a path in Tr $u = w_0 \xrightarrow{\delta(w_0,w_1)} w_1 \cdots w_{k-1} \xrightarrow{\delta(w_{k-1},w_k)} w_k = w$. Since $\lambda(V \setminus \mathsf{Front})$ is an antichain any $\lambda(w_i)$ is incomparable with any $\lambda(w')$ in $\lambda(V \setminus \mathsf{Front})$.
  Since $V' \setminus \mathsf{Front}' \subseteq V \setminus \mathsf{Front}$, the sequence
  $\lambda(u) = \lambda(w_0) \xrightarrow{\delta(w_0,w_1)} \lambda(w_1) \cdots \lambda(w_{k-1}) \xrightarrow{\delta(w_{k-1},w_k)} \lambda(w_k) = \lambda(w) \geq \mathbf{m}^*$
  is an exploring sequence.

• There is an exploring sequence $\mathbf{m} \xrightarrow{\sigma} \mathbf{m}' \geq \mathbf{m}^*$ with some w such that $\mathbf{m} = \lambda(w)$. Let $\sigma = \sigma_0 t_1 \sigma_1 \cdots t_k \sigma_k$, where $t_i \in T$ and $\sigma_i \in \mathsf{Acc}^*$.

- Assume that w is not a descendant of u. Thus $w \in \mathsf{Front}'$. Since $V' \setminus \mathsf{Front}' \subseteq V \setminus \mathsf{Front}$, and for all v', $\lambda'(v') = \lambda(v')$, $\mathbf{m} \xrightarrow{\sigma} \mathbf{m}' \geq \mathbf{m}^*$ is still an exploring sequence.
- Assume that w is a descendant of u by a path in Tr $u = w_0 \xrightarrow{\delta(w_0,w_1)} w_1 \cdots w_{k-1} \xrightarrow{\delta(w_{k-1},w_k)} w_k = w$. Since $\lambda(V \setminus \mathsf{Front})$ is an antichain any $\lambda(w_i)$ with $i < k$ is incomparable with any $\lambda(w')$ in $\lambda(V \setminus \mathsf{Front})$. Consider the sequence $\lambda(u) = \lambda(w_0) \xrightarrow{\delta(w_0,w_1)} \lambda(w_1) \cdots \lambda(w_{k-1}) \xrightarrow{\delta(w_{k-1},w_k)} \lambda(w_k) \xrightarrow{\sigma} \mathbf{m}' \geq \mathbf{m}^*$. Since $V' \setminus \mathsf{Front}' \subseteq V \setminus \mathsf{Front}$, and for all v', $\lambda'(v') = \lambda(v')$, this sequence is an exploring sequence.

□

Next we show that, any $\mathbf{m} \in Cover(\mathcal{N}, \mathbf{m}_0)$ is still quasi-covered after pruning a subtree smaller than current marking:

Lemma 4.3.10. *Let* Tr, Front *and* Acc *be the values of corresponding variables at some execution point of* `MinCov`, $u \in \mathsf{Front}$ *and* $u' \in V$ *such that the following items hold:*

1. *All* $\mathbf{m} \in Cover(\mathcal{N}, \mathbf{m}_0)$ *are quasi-covered;*
2. $\lambda(V \setminus \mathsf{Front})$ *is an antichain;*
3. *For all* $v \in V \setminus \{r\}$, $\lambda(prd(v)) \xrightarrow{\delta(prd(v),v)} \lambda(v)$*;*
4. $\lambda(u') < \lambda(u)$ *and* u *is not a descendant of* u'*.*

Then after performing `Prune`(u');`Delete`(u')*,*

1. *All* $\mathbf{m} \in Cover(\mathcal{N}, \mathbf{m}_0)$ *are quasi-covered;*
2. $\lambda(V \setminus \mathsf{Front})$ *is an antichain;*
3. *For all* $v \in V \setminus \{r\}$, $\lambda(prd(v)) \xrightarrow{\delta(prd(v),v)} \lambda(v)$*.*

Proof. The second and third assertions of the conclusion are still satisfied, since we do no add vertices and edges. So let us establish that all $\mathbf{m} \in Cover(\mathcal{N}, \mathbf{m}_0)$ are still quasi-covered. Let $\mathbf{m}^* \in Cov(\mathcal{N}, \mathbf{m}_{ini})$.
• There exists $w \in V \setminus \mathsf{Front}$ with $\lambda(w) \geq \mathbf{m}^*$.

- Assume that w is not a descendant of u'. Since w is not a descendant of u', w is still in $V \setminus \mathsf{Front}$. Since $\lambda(w) \geq \mathbf{m}^*$, $\mathbf{m}^*$ is quasi-covered.
- Assume that w is a descendant of u', by a path in Tr $u' = w_0 \xrightarrow{\delta(w_0,w_1)} w_1 \cdots w_{k-1} \xrightarrow{\delta(w_{k-1},w_k)} w_k = w$. Since $\lambda(V \setminus \mathsf{Front})$ is an antichain any $\lambda(w_i)$ is incomparable with any $\lambda(w')$ in $\lambda(V \setminus \mathsf{Front})$. Since $V \setminus \mathsf{Front}$ does not include new items, the sequence $\lambda(u) \xrightarrow{\delta(w_0,w_1)\cdots\delta(w_{k-1},w_k)} \mathbf{m}'$ is an exploring sequence with $\mathbf{m}' \geq \mathbf{m}^*$.

• There is an exploring sequence $\mathbf{m} \xrightarrow{\sigma} \mathbf{m}' \geq \mathbf{m}^*$ with some w such that $\mathbf{m} = \lambda(w)$. Let $\sigma = \sigma_0 t_1 \sigma_1 \cdots t_k \sigma_k$, where $t_i \in T$ and $\sigma_i \in \mathsf{Acc}^*$.

- Assume that w is not a descendant of u'. Since $V \setminus \mathsf{Front}$ does not include new items, $\mathbf{m} \xrightarrow{\sigma} \mathbf{m}' \geq \mathbf{m}^*$ is still an exploring sequence.
- Assume that w is a descendant of u' by a path in Tr $u' = w_0 \xrightarrow{\delta(w_0,w_1)} w_1 \cdots w_{k-1} \xrightarrow{\delta(w_{k-1},w_k)} w_k = w$. Since $\lambda(V \setminus \mathsf{Front})$ is an antichain any $\lambda(w_i)$ with $i < k$ is incomparable with any $\lambda(w')$ in $\lambda(V \setminus \mathsf{Front})$. Consider the sequence $\lambda(u') < \lambda(u) \xrightarrow{\delta(w_0,w_1)\cdots\delta(w_{k-1},w_k)\sigma} \mathbf{m}' \geq \mathbf{m}^*$. Since $V \setminus \mathsf{Front}$ does not include new items, this sequence is an exploring sequence.

□

Finally, we show that, any $\mathbf{m} \in Cover(\mathcal{N}, \mathbf{m}_0)$ still quasi-covered after exploring and adding the new children of a current node:

Lemma 4.3.11. *Let* Tr*,* Front *and* Acc *be the values of corresponding variables at some execution point of* `MinCov`*. and* $u \in \mathsf{Front}$ *such that the following items hold:*

1. *All* $\mathbf{m} \in Cover(\mathcal{N}, \mathbf{m}_0)$ *are quasi-covered;*
2. $\lambda(V \setminus \mathsf{Front}) \cup \{\lambda(u)\}$ *is an antichain;*
3. *For all* $\mathbf{a} \in \mathsf{Acc}$ *fireable from* $\lambda(u)$, $\lambda(u) = \lambda(u) + \mathbf{C}(\mathbf{a})$.

Then after removing u *from* Front *and for all* $t \in T$ *fireable from* $\lambda(u)$, *adding a child* v_t *to* u *in* Front *with marking of* v_t *defined by* $\lambda_u(v_t) = \lambda(u) + \mathbf{C}(t)$, *all* $\mathbf{m} \in Cover(\mathcal{N}, \mathbf{m}_0)$ *are quasi-covered.*

Proof. Let $u \in \mathsf{Front}$, denote by $Tr' = (V', E', \lambda', \delta')$ and Front' the value of variables Tr and Front after removing u from Front and for all transition $t \in T$ fireable from $\lambda(u)$, adding a child v_t to u in Front' with marking of v_t defined by $\lambda'(v_t) = \lambda(u) + \mathbf{C}(t)$. In words, this corresponds to lines 21-25 of `MinCov`. Assume the assertions of the lemma hold.
Let $\mathbf{m}^* \in Cov(\mathcal{N}, \mathbf{m}_{ini})$. Due to the assertions of the lemma, $\mathbf{m}^*$ is quasi-covered. By definition, $V' \setminus \mathsf{Front}' = (V \setminus \mathsf{Front}) \cup \{u\}$. We split the proof in two cases:
• There exists $w \in V \setminus \mathsf{Front}$ with $\lambda(w) \geq \mathbf{m}^*$. Since only u has been added to $V \setminus \mathsf{Front}$ and $\lambda(u)$ is incomparable to any $\lambda(w)$, $\mathbf{m}$ is still quasi-covered.
• There is an exploring sequence $\mathbf{m} \xrightarrow{\sigma} \mathbf{m}' \geq \mathbf{m}^*$ with some w such that $\mathbf{m} = \lambda(w)$. Let $\sigma = \sigma_0 t_1 \sigma_1 \cdots t_k \sigma_k$, where $t_i \in T$ and $\sigma_i \in \mathsf{Acc}^*$.

- Either there does not exist $i \leq k$ such that $\lambda(w) + \mathbf{C}(\sigma_0 t_1 \sigma_1 \cdots t_i \sigma_i) \leq \lambda(u)$. Observe that this implies $w \neq u$. Otherwise, since from $\lambda(u)$ no fireable acceleration produces some ω, one would obtain $\lambda(u) + \mathbf{C}(\sigma_0) = \lambda(u)$. Since only u has been added to $V \setminus \mathsf{Front}$, $\mathbf{m} \xrightarrow{\sigma} \mathbf{m}' \geq \mathbf{m}^*$ is still an exploring sequence.
- Otherwise, pick the greatest index $i \leq k$ such that $\lambda(w) + \mathbf{C}(\sigma_0 t_1 \sigma_1 \cdots t_i \sigma_i) \leq \lambda(u)$. If $i = k$, then $\lambda'(u) = \lambda(u) \geq \lambda(w) + \mathbf{C}(\sigma_0 t_1 \sigma_1 \cdots t_k \sigma_k) = \mathbf{m}' \geq \mathbf{m}$ implying that $\mathbf{m}$ is quasi-covered. If $i < k$ then denote by $\sigma' = t_{i+1} \sigma_{i+1} \cdots t_k \sigma_k$, this suffix of σ. σ' is fireable from $\lambda(u)$, hence t_{i+1} is fireable from $\lambda(u)$. So u has a child $v_{t_{i+1}}$ in V' with $\lambda'(v_{t_{i+1}}) = \lambda(u) + \mathbf{C}(t_{i+1})$. Let $\sigma'' = \sigma_{i+1} t_{i+2} \cdots t_k \sigma_k$. Then σ'' is fireable from $\lambda'(v_{t_{i+1}})$ and $\lambda'(v_{t_{i+1}}) \xrightarrow{\sigma''} \mathbf{m}'' \geq \mathbf{m}' \geq \mathbf{m}$. Since only u has been added to $V \setminus \mathsf{Front}$, for all $i < j \leq k$ there does not exist $v' \in V' \setminus \mathsf{Front}'$ such that $\lambda_u(v_{t_{i+1}}) + \mathbf{C}(\sigma_{i+1} t_{i+2} \cdots t_j \sigma_j) \leq \lambda_u(v')$. Therefore, since $v_{t_{i+1}} \in \mathsf{Front}_u$, $\lambda'(v_{t_{i+1}}) \xrightarrow{\sigma''} \mathbf{m}''$ is an exploring sequence. □

Combining all the above, we get that all $\mathbf{m} \in Cover(\mathcal{N}, \mathbf{m}_0)$ are quasi-covered.

Proposition 4.3.12. *At the beginning of every iteration, all* $\mathbf{m} \in Cover(\mathcal{N}, \mathbf{m}_0)$ *are quasi-covered.*

Proof. Let us prove by induction on the number of iterations that all $\mathbf{m} \in Cover(\mathcal{N}, \mathbf{m}_0)$ are quasi-covered.

Let us consider the base case. `MinCov` initializes V and Front to $\{r\}$ and $\lambda(r)$ to $\mathbf{m}_{ini}$. By definition, for all $\mathbf{m} \in Cov(\mathcal{N}, \mathbf{m}_{ini})$, there exists $\sigma = t_1 t_2 \cdots t_k \in T^*$ such that $\mathbf{m}_{ini} \xrightarrow{\sigma} \mathbf{m}' \geq \mathbf{m}$. Since $V \setminus \mathsf{Front} = \emptyset$, this firing sequence is an exploring sequence.
Assume that all $\mathbf{m} \in Cover(\mathcal{N}, \mathbf{m}_0)$ are quasi-covered at the beginning of some iteration. Let us examine what may happen during the iteration. In lines 4-6, `MinCov` computes the maximal fireable sequence $\sigma \in \mathsf{Acc}_n^*$ from $\lambda_n(u)$ (line 4) and sets u's marking to $\mathbf{m}_u := \lambda_n(u) + \mathbf{C}(\sigma)$ (line 5). Afterwards, there are three possible cases: (1) either $\mathbf{m}_u$ is covered by some marking associated with a vertex out of Front, (2) either an acceleration is found, (3) or `MinCov` computes the successors of u and removes u from Front.

Line 7. `MinCov` calls `Delete`(u). So Tr_{n+1} is obtained by deleting u. Moreover, $\lambda(u') \geq \mathbf{m}_u$. Let us check the hypotheses of Lemma 4.3.8. Assertion 1 follows from induction since (1) the only change in the data is the increasing of $\lambda(u)$ by firing some accelerations and (2) u belongs to Front, so it cannot cover intermediate markings of exploring sequences. Assertion 2 follows from Proposition 4.3.2 since $V \setminus \mathsf{Front}$ is unchanged. Assertion 3 follows immediately from lines 4-6. Assertion 4 follows with $v = u'$. Thus, using this lemma, the induction is proved in this case.

Lines 8-16. Let us check the hypotheses of Lemma 4.3.9. Assertions 1 and 2 are established as in the previous case. Assertion 3 holds due to Lemma 4.3.3, and the fact that no edge has been added since the beginning of iteration. Thus, using this lemma, the induction is proved in this case.

Lines 18-25. We first show that the hypotheses of Lemma 4.3.11 hold before line 21.

Let us denote the values of Tr and Front after line 20 by $\widehat{Tr}_n$ and $\widehat{\mathsf{Front}}_n$. Observe that for all iteration of Line 19 in the inner loop, the hypotheses of Lemma 4.3.10 are satisfied. Therefore, in order to apply Lemma 4.3.11 it remains only to check assertions 2 and 3 of this lemma. Assertion 2 holds since (1) $\lambda(V \setminus \mathsf{Front})$ is an antichain, (2) due to Line 7 there is no $w \in V \setminus \mathsf{Front}$ such that $\lambda(w) \geq \lambda(u)$, and (3) by iteration of Line 19 all $w \in V \setminus \mathsf{Front}$ such that $\lambda(w) < \lambda(u)$ have been deleted. Assertion 3 holds due to Line 5 (all useful enabled accelerations have been fired) and Line 8 (no acceleration has been added).

Lines 21-25 correspond to the operations related to Lemma 4.3.11. Thus, using this lemma, the induction is proved in this case. □

The completeness of `MinCov` is an immediate consequence of the previous proposition.

Corollary 4.3.13. *When `MinCov` terminates,* $Cover(\mathcal{N}, \mathbf{m}_0) \subseteq [\![\lambda(V)]\!]$.

Proof. By Proposition 4.3.12 all $\mathbf{m} \in Cover(\mathcal{N}, \mathbf{m}_0)$ are quasi-covered. Since on termination, Front is empty for all $\mathbf{m} \in Cover(\mathcal{N}, \mathbf{m}_0)$, there exists $v \in V$ such that $\mathbf{m} \leq \lambda(v)$. □

Chapter 5

The Tool `MinCov`

In the previous chapter we have introduced the algorithm `MinCov` which computes the Clover to a given marked Petri net. This algorithm does not only fix the bug in the algorithm from [56] but also introduces new ideas. In order to demonstrate the efficiency of `MinCov` in generating the Clover, we implemented it. We will refer to its implementation with the same name, i.e., `MinCov`. Our goals in this chapter are to 1) discuss some extra optimizations used, 2) describe some details of its implementation, and 3) benchmark `MinCov`.

Benchmarks and measurements. As usual, in order to benchmark `MinCov`, we need to have answers to the following 3 questions:

1. What are the benchmarks?
2. What are we measuring?
3. Against what are we comparing `MinCov`?

The instances of the benchmarks are a set of marked Petri nets to which we are asked to generate the Clover. These Petri nets come from two sources: 1) a set of standard benchmarks from the literature, and 2) a set of randomly generated Petri nets generated in house. We will be measuring two indicators. The first one is the time it takes to generate the Clover. The second one is the maximal space taken by the algorithm. However, since the tools we are comparing are not necessarily written in the same language, we devise a way to compare the memory usage in an abstract way. Finally, the tools we will compare `MinCov` to are the tools described in the Section 4.1.

Implementation and optimizations. In order to implement `MinCov` we had to first clarify the parts of the algorithm which were left ambiguous (on purpose). For example, the exploration strategy, i.e., the implantation of "`Select` $u \in$ Front" (Line 3). Second, we need to deiced on low-level details of the design, such as the data structures. Lastly, we note that the "vanilla" version of `MinCov` is very efficient in space as compared to all the other tools (as we will see below), but it

is not the fastest one. For that, we develop some new heuristics and optimizations that will give us the missing edge.

Coverability. Recall that a coverability problem provides one with a marked Petri net $(\mathcal{N}, \mathbf{m}_0)$, a target marking $\mathbf{m}$, and asks you whether $\mathbf{m} \in Cover(\mathcal{N}, \mathbf{m}_0)$. Using our `MinCov` we can solve coverability problems by computing the Clover and checking whether $\mathbf{m} \in Cover(\mathcal{N}, \mathbf{m}_0)$. A straight forward optimization for coverability, `MinCov` can check in each iteration whether the target is already covered. By doing so `MinCov` is very fast in proving that the target is *unsafe*, i.e., coverable. Unfortunately, proving that a target is *safe* (i.e., uncoverable) necessitates generating the entire Clover, which can be very slow and require large memory. On the other hand, the tool `qCover`[26] is very fast in proving the target is safe, but it is slow proving unsafe. The question is whether it is even possible to have a tool which shows safety and unsafety quickly. We will show that by using the ideas from both `MinCov` and `qCover` gives us a tool which is greater than the sum of its parts.

Organization. In Section 5.1 we discuss the different exploration strategies one can use while running `MinCov`. In Section 5.2 we describe different high-level optimizations for `MinCov`. In Section 5.3 we describe how to use `MinCov` to solve the coverability problem efficiently. In Section 5.4 the implantation of `MinCov` is detailed. In Section 5.5 we present benchmarks comparing `MinCov` to other tools.

Based on. This chapter is mainly based on our work in [60, 61].

5.1 Exploration Strategies

In the being of each iteration of the main while loop (Line 3) the Algorithm 5 picks a vertex from the Front. In previous section we did not specify how does the algorithm choose this vertex. This is because our algorithm works correctly with any type of exploration. But, not all exploration strategies are equivalent for performance. Similar to the approach of Valmari and Hansen [152] we tried 3 types of explorations: Breadth First Search (BFS), Depth First Search (DFS), and Most Tokens First (MTF). BFS and DFS are usual graph explorations strategies, i.e., BFS explores all the neighbors of the current vertex first, and DFS explorers the descendants of its current vertex first. The MTF strategy turns the Front to a max-heap according to lexicographic order on vectors of natural numbers, $h \in \mathbb{N}^3$, calculated as follows:

1. h_1 is the number of places with ω of the vertex markings;
2. h_2 is the sum of the tokens in places which are not ω;
3. h_3 is the distance of the node from the root, i.e., the depth of the vertex in the tree.

For example, given the vertices v_1, v_2, v_3, v_4 labeled with the markings:

$$\underbrace{\begin{pmatrix}1\\1\\\omega\\0\end{pmatrix}}_{m_1}, \underbrace{\begin{pmatrix}1\\1\\1\\\omega\end{pmatrix}}_{m_2}, \underbrace{\begin{pmatrix}0\\0\\\omega\\\omega\end{pmatrix}}_{m_3}, \underbrace{\begin{pmatrix}0\\\omega\\0\\\omega\end{pmatrix}}_{m_4}$$

and depths$d_1 = d_2 = d_3 = 1$, and $d_4 = 5$. The vectors h_1, h_2, h_3, h_4 associated with $v_1, ..., v_4$ are:

$$\underbrace{\begin{pmatrix}1\\2\\1\end{pmatrix}}_{h_1} \leq \underbrace{\begin{pmatrix}1\\3\\1\end{pmatrix}}_{h_2} \leq \underbrace{\begin{pmatrix}2\\0\\1\end{pmatrix}}_{h_3} \leq \underbrace{\begin{pmatrix}2\\0\\5\end{pmatrix}}_{h_4} \begin{pmatrix}\#\text{ of }\omega\\\text{sum of not }\omega\\\text{depth}\end{pmatrix}$$

The MTF strategy would explore the vertex in the following order: v_4, v_3, v_2, v_1.

Running three versions of exploration strategies, we did not find a clear preference for the exploration strategy. In total, the fastest exploration strategy was DFS. However, the other strategies were not far behind, the MTF strategy had less than 1% difference in total time, and the BFS strategy had less than a 1.5% difference. However, it is important to note that there exist instances of benchmarks for which each of the strategies preformed better than the other, with speed up of $\times 4$ faster. Moreover, we note that there are numerous nodes which are deleted not during the accelerations. These subsets of nodes could be left unexplored if we had some better oracle to guide the exploration. We conjecture that there exists a better exploration strategy that can have significant speed-ups.

In the rest of this chapter when measuring the performance of `MinCov` we use the DFS exploration strategy unless specified otherwise.

5.2 Optimizations

The greatest strength of `MinCov`, compared to its competition, is that it is very memory efficient (as we will show on the benchmarks in the next section). However, for `MinCov` to compete against the fastest tools, there are still some optimizations to be done. These optimizations come in two 'tastes': Figuring out the appropriate data structures (will be discussed in Section 5.4) and heuristics. These optimizations are an attempt to deal mainly with the bottleneck identified by Piipponen and Valmari [129] for their algorithm `VH` which also holds for `MinCov`. Namely, comparing the current marking to all the ones previously discovered (lines 7 and 18-20) is extremely time-consuming.

To empirically benchmark the optimizations below, we ran the different versions of `MinCov` on a subset (of size 15) of the benchmarks from the literature (described in the next section). The execution of these benchmarks is limited to 900 seconds. When we compare the time between different versions of `MinCov` , the time measurement consists of the total time on instances that did not time out plus 900 seconds for any instance that led to a timeout.

Parallel checking. For each iteration of Algorithm 5 (implementing `MinCov`), in which we do not delete the current node (denote by u the current vertex), we check whether $\lambda(u) \leq \lambda(v)$ or whether $\lambda(v) < \lambda(u)$ for every vertex $v \in V$.

In both of these checks, we iterate over all the vertices, which ends up being very expensive timewise. In this optimization, instead of iterating twice over all the vertices we iterate once, checking for every vertex v whether $\lambda(u) \leq \lambda(v)$ or $\lambda(v) < \lambda(u)$. Note that on one hand, if we find v such that $\lambda(u) \leq \lambda(v)$ we: 1) stop comparing vertices, 2) delete the vertex u, and 3) restart the main loop. On the other hand, if we find a vertex v such that $\lambda(v) < \lambda(u)$ we do not need to continue checking whether $\lambda(u) \leq \lambda(v)$, since we know that the set V is an anti-chain. Finally, if u is incomparable to any other vertex, then there is also no reason to check for accelerations.
Results on small benchmarks: Ruining the benchmarks, we got around 40% performance boost compared to the vanilla version.

Lazy check. As we discussed above, one of the main bottlenecks of `MinCov` is comparing a current marking to the ones already discovered. This action has an effect on the current set of vertices if one of the two happens: 1) `MinCov` discovers a marking larger the current one, and it deletes the current marking, or 2) it finds a marking smaller than the current one, and it removes the subtree rooted at it. Running `MinCov` on the benchmarks, one can notice that both cases do not happen often, around 2% of iteration. Hence, comparing the current marking to the ones already discovered is "useless" a lot of times. Therefore, we stop comparing the current marking at every iteration, instead we check it every $\texttt{CI} \in \mathbb{N}$ iteration of the main loop. The value of `CI` is dynamic, i.e., it changes between each iteration as follows: In the first iteration, we set it to $\texttt{CI} = 1$ (i.e., check every iteration). Its value changes if and only if in the current iteration `MinCov` compares the current marking to the previous ones. It changes as follows:

If `MinCov` found marking comparable to the current marking then:

$$\texttt{CI} \leftarrow \max\{\frac{\texttt{CI}}{100}, 1\}$$

Else:

$$\texttt{CI} \leftarrow \min\{\ 2\texttt{CI}, 5000\}$$

It is important to note that in each iteration we check for accelerations, even if we do not compare the current marking to the one previously discovered. Finally, while using this optimization, we don't use the DFS exploration but the MTF one. This is due to the fact that not only do we get 5% speed up compared to DFS, but we also get 1% less memory usage.
Results on small benchmarks: Ruining the benchmarks, we got around 65% performance boost compared to the vanilla version.
One would think that we would pay a heavy price in the form of larger memory usage by performing the Lazy optimization, fortunately this is not the case. `MinCov` is a very destructive algorithm, i.e., it can remove entire subtrees of vertices in one iteration. We compare the memory usage by counting for each benchmark that did not time out (on both versions) the number of the peak number of vertices in instances, plus the peak number of accelerations. Comparing to the vanilla version, we get an extra 5% (15851 vs 16621). Moreover, if we compare to `MinCov` using all other optimizations except the lazy one (fore which less benchmarks time-out), we get a smaller increase of 2.5% (35570 vs 36475).

Since in the rest of the chapter we are going to use the Lazy and non-Lazy version, we denote the Lazy version by `L-MinCov`, and the non-Lazy by `MinCov`.

5.3 `MinCov` and the coverability problem

As discussed in the introduction, we are looking for a tool that is good at proving safety and unsafety. As mentioned before (and this will be demonstrated in Subsection 5.5.3), `MinCov` is quite quick in showing that a benchmark in unsafe, and slow for proving safety.

`qCover` is an implementation of an algorithm solving the coverability problem, designed in [26]. The heart of its approach is a continuous over-approximation of the reachability set using the reachability set of continuous Petri nets. This tool `qCover` , contrary to `MinCov` , is very fast when showing that an instance is safe but very slow otherwise.

The most straight forward implementation of a tool combining the strength of these two tools is running them in parallel. This simple strategy gives quite a big boost of performance of up to $\times 3$ faster than `qCover` or `MinCov`.

Now, instead of just running the two tools in parallel, we can take the idea of over-approximation using Linear-Programming (LP) from `qCover` and implement it in `MinCov`. Given a Petri net $\mathcal{N}$ such that $T = \{t_i\}_{i=1}^{n}$. Let its *incidence matrix* be:

$$\mathbf{C} = \begin{pmatrix} | & | & & | \\ \mathbf{C}(t_1) & \mathbf{C}(t_2) & \cdots & \mathbf{C}(t_n) \\ | & | & & | \end{pmatrix}$$

Given an initial marking $\mathbf{m}_0$ and a target marking $\mathbf{m}$. If $\mathbf{m}$ is coverable, then there exists $x \in \mathbb{N}^P$ such that $C \cdot x \geq \mathbf{m} - \mathbf{m}_0$. Moreover, the problem of finding an $x \geq \mathbf{0}$ such that $C \cdot x \geq \mathbf{m} - \mathbf{m}_0$ can be solved by an LP solver. Therefore, given a current vertex v in some iteration of a run of `MinCov`. Using an LP solver, we can check whether there exists $x \in \mathbb{R}^P$ such that $C \cdot x \geq \mathbf{m} - \lambda(v)$. If x does not exist, we do not explore its children. In order to implement these changes into `MinCov`, all we need is to add 3 lines to the original code. See Algorithm 6 where the red lines are the modified ones.

Moreover, we can use this idea in implementing a smart exploration strategy. Recalling the LP is an optimization problem, which not only finds the solution to x but also finds the minimal one with regard to some $y \in \mathbb{R}^P$. LP finds an x which minimizes $y^T x$ that is subjected to:

$$\begin{aligned} C \cdot x &\geq \mathbf{m} - \mathbf{m}_0 \\ x &\geq \mathbf{0} \end{aligned}$$

Taking $y = \mathbf{1}$ (the vector with all entries equal to one), the resulting x has the property that for any covering sequence $\mathbf{m}_0 \xrightarrow{\sigma} \mathbf{m}' \geq \mathbf{m}$:

$$\sum_{p \in P} x(p) \leq |\sigma|$$

Now, assume that at some time, the set `Front` (in `MinCov`) contains two vertices v_1, v_2 and by using an LP solver, it produced a solution x_1, x_2 for each of v_1, v_2. If $\sum_{p \in P} x_1(p) \leq \sum_{p \in P} x_2(p)$, it would mean that potentially a marking that would cover the target is "closer" to the vertex v_1. Therefore, it would be more beneficial to explore v_1 first. We implement this type of graph exploration, calling it A^*. Finally, using solvers is a very expensive action, and it slows `MinCov` to a crawl if used on every iteration, therefore we only use it sporadically.

5.4 Implementation

`MinCov` is implemented in Python 3.7 using the NumPy, Scipy and the Z3-solver libraries, and it is around 2000 lines of code. `MinCov` imports Petri net in .spec format from Mist[1]. It can be found on GitHub in here:
https://github.com/IgorKhm/MinCov

`MinCov` uses a large amount of memory to store all the vertices and manipulate them, therefore using the right type of data structure is paramount. In the following, we point out the two data structures that we used in the implementation:

[1] `https://github.com/pierreganty/mist/wiki`.

Algorithm 6: Checking for coverability with LP

```
MinCov(𝒩, m_ini)
Input: A marked Petri net (𝒩, m_ini) and a target m
Data: V set of vertices; E ⊆ V × V; Front ⊆ V; λ : V → ℕ_ω^p;
      δ : E → T_o Acc*; Tr = (V, E, λ, δ) a labeled tree; Acc a set of
      ω-transitions;
Output: True if m ∈ Cov(𝒩, m_ini), false otherwise
1  V ← {r}; E ← ∅; Front ← {r}; λ(r) ← m_ini; Acc ← ∅; δ(r, r) ← ε
2  while Front ≠ ∅ do
3      Select u ∈ Front
4      Let σ ∈ Acc* a maximal fireable sequence of accelerations from λ(u)
5      λ(u) ← λ(u) + C(σ)
6      δ((prd(u), u)) ← δ((prd(u), u)) · σ
7      if ∃u' ∈ V \ Front s.t. λ(u') ≥ λ(u) then Delete(u)
8      else if ∃u' ∈ Anc(V) s.t. λ(u) > λ(u') then
9          Let γ ∈ E* the path from u' to u in Tr
10         a ← NewAcceleration()
11         foreach p ∈ P do
12             if C(p, δ(γ)) < 0 then Pre(p, a) ← ω; C(p, a) ← ω
13             if C(p, δ(γ)) = 0 then Pre(p, a) ← Pre(p, δ(γ)); C(p, a) ← 0
14             if C(p, δ(γ)) > 0 then Pre(p, a) ← Pre(p, δ(γ)); C(p, a) ← ω
15         end
16         a ← trunc(a); Acc ← Acc ∪ {a}; Prune(u'); Front = Front ∪ {u'} ;
17     else
18         for u' ∈ V do
19             if λ(u') < λ(u) then Prune(u'); Delete(u')
20         end
21         Front ← Front \ {u}
22         if ∃x s.t. C · x ≥ m − λ(u) then
23             foreach t ∈ T ∧ λ(u) ≥ Pre(t) do
24                 u' ← NewNode(); V ← V ∪ {u'}; Front ← Front ∪ {u'});
                   E ← E ∪ {(u, u')}
25                 λ(u') ← λ(u) + C(t); δ((u, u')) ← t
26             end
27         end
28         if λ(u) ≥ m then return true
29     end
30 end
31 return false
```

Hash table for equality. Due to the structure of Petri nets, there are often multiple runs to reach the same markings. For this reason, during a run of MinCov one gets many duplicates of marking already processed. Recall, that hash tables are very efficient in finding equality. Therefore, keeping a hash table of all the current markings (the set V), we can check quickly for equality. This optimization don't require a lot of memory since the hash table can consist of pointers to the markings already in memory. This type of optimization is widely used in algorithm dealing with reachability of Petri nets, and was first suggested for coverability trees in [152].
Results on small benchmarks: Ruining the benchmarks, we got around 30% performance boost compared to the vanilla version.

Short-transitions. Looking at Petri nets from the literature, one notices that **Pre** and **C** vectors associated to transitions are sparse. By sparse, we mean that the number of places $p \in P$ for which $\mathbf{Pre}(p) = \mathbf{C}(p) = 0$ is more than 95% of the total number of places. Therefore, instead of storing the entire transition as a vector, we store it as two lists of tuples, where each tuple is a place and an integer. For example, the transition t with $\mathbf{Pre}, \mathbf{C} \in \mathbb{N}^P$ where $P = \{p_i\}_{i=1}^4$, turns into two lists each of length 2 (since only two entries were not equal to 0):

$$\mathbf{Pre}(t) = \begin{pmatrix} 1 \\ 0 \\ 0 \\ \omega \end{pmatrix}; \mathbf{C}(t) = \begin{pmatrix} \omega \\ 0 \\ 0 \\ \omega \end{pmatrix} \Longrightarrow \begin{pmatrix} (p_1, 1) \\ (p_4, \omega) \end{pmatrix}; \begin{pmatrix} (p_1, \omega) \\ (p_4, \omega) \end{pmatrix}$$

Note that in this case, we probably don't save much, but when a transition has 1000 places with only 30 of them are none zero, we start seeing the difference.

The same idea can be used for the $\omega-$markings, but unfortunately it was not practical in our Python implantation. This is due to the use of the NumPy library to represent markings. But we use it because its array implementation is very optimized, and it works faster than if we had written one in Python on our own.
Results on small benchmarks: Ruining the benchmarks, we got around 5% performance boost compared to the vanilla version.

5.5 Benchmark results

5.5.1 Benchmarks

The benchmarks used for evaluation come in two types: 1) benchmarks from the literature, and 2) benchmarks randomly generated by us.

Benchmarks from the literature These benchmarks were taken from [25, 54] which in turn were gathered from five sources:

- **MIST**[1]: containing both real and artificial systems (mutual exclusion protocols, communication protocols, ...);
- **BFC**[2] **[84]**: systems originated from analysis of concurrent C programs;
- **SOTER**[3] **[47]**: systems originating from the analysis of Erlang programs in order to test the tool SOTER;
- **Medical [93]**: systems originating from the analysis of a simple medical messaging system of Vanderbilt University Medical Center;
- **Bug tracking [93]**: systems originating from the analysis of messages of a bug-tracking system.

Random benchmarks To generate these, we first generated random Petri nets with the following properties:

- $50 < |P|, |T| < 100$;
- the number of places connected to each transition is bounded by 10;
- the resulting Petri nets are not structurally bounded, i.e., there exists a marking $\mathbf{m}$ such that $\mathbf{m} \xrightarrow{\sigma} \mathbf{m}' > \mathbf{m}$.

For each of these we randomly picked an initial marking $\mathbf{m}_0$, and made sure that the following holds:

- $1 \leq \sum_{p \in P} \mathbf{m}_0(p) \leq \frac{|P|}{2}$
- running `MinCov` on it one of the following happened:
  - `MinCov` did not terminate after 900 seconds
  - `MinCov` terminated after longer than 1 second
  - `MinCov` terminated with a vertex set of size larger than 1000.

5.5.2 Clover generation

We compare `MinCov` and `L-MinCov` with the tool `MP` [135], the tool `VH` [152], and the tool `CovProc` [65]. We have also implemented the (incomplete) minimal coverability tree algorithm denoted by `MCT` in order to measure the additional memory needed for the (complete) tools. Both `MP` and `VH` tools were sent to us by the courtesy of the authors. The tool `MP` has two implementations one in Python and another in C++. For comparison, we selected the Python one to avoid biases due to programming language.

[2] `http://www.cprover.org/bfc/`.
[3] `http://mjolnir.cs.ox.ac.uk/soter/`

All benchmarks were performed on a computer equipped by Intel i5-8250U CPU with 4 cores, 16 GB of memory and Ubuntu Linux 18.03.

Table 5.1 contains a summary of all the instances of the benchmarks. We compare them according to the same metrics we used in the previous section, namely:

The first column shows the number of instances on which the tool timed out. The time column consists of the total time on instances that did not time out, plus 900 seconds for any instance that led to a timeout. The #Nodes column consists of the peak number of nodes in instances that did not time out on any of the tools (except `CovProc` which does not provide this number). For `MinCov` and `L-MinCov` we take the peak number of nodes plus accelerations.

Table 5.1: Benchmarks for clover (red is the best and blue second best)

123 benchmarks from the literature

	T/O	Time	#Nodes
`MinCov`	16	18127	48218
`L-MinCov`	12	13659	55541
`VH`	15	14873	75225
`MP`	24	23904	478681
`CovProc`	49	47081	N/A
`MCT`	19	19223	45660

100 random benchmarks

	T/O	Time	#Nodes
`MinCov`	14	13989	62345
`L-MinCov`	13	12565	91470
`VH`	15	13692	208134
`MP`	21	21726	755129
`CovProc`	80	74767	N/A
`MCT`	16	15888	63275

In the benchmarks from the literature, we observed that `L-MinCov` is much faster than `MinCov` (around 25% faster), the extra memory used by it is around 15% and it is still 2nd place. On the Random benchmarks the time difference between `L-MinCov` and `MinCov` are less obvious but still `L-MinCov` wins timewise, and it is not much worse in space. Both `MinCov` and `L-MinCov` are more space efficient. `L-MinCov` is the fastest tool, and it times out the least. Interestingly the tool `VH` is not far behind, and if we look only on the instances where both the tools did not time out then `VH` is 20% faster then `L-MinCov`. Looking more closely, we note that `L-MinCov` is faster than `VH` on benchmarks where the set of vertices is very large. A possible explanation could be that `VH` is implemented in C++ and therefore it is much faster on the smaller benchmarks.

5.5.3 Coverability

We compare `MinCov` and `L-MinCov` with and without A^* to the tool `qCover` [26] on the set of benchmarks from the literature in Table 5.2. In [26], `qCover` is compared to the most competitive tools for coverability and achieves a score of 142 solved

instances while the second best tool achieves a score of 122. We split the results into safe instances (not coverable) and unsafe ones (coverable). In both categories, we counted the number of instances on which the tools failed (columns T/O) and the total time (columns Time) as in Table 5.1.

Table 5.2: Benchmarks for the coverability problem (red is the best and blue second best)

	Time Unsafe	T/O Unsafe(60)	Time safe	T/O safe(115)	T/O	Time
`MinCov`	1754	1	51323	53	54	53077
`L-MinCov`	1219	1	48851	49	50	50070
`qCover`	26467	26	11865	11	37	38332
`MinCov`+A^*	186	0	16163	16	16	16349
`MinCov` $\|$ `qCover`	1841	2	13493	11	13	15334
`L-MinCov`$\|$ `qCover`	1538	1	12732	9	10	14270
`L-MinCov`+A^*	318	0	12582	10	10	12900

Next we note that, without the A^* optimization, `MinCov` and `qCover` are complementary i.e., `qCover` is faster at proving that an instance is safe and `MinCov` is faster at proving that an instance is unsafe. The 5^{th} and 6^{th} rows of Table 5.2 represent a parallel execution of the tools, where the time for each instance is computed as follows:

$$\text{Time}(\texttt{MinCov} \parallel \texttt{qCover}) = 2 \min\left(\text{Time}(\texttt{MinCov}), \text{Time}(\texttt{qCover})\right).$$

Combining both tools is around 2.5× faster than `qCover` and 3.5× faster than `MinCov`and `L-MinCov`. This is in some sense the best version possible of running them in parallel. Therefore, fusing their ideas into one tool, as done in `L-MinCov`+A^*, is greater than the sum of its parts. We get that `L-MinCov`+A^* is around 3× faster than `qCover` and 4× faster than `MinCov`and `L-MinCov`.

It is interesting to note that `qCover` is still the fastest tool for verifying safe instances, even if it timed out more than `L-MinCov`+A^*. We believe that this points at another avenue to optimization for `L-MinCov`. Namely, `qCover` uses a type of backward exploration from the target marking instead of a forward exploration like the one used by `L-MinCov`. We could maybe combine the two explorations to boost `L-MinCov`+A^* even further.

Another important detail, is that even for Unsafe (i.e., coverable) instances, `L-MinCov`+A^* is faster then `L-MinCov`. This means that `L-MinCov`+A^* found a firing sequence leading to a marking bigger than the target faster than `L-MinCov`. We strengthen the idea that having a good exploration strategy is crucial.

In the bottom line, for checking coverability it seems that in the general case it is best to use `L-MinCov`+A^*, unless one has a strong reason to suspect that the instance is safe then use `qCover` and `MinCov` $\|$ `qCover`, or unsafe then use `MinCov`+A^*.

Part II

Recursive Petri Nets

Chapter 6

Extending Petri nets

While Petri nets have been established as useful to model and analyze concurrent and distributed systems, they have inherent limitations that forbid or make it very difficult to model some real life systems. Therefore, numerous researches are devoted to extend in several directions the Petri nets model. Such extensions can be roughly split into three types:

Abbreviations. This type of extension does not extend the expressive power of Petri nets but adds structures that compacts the model while keeping the possibility to unfold the model into a Petri net for applying analysis algorithms. Let us illustrate this kind of extension by the well-known formalism of colored Petri nets [83].

- In colored Petri nets, places and transitions are equipped with a finite color domain and the items of the incidence matrices are no more integers but mappings from the color domain of a transition to the multisets over the color domain of a place. The unfolding of a colored Petri into an ordinary Petri net is obtained by considering pairs (place, color) (resp. (transition, color)) as places (resp. transitions) of the ordinary net and evaluating the mappings for these colors to determine the incidence matrices of this net. Generally the unfolding yields an exponential blow-up.
- Fortunately, for analysis purposes, dedicated methods for colored Petri nets have been developed that avoid such an unfolding. For instance in [32], Chiola et al. show how to efficiently build a symbolic reachability graph for a subclass of colored Petri nets called well-formed colored nets.
- Furthermore, letting the size of the color domains variable and considering them as parameters, it is possible to design specific analysis methods. For instance in [**?**, 39] the authors show how to compute in a parametrized way a generative family of (positive) linear invariants in large subclasses of well-

formed colored nets.

Semantic extensions. A possible semantic of a Petri net is the set of maximal (finite or infinite) firing sequences of the net. This semantic is more generally the one associated with discrete event systems. However, one often needs to first introduce timed delays between the occurrences of events (e.g., transition firings) and then to equip the set of these timed sequences with a probability measure in order to proceed with a quantitative analysis. Let us illustrate these extensions with time(d) Petri nets and stochastic Petri nets.

- In *time Petri nets* [20], each transition is equipped with a time interval. This interval is downward translated with time elapsing as long as the transition is enabled: when reaching zero its lower bound remains unchanged while time can no more elapse when some upper bound reaches zero. An enabled transition may fire when its associated interval includes zero, and after the firing its time interval and the time interval of all newly disabled transitions are reinitialized. An important advantage of this formalism is that it is able to model urgency requirements. However, all the relevant properties are undecidable when the reachability graph of the associated Petri net is infinite.
- In *timed Petri nets* [139], each arc is equipped with a time interval. The tokens have an age, and a transition can fire if every input arc has a token in the corresponding place whose age belongs to the associated interval. Then these tokens are consumed, and the token produced by an output arc has its (initial) age set to some value inside the associated interval. When time elapses, the age of all tokens evolves accordingly. Whilst this formalism cannot model urgency requirements, some relevant properties like coverability are decidable.
- In *stochastic Petri nets* [112], each transition is equipped with a rate which is the parameter of the negative exponential distribution. When a transition is newly enabled, one samples the associated distribution to get the delay before firing. The enabled transition with the smallest delay fires (such semantic is called race policy). It can be shown that the stochastic process generated by this formalism is a continuous time Markov chain (CTMC) thus allowing the standard analyses of CTMC.

Syntactic extensions. These extensions add capabilities to the transitions of the net, but the operational semantic remains the one of discrete event systems: a sequence of alternating states and transition firings. The extensions that we will present in this part are syntactic extensions. Such an extension is interesting if it is a strict extension with respect to some criterion (e.g., the family of generated

languages) and some relevant properties remain decidable. For example, a Petri net with inhibitor arcs is a Petri net with the added capability of performing zero tests, i.e., checking whether some places are empty, as a prerequisite of firing a transition. It achieves zero testing by adding incoming inhibitor arcs from its places to its transitions. At first look, Petri net with inhibitor arcs may seem very close to Petri net, but given only two inhibitor arcs (two zero tests), the net becomes equivalent to a Minsky machine [5]. Therefore, all relevant properties are undecidable.

For many years, it has been considered that there do not exist any natural extensions strictly extending the capabilities of the Petri net, but still keeping some important properties of the Petri net decidable:

> "*In general, it seems that any extension which does not allow zero testing will not actually increase the modeling power (or decrease the decision power) of Petri nets but merely result in another equivalent formulation of the basic Petri net model. (Modeling convenience may be increased.) At the same time, any extension which does allow zero testing will increase the modeling power to the level of Turing machines and decrease decision power to zero. Thus, Petri net extensions would seem to have few practical advantages for analysis.*" — James L. Peterson [127]

In 1978 this idea was shown to be wrong. In [150] Valk introduced *post self-modified* nets for which coverability is decidable, and it strictly extends Petri nets. Soon after came several extensions which manage to extend Petri nets while keeping interesting properties decidable.

In this chapter, we review in more detail shortcomings of Petri nets and some of their extensions. In Section 6.1 we describe some limitations of Petri nets. Afterwards, in Section 6.2 we review some well known syntactic extensions that strictly extend the capabilities of Petri nets. We partition these extensions according to whether their state is still an ordinary regular marking or not.

6.1 Relevant Petri Nets Limitations

There are many limitations of the Petri net model, and here we only discuss three of them. We have chosen them for their modeling importance and their relevance for theoretical researches. In order to formally characterize their limitations, we use two criteria: the family of generated languages of Petri nets (for the two first limitations) and the family of the generated transition systems (for the last one).

6.1.1 Zero testing

A classical limitation of Petri net can be found in the following example, which was inspired by the *cigarette smokers* problem [125], and developed in [94, 2, 124] (the details of this example are taken from Peterson's book [127]). Let us consider consumers, C_1 and C_2, producers, P_1 and P_2, buffers, B_1 B_2, and a shared channel Ch, as depicted in Figure 6.1. Producer P_i can produce items and store them in

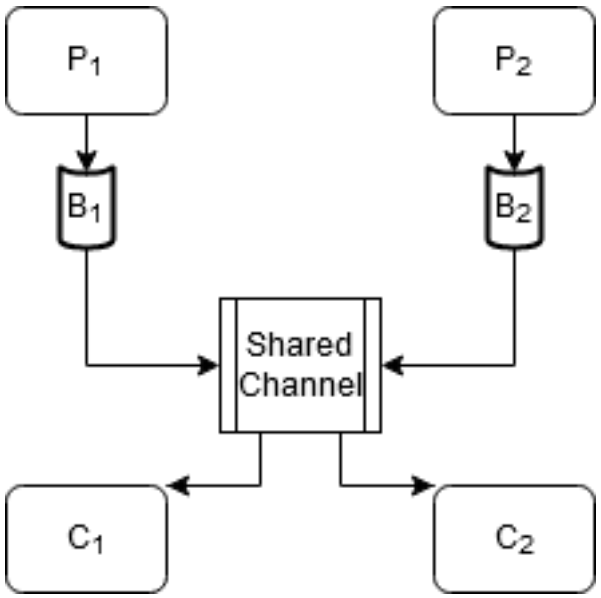


Figure 6.1: Two producers with buffers and a prioritized shared channel (this figure originates from [127]).

buffer B_i. From the buffer B_i, one can send, through the channel Ch, the items one by one to consumer C_i. In addition, there is a constraint on the shared channel where C_1 has a priority over C_2, i.e., if there is any item in buffer B_1, then the channel is closed for B_2. The impossibility to model this process using Petri nets is due to the model monotonicity (i.e., if a transition can be fired from a state, it can be fired from any other larger state) and the well-quasi order of its states. Assume for contradiction that there exists a Petri net modeling this system. Therefore, there exists a sequence of markings $\{\mathbf{m}_i\}_{i\in\mathbb{N}}$ such that $\mathbf{m}_i$ represents the state where there are i items in B_1 and one item in B_2. Since $(\mathbb{N}^P, <)$ is a well quasi-ordered set, there exist two natural numbers $i < j$ such that $\mathbf{m}_i \leq \mathbf{m}_j$. From state $\mathbf{m}_i$ there exists a firing sequence $\mathbf{m}_i \xrightarrow{\sigma} \mathbf{m}$ sending i items from B_1 to C_1 after which the channel is open to send the item from B_2 to C_2 by some sequence σ'. Since $\mathbf{m}_j > \mathbf{m}_i$, the sequence σ is also fireable from $\mathbf{m}_j$ sending i items to C_1,

i.e., $\mathbf{m}_j \xrightarrow{\sigma} \mathbf{m}' \geq \mathbf{m}$. By the monotonicity of the Petri net, σ' is fireable from $\mathbf{m}'$ emptying the buffer B_2 while B_1 is not empty. This limitation can be generally expressed as the inability to perform "zero tests" on unbounded places. Note that, if we would for example bound the size of both buffers, it would become possible to model the system as a Petri net using "complementary" places.

6.1.2 Pushdown capabilities

The stack is an abstract data type that serves as a collection of elements, with two operations, push and pop. Adding a stack to a model, i.e., adding pushdown capability is very useful for modeling relevant systems like procedural programs with unbounded integer variables [13].

An example of a system one would like to model is a syntactical analyzer of a programming language. More specifically, we are interested in verifying that all the expressions have balanced brackets. Recall from Section 2.3 that Dyck languages are the languages of balanced brackets. Assume, for contradiction, that Dyck languages are included in the family of reachability languages of Petri nets. Recall that by Theorem 2.3.7, any CFG language is equal to a homomorphism of an intersection of a Dyck language and a regular language. Since, Petri nets are closed under homomorphism and intersection by Proposition 2.3.13 and any regular language is a Petri net language by 2.3.14 we would get that any CFG language is a Petri net language. But this contradicts Proposition 2.3.15 which states that the palindrome language on two letters is not a Petri net language.

6.1.3 Modeling of faults

In the last decades there has been an intensive research about the synthesis of Petri nets (for more details see [15]). The question of synthesis asks whether taking as input a (finitely represented) transition system, there exists a Petri net whose reachability graph is isomorphic to this transition system. The synthesis problem for finite transition systems belongs to PTIME. In the following, we deal with an infinite transition system, focusing on the expressiveness point of view.

A modeling requirement suggested in [71] concerns the addition of a "faulty behavior" to an arbitrary Petri net. This behavior, when triggered, resets the current state to the initial one. For example, assume one wants to study a simple printing system which does the following:

1. **Input.** inputs data from the user;
2. **Processsing.** processes the input data;
3. **Printing.** prints the data;
4. **Finished.** returns to the input state.

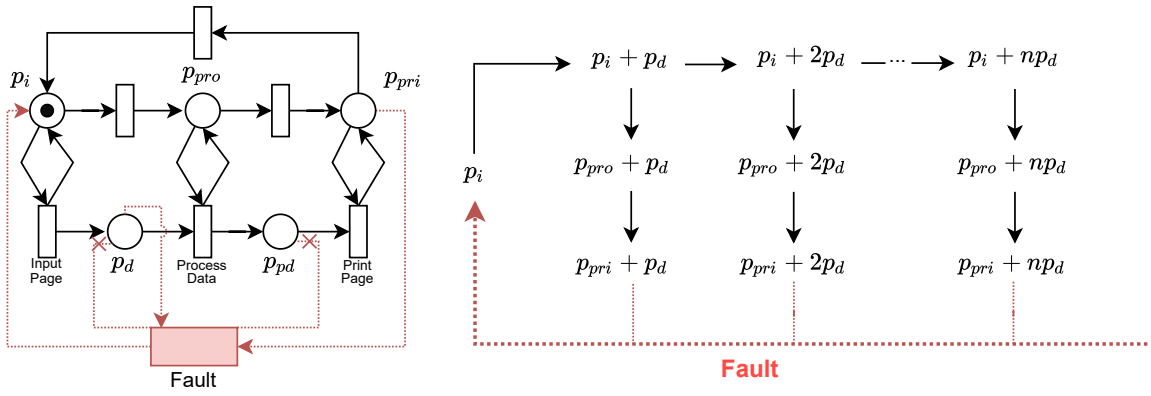


Figure 6.2: A Petri net modeling of the printer behavior.

One possible way to model this system can be seen in Figure 6.2 (the black part of the Petri net). We spot at least two faulty states from which it would be beneficial to reset the system to its initial state: (1) the system starts printing before processing all the input data (modelled by the red transition with reset arcs in Figure 6.2) and (2) the system inputs data before printing all the previous processed data.

Is it possible to synthesis this transition system as a Petri net? In order to model this functionality in an atomic way, the initial state needs to have an infinite incoming degree (i.e., infinitely many states which are a transition away from the initial one), see the right side of Figure 6.2(note that this is a subgraph of the transition system). Unfortunately, this is impossible in Petri nets, since every state in a Petri net has only a finite incoming degree.

6.2 Petri Net Extensions

In the following, we present some relevant Petri nets extensions. We split these extensions into two types, those that only extend the transition rules, and those that extend the type of possible states. This presentation is not exhaustive but related to the extensions we will be discussing in the remaining chapters of this part, namely (Dynamic) Recursive Petri nets.

6.2.1 Firing rule based extensions

Petri net with reset arcs. This extension was first introduced in [12]. These nets include reset arcs between a transition $t$and a place p. When t is fired, the marking of p is reset (i.e., all tokens of p are removed). Figure 6.3 shows a Petri

net with reset arcs. Transition t_r has two reset arcs (arcs with a cross on one of their vertices) one ends at p_d and the other at p_r. If t_r is fired, it resets the places p_d and p_r.

One can design a net, Figure 6.3, which adds one of the faulty behavior described in Sub-Section 6.1.3. When the model is inputting new data (i.e., $\mathbf{m} \geq p_i$) but it still has data ready for printing (i.e., $\mathbf{m} \geq p_r$) the transition t_r is enabled. Firing t_r resets the places p_d, and p_r. Moreover, It puts the system into its initial state (i.e., a token in p_i). Other relevant modelings can be found in numerous publications (e.g., [23, 97]).

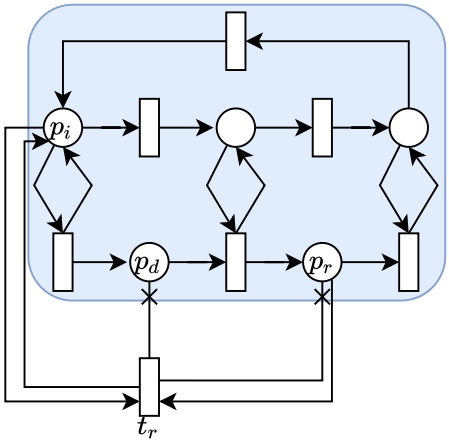


Figure 6.3: Modeling faults using reset arcs

In [12] Toshiro et al. show that the reachability problem of reset Petri nets is undecidable, and in [51] Dufourd et al. proved that it remains undecidable with two resettable places. The boundedness problem was at first erroneously thought to be decidable [92]. It was later shown in [51], that in fact the boundedness problem and the place boundedness problems are undecidable. Moreover, these authors have shown that the coverability and termination problems are decidable, using the backward procedure of WSTS (see Section 2.1). In later work [143], Schnoebelen established that the coverability and termination problems are Ackerman-hard.

Petri net with transfer arcs. This extension was first presented in [34]. Such nets include a new type of arc, allowing for the atomic movement of all the tokens currently present in a source place to a target place. Figure 6.4 illustrates such a net. In this net, one can see: an incoming transfer arc (trident head) starting at the place p_r and ending at the transition t_r, and an outgoing transfer arc (regular head but with trident tail) starting at the transition t_r and ending at the place p_s,

and labeled by p_d, p_r. Firing the transition t_r, moves all tokens from p_r and p_d to p_s (after consuming the tokens associated with the precondition of t_r).

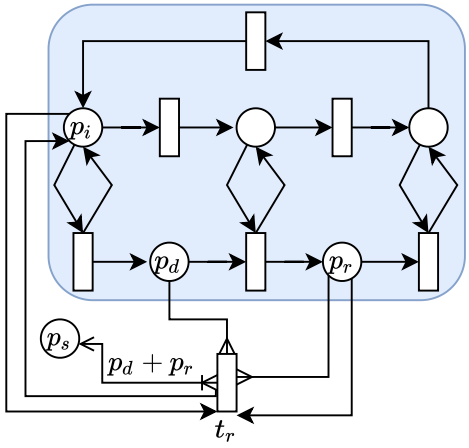


Figure 6.4: Faults using transfer arcs

Figure 6.4 describes a net with transfer arcs extending the functionalities of the Petri net of Figure 6.2. This extension provides us with a partial solution of one of the faulty behavior described in Subsection 6.1.3. When the model is inputting new data (i.e., $\mathbf{m} \geq p_i$) but it still has leftover data ready for printing (i.e., $\mathbf{m} \geq p_r$) transition t_r is enabled. Firing t_r transfers all the tokens from places p_d and p_r to place p_s and puts the system into its initial state (i.e., a token in p_i). Here we need to add a sink place p_s in which we drop all 'deleted' tokens. In fact we more or less weakly simulate a transition with reset arcs. In [164]. the author presents a process calculus which impossible to model with Petri nets, but possible using transfer arcs. For other examples of modeling with transfer arcs, see e.g. [97].

In [51] the authors show that the reachability problem for transfer Petri nets is undecidable (for a net having at least 2 extended arcs). However, the coverability and termination problems remain decidable. Surprisingly, they show that the boundedness problem is decidable, but that the place-boundedness problem is undecidable. The undecidability of the place-boundedness problem can be viewed as a consequence of the undecidability of the boundedness problem for reset nets. Similarly, as in the transfer Petri net in Figure 6.4 one can 'simulate' a reset arc by transfer arcs with a sink place. To the best of our knowledge no other Petri net extension exhibits this separation (this observation was first made in [51]).

Affine nets. This extension was first introduced in [55]. Affine nets extend the firing rule of transitions in order to include affine functions for marking updates.

For any $t \in T$ we have that $\mathbf{C}(t)$ is an affine function, i.e., for marking $\mathbf{m} \in \mathbb{N}^P$, $\mathbf{C}(t)(m) = A_t\mathbf{m} + B_t$ for some $A_t \in \mathbb{N}^{P\times P}$ and $B_t \in \mathbb{Z}^P$. The transition t is fireable from $\mathbf{m}$ if $\mathbf{m} \geq \mathbf{Pre}(t)$ and $\mathbf{C}(t)(m) \geq 0$. Firing the transition t from the marking $\mathbf{m}$, the reached marking is $\mathbf{m}' = \mathbf{C}(t)(\mathbf{m})$. Note that, for any $t \in T$, if $A_t = \mathbf{Id}$ then, the transition is equivalent to a standard Petri net transition, and we can represent it (in drawing) in the same way we do transitions in Petri nets. For transitions with $A_t \neq \mathbf{Id}$, for any place $p \in P$ where $C(t)(p) \neq p$, we draw an arc (ending with a diamond) from the transition t to the place p, and we label it with $\sum_{p'\in P} A_t(p, p')p'$. For example, see the transition t_r in Figure 6.5, where:

$$\mathbf{C}_{t_r} = \begin{array}{c} \\ p_i \\ p_2 \\ p_3 \\ p_d \\ p_r \end{array} \begin{array}{c} \begin{array}{ccccc} p_i & p_2 & p_3 & p_d & p_r \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

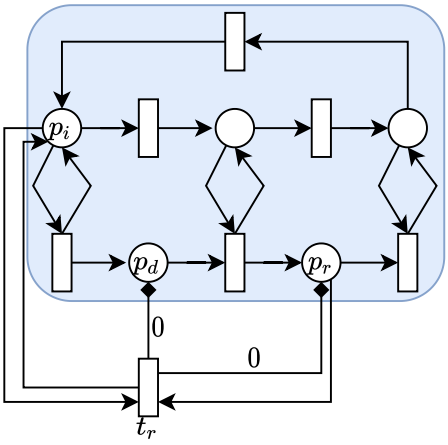


Figure 6.5: Faults using an affine net

These nets extend Petri nets with reset and transfer arcs. For instance, the affine net of Figure 6.5 simulates the net in Figure 6.3. Therefore, the reachability and boundedness problems are undecidable (as was done in [55]). On the other hand, in [55] Finkel et al. use WSTS theory to show that coverability and termination problems are decidable. Moreover, the coverability for affine nets was shown to be Ackermann-hard since it extends the coverability problem for reset nets. Hence, we get a model which extends both reset and transfer nets and where some interesting properties remain decidable.

In [30], Bonnet et al. suggest that the main reason of the undecidability of boundedness and the hardness of the coverability, comes from the ability to simulate reset arcs. So Bonnet et al. define *Strongly increasing Affine nets* (SIAN), a subclass of affine nets. In this subclass for any $t \in T$, we have that $A_t \geq Id.$. This model is equivalent to post self-modifying nets, defined by Valk in [150], the first Petri net extension shown to be a strict extension of Petri nets that keeps some important properties decidable. They show that in this subclass we get that the coverability, termination, and boundedness problems are EXPSPACE-complete.

6.2.2 State based extensions

ν-Petri nets. In ν-Petri nets, every token is labelled by a data picked from some countable set. These data are *pure names*, they can only be compared for equality (i.e., unordered data). A transition in the net is fireable if there are sufficient tokens in each place with the correct data (as specified by the arcs surrounding the transition). Firing a transition consumes the prerequisite token and creates new tokens with data. Furthermore, this data might be fresh, i.e., a value never seen before in the computation.

Figure 6.6 represents a ν-Petri net with a single transition. In order to be fireable this transition needs to have at least one token in each of its input places, and these tokens need to have different data (in our case c and a). Once it is fired it consumes these tokens, and generates a new tokens one with the labeled y (in our case it is a) and a token with a fresh label (in our case d).

The model of ν-Petri nets was originally introduced in [137] with the aim to model distributed protocols where process identities had to be taken into account. Other modeling uses of ν-Petri can be found in [120, 136].

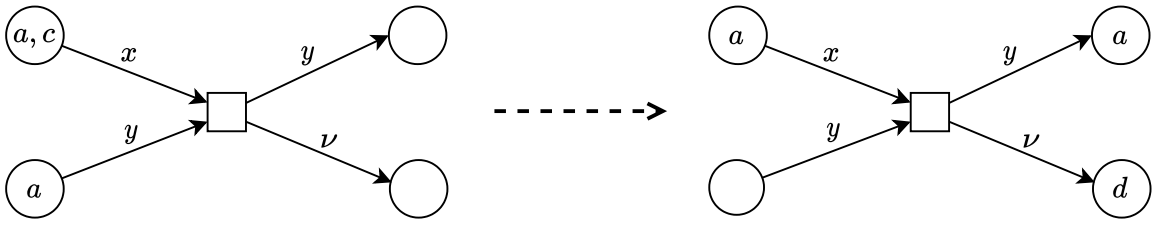


Figure 6.6: A transition firing in a ν-Petri net

Rosa et al. in [138], reduce the reachability problem in inhibitor Petri nets to the reachability problem of ν-Petri nets, entailing the undecidability of the reachability problem. Moreover, they show that the coverability and termination problems of ν-Petri nets are decidable. In [101] Lazič et al. show that coverability, boundedness and termination problems are Ackermann-hard.

Data nets. In data nets, originally defined in [99], every token carries a data picked from a linearly-ordered infinite domains and the transition firing rule includes whole-place operations (similar to the ones defined in affine nets). A transition in the net is fireable if there is a sufficient amount of tokens in each place and the data which is carried by the tokens is ordered in a way specified by the transition. Firing a transition may add tokens that carry extra data, possibly fresh as in the ν-Petri nets.

Figure 6.6 represents a data net with a single transition. In fact this is a data Petri net, i.e., a data net which does not allow whole-place operation. In order to be fireable this transition needs to have at least one token in each of its input places, where the one labeled x (in our case a) needs to be smaller than the one labeled with z (in our case c). Once it is fired, it consumes these tokens, and generates a new token labeled y such that $x < y < z$ (in our case b).

Data nets can simulate all previous discussed extensions: ν-Petri can be simulated by Data nets as done in [138] and affine nets can be simulated by data nets which do not use the data attached to the tokens.

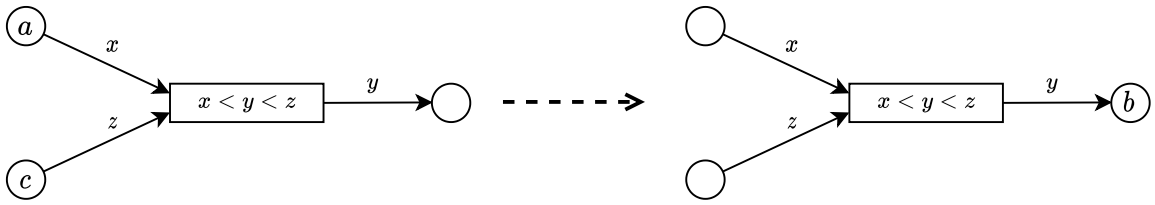


Figure 6.7: A transition firing in a Data net (assuming that $a < b < c$)

Thus, the reachability problem for data nets is undecidable, while coverability and termination problems are decidable for any data nets. Finally, the boundedness problem is only decidable for data nets in which the transitions are restricted to transfer arcs [99].

Nested Petri nets. First introduced in [108], this extension has tokens that may themselves be nets. Nested Petri nets have four type of steps: (1) transfer step - moving/generating/removing tokens containing nets, without changing their internal state, (2) object-autonomous - moving regular tokens, (3) horizontal synchronization - firing synchronously a transition between two systems associated with two tokens in the same place, and 4) vertical synchronization - firing synchronously a transition in a system on a token and in the system containing this token. Note that the possible nets attached to tokens are predefined. Moreover, the maximal depth of the system is bounded and predefined.

Nested Petri nets were used in order to model task planning systems, multiagent systems and recursive-parallel systems [109]. This model extends Petri reset net, as shown in [110]. Due to the simulation of reset Petri nets, Lamozova et al. concluded in [110] that the reachability and boundedness problems for nested Petri nets are undecidable. Moreover, using WSTS results and building an appropriate wqo for the states of Nested Petri nets, they established that the termination problem for nested Petri nets is decidable.

Pushdown Petri nets. In Pushdown Petri nets, the states are equipped with a stack which contains symbols of some alphabet. The firing rule changes accordingly: the fireabllity of a transition depends on whether there is a specific symbol on the top of the current stack, and when firing pops and/or pushes symbols in the stack.

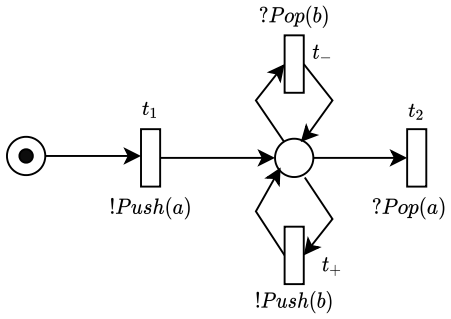


Figure 6.8: A pushdown Petri net

Figure 6.8 represents a pushdown Petri net. Firing transition t_1 pushes 'a' in to the stack and moves the token. Transition t_- pops b and transition t_+ pushes b. Note, that if b is pushed to the stack, then t_2 cannot fire until the stack is cleared from all the b's.

There is no natural wqo on the states of a pushdown Petri net, which makes it a WSTS. This makes this model stand out of all the rest of the models we introduced in this chapter. It also means that one cannot use the usual techniques (e.g., backward coverability algorithm) for proving properties such as coverability and termination. As far as we know, the decidability status of the coverability and reachability problems for pushdown Petri nets are still unknown. For pushdown Petri nets with dimension of size one, Leroux et al. show in [106], that the coverability problem is decidable. Termination and boundedness problems were shown to be decidable in [105].

Table 6.1 summarizes the decidability status and the complexity of standard problems for the above models. In the next chapters we are going to introduce Recursive Petri net which strictly extends Petri net. We will in particular establish that its family of coverability languages includes the family of CFG languages and that the coverability, termination, and boundedness problems for Recursive Petri nets are EXPSPACE-complete.

Extensions	Reachability	Coverability	Termination	Boundedness
Petri net	Ackerman-complete	EXPSPACE-Complete	EXPSPACE-Complete	EXPSPACE-Complete
Reset net	Undecidable	Ackerman-hard	Ackerman-hard	Undecidable
Transfer net	Undecidable	Ackerman-hard	Ackerman-hard	Decidable
SIAN	Undecidable	EXPSPACE-complete	EXPSPACE-complete	EXPSPACE-complete
Affine net	Undecidable	Ackerman-hard	Ackerman-hard	Undecidable
ν-Petri net	Undecidable	Ackermann-hard	Ackermann-hard	Ackermann-hard
Data net	Undecidable	Ackermann-hard	Ackermann-hard	Undecidable
Nested net	Undecidable	?	Decidable	Undecidable
Pushdown Petri net	?	?	Decidable	Decidable

Table 6.1: Decidability and complexity of standard problems for Petri net extensions

Chapter 7

Recursive Petri Nets

In this chapter, we introduce *Recursive Petri Nets* (RPN). The first appearance of RPN can be found in [52] by El Fallah et al. They noticed that the standard patterns of dynamical systems highlight two modeling needs:

1. The ability to handle the concurrent execution of parallel sequential processes;
2. The ability to manage the dynamical creation of objects.

The models that were available, then, were either not sufficiently expressive for the two patterns or led to undecidability of standard verification problems. Thus, El Fallah and Haddad introduced the RPN formalism. The RPN formalism is able to model distributed planning of multiagent systems for which counters and recursivity are necessary.

The interest of the formalism were first illustrated by modeling a scenario from the transportation domain of conveyors and clients in [52]. Other applications of RPN like systems featuring faults and interruptions, and goal-oriented programs were later discussed, one can find some examples here [71, 72].

Recall that in Section 2.3, we showed that the family of Petri net reachability languages does not include the family of context-free languages. On the contrary, the family of RPN reachability languages was shown not only to include the family Petri net reachability languages but also the one of context-free languages [71]. Moreover, since these families are incomparable, this shows that the family of RPN reachability languages is strictly larger than both (and in fact even their union). Another known model whose family of languages was shown to include both of these families is Process Algebra Net, defined in [117]. However, this model was also shown in [72] to be included in RPN.

In addition, RPN was shown to retain many of the decidable properties from Petri nets. Such properties include: reachability, coverability, finiteness etc.

RPN is not the only model extending the capabilities of Petri nets with dynamical creation of objects. For example, the *Net systems* designed in [91] has a special type of transitions, whose firing starts a new token game for the net. Moreover, these created new nets can return some tokens to their creator. However, this model can also be simulated by RPN. Another example is the *object Petri nets*, a model introduced in [151]. In this model, the tokens themselves are Petri nets. However, these nets are limited by the depth, and their reachability problem was shown to be undecidable. In the same spirit, there is also the *nested Petri nets* model, that has been introduced in [110]. Here its depth of hierarchy is not bounded but still reachability and other properties were shown to be undecidable.

Roughly speaking, a state of an RPN consists of a tree of *threads*, where the local state of each thread is a marking. Any thread may fire an *elementary*, *abstract* or *cut* transition. When the transition is elementary, the firing updates its marking as in Petri nets; when it is abstract, this only consumes the tokens specified by the input arcs of the transition and creates a child thread initialized with the *initial marking* of the transition. When a cut transition is fired, the thread and its subtree are pruned, producing in its parent the tokens specified by the output arcs of the abstract transition that created it. We leave the formal definition of the Recursive Petri Nets (RPN) to Section 7.1.

Organization. In Section 7.1 we give a formal definition of RPN. We then discuss its modeling capabilities is Section 7.2. Finally, in Section 7.3 we recall some previous results that were shown for RPN.

7.1 Presentation

The state of an RPN has a structure akin to a 'directed rooted tree' of Petri nets. Each vertex of the tree, hereafter *thread*, is an instance of the RPN and possessing some marking on it. Each of these threads can fire *three* types of transitions. An *elementary* transition updates its own marking according to the usual Petri net firing rule. An *abstract* transition consumes tokens from the thread firing it and creates a new child (thread) for it. The marking of the new thread is determined according to the fired abstract transition. A *cut* transition can be fired by a thread if its marking is greater or equal than some marking. Firing a cut transition, the thread erases itself and all of its descendants. Moreover, it creates tokens in its parent, which are specified by the output arcs of the abstract transition that created it (except when the thread is at the root of the tree, in which case it yields the empty tree).

Definition 7.1.1 (Recursive Petri Net)**.** A *Recursive Petri Net* is a 6-tuple $\mathcal{N} = \langle P, T, W^-, W^+, \Omega \rangle$ where:

- P is a finite set of places;
- $T = T_{el} \uplus T_{ab} \uplus T_{\tau}$ is a finite set of transitions with $P \cap T = \emptyset$, and T_{el} (respectively T_{ab}, T_{τ}) is the subset of elementary (respectively abstract, cut) transitions;
- W^- is the $\mathbb{N}^{P \times T}$ backward incidence matrix;
- W^+ are the $\mathbb{N}^{P \times (T_{el} \uplus T_{ab})}$ forward incidence matrix;
- $\Omega : T_{ab} \to \mathbb{N}^P$ is a function that labels every abstract transition with an initial marking;

For brevity reasons, we denote by $W^+(t)$ a vector in $\mathbb{N}^P$, where for all $p \in P$, $W^+(t)(p) = W^+(p,t)$, and we do the same for $W^-(t)$.

Figure 7.1 graphically describes an example of an RPN with:

$$P = \{p_{ini}, p_{fin}, p_{beg}, p_{end}\} \cup \{p_{b_i}, p_{a_i} : i \leq 2\};$$
$$T_{el} = \{t_{b_1}, t_{b_3}, t_{a_1}, t_{a_3}, t_{sa}, t_{sb}\}\,;\; T_{ab} = \{t_{beg}, t_{b_2}, t_{a_2}\};$$
$$T_{\tau} = \{t_{\tau_1}, t_{\tau_2}\}.$$

The elementary transitions are depicted by rectangles with a single border while abstract transitions are depicted by rectangles with a double border and cut transitions are depicted by rectangles filled in black. As for Petri nets, the items of the incidence matrices label the arcs surrounding transitions: for instance $W^-(p_{ini}, t_{beg}) = 1$. The initial markings of the abstract transitions are indicated close to the transition and framed in a rectangle: for instance $\Omega(t_{b_2}) = p_{beg}$ (where p_{beg} denotes the marking with one token in place p_{beg} and zero elsewhere).

A *concrete state* s of an RPN is a labeled tree representing relations between threads and their associated markings. Every vertex of s is a thread, and edges are labeled by abstract transitions. We introduce a countable set $\mathcal{V}$ of vertices in order to pick new vertices when necessary.

Definition 7.1.2 (State of an RPN)**.** A *concrete state (in short, a state)* s of an RPN is a tree over the finite set of vertices $V_s \subseteq \mathcal{V}$, inductively defined as follows:

- either $V_s = \emptyset$ and thus $s = \emptyset$ is the empty tree;
- or $V_s = \{r_s\} \uplus V_1 \uplus \ldots \uplus V_k$ with $0 \leq k$ and $s = (r_s, m_0, \{(m_i, s_i)\}_{1 \leq i \leq k})$ is defined as follows:
  - r_s is the root of s labelled by a marking $m_0 \in \mathbb{N}^P$;
  - For all $i \leq k$, s_i is a state over $V_i \neq \emptyset$ and there is an edge $r_s \xrightarrow{m_i}_s r_{s_i}$ with $m_i \in \{W^+(t)\}_{t \in T_{ab}}$.

For all $u, v \in V_s$, one denotes $M_s(u)$ the marking labelling u and when $u \xrightarrow{m}_s v$, one writes $\Lambda(u,v) := m$. State s_v is the (maximal) subtree of s rooted in v.

While the set of vertices V_s will be important for analyzing the behavior of a firing sequence in an RPN, one can omit it and get a more abstract representation

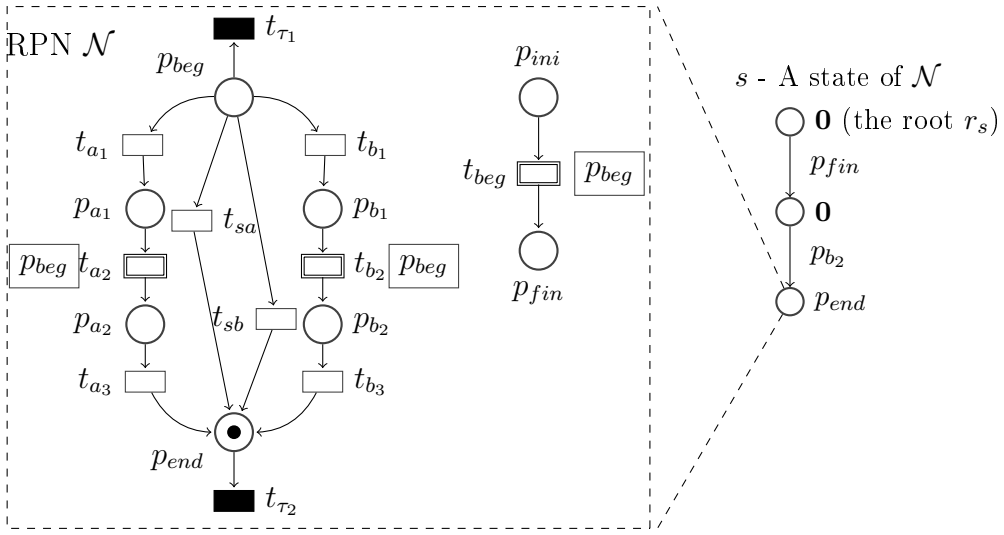


Figure 7.1: An example of a marked RPN.

of the state. Note that contrary to the previous definition where $\{(m_i, s_i)\}_{1 \leq i \leq k}$ was a set, in the following definition we need a multiset denoted by $Child_s$.

Definition 7.1.3 (Abstract state of an RPN)**.** An *abstract state* s of an RPN is inductively defined as follows:

- either $s = \emptyset$ is the empty set ;
- or $s = (m_s, Child_s)$ where $m_s \in \mathbb{N}^P$ and $Child_s$ is a finite multiset of pairs (m', s')
  where $m' \in \{W^+(t)\}_{t \in T_{ab}}$ and s' is an abstract state different from $\emptyset$.

Given a concrete state s, we denote by $[s]$ its abstract state. Except if explicitly stated, a state is a concrete state.

In the other direction, given an abstract state s, one recovers its set of concrete states by picking an arbitrary set of vertices $V_s \subseteq \mathcal{V}$ of appropriate cardinality and, inductively, arbitrarily splitting V_s between the root and the pairs (m, s').

For example, on the right side of Figure 7.1, there is a (concrete) state of the RPN $\mathcal{N}$. This state consists of three threads with markings $\mathbf{0}, \mathbf{0}$, and p_{end} (where $\mathbf{0}$ is the *null marking*) and two edges with the labels $W^+(t_{beg})$ and $W^+(t_{b_2})$.

Let s be a state of some RPN. Every thread u different from the root has a unique *parent*, denoted by $prd_s(u)$. We write $prd(u)$ when the specific state is clear. The *descendants* of a thread u consist of threads in the subtree rooted in, u including u itself. We denote this set by $Des_s(u)$. For $m \in \mathbb{N}^P$, we denote by $s[r, m] := (r, m, \emptyset)$, the state consisting of a single vertex r whose marking is m.

As usual, two markings $m, m' \in \mathbb{N}^P$, over a set of places P, are partially ordered as follows: $m \leq m'$ if for all places $p \in P$, $m(p) \leq m'(p)$.

Definition 7.1.4 (Operational semantics)**.** Let $s = (r, m_0, \{(m_i, s_i)\}_{1 \leq i \leq k})$ be a state. Then the firing rule $s \xrightarrow{(v,t)} s'$ where $v \in V_s$ and $t \in T$ is inductively defined as follows:

- Let $t \in T_{el}$ such that $W^-(t) \leq m_0$, then one has $s \xrightarrow{r,t} (r, m_0 - W^-(t) + W^+(t), \{(m_i, s_i)\}_{i \leq k})$
- Let $t \in T_{ab}$ such that $W^-(t) \leq m_0$, then one has $s \xrightarrow{r,t} (r, m_0 - W^-(t), \{(m_i, s_i)\}_{i \leq k+1}))$ where $m_{k+1} = W^+(t)$, $s_{k+1} = s[v, \Omega(t)]$ with $v \in \mathcal{V} \setminus V_s$
- Let $t \in T_\tau$ such that $W^-(t) \leq m_0$, then one has $s \xrightarrow{r,t} \emptyset$
- Let $i \leq k$ such that $s_i \xrightarrow{v,t} s'_i$
  if $s'_i = \emptyset$ then $s \xrightarrow{v,t} (m_0 + m_i, \{(m_j, s_j)\}_{1 \leq j \neq i \leq k})$
  else $s \xrightarrow{v,t} (m_0, \{m_j, s_j\}_{1 \leq j \neq i \leq k} \cup \{m_i, s'_i\})$

Figure 7.2 illustrates a sequence of transition firings in the RPN described by Figure 7.1. The first transition $t_{beg} \in T_{ab}$ is fired by the root. Its firing results in a state in which the root has a new child (denoted by v) and a new outgoing edge with label p_{fin}. The marking of the root is decreased to $\mathbf{0}$ and v is initially marked by $\Omega(t_{beg}) = p_{beg}$. The second firing is due to an elementary transition $t_{b_1} \in T_{el}$ which is fired by v. Its firing results in a state for which the marking of v is changed to $M'_s(v) = M_s(v) + W^+(t_{b_1}) - W^-(t_{b_1}) = p_{b_1}$. The fifth transition to be fired is the cut transition t_{τ_2}, fired by the thread with the marking p_{end} (denoted by w). Its firing results in a state where the thread w is erased, and the marking of its parent is increased by $W^+(t_{b_2}) = p_{b_2}$.

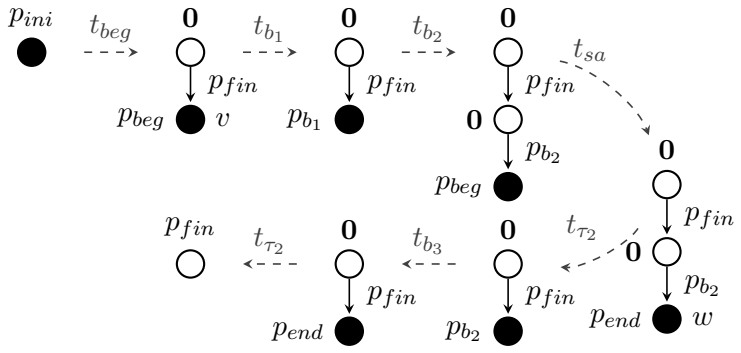


Figure 7.2: A firing sequence for the RPN in Figure 7.1.

A *firing sequence* is a sequence of transition firings, written in a detailed way: $s_0 \xrightarrow{(v_1,t_1)} s_1 \xrightarrow{(v_2,t_2)} \cdots \xrightarrow{(v_n,t_n)} s_n$, or when the context allows it, in a more concise

way like $s_0 \xrightarrow{\sigma} s_n$ for $\sigma = (v_1, t_1)(v_2, t_2)\dots(v_n, t_n)$. Let $\sigma \in T^*$ with $\sigma = t_1 \dots t_n$ and v be a vertex, (v, σ) is an abbreviation for $(v, t_1)\dots(v, t_n)$. When we deal with several nets, we indicate by a subscript in which net, say $\mathcal{N}$, the firing sequence takes place: $s_0 \xrightarrow{\sigma}_{\mathcal{N}} s_n$. Infinite firing sequences are similarly defined. In a firing sequence, a thread v that has been deleted is *never reused* (which is possible since $\mathcal{V}$ is countable). A thread is *final* (respectively *initial*) w.r.t. σ if it occurs in the final (respectively initial) state of σ. We say that $v \in Des_\sigma(u)$ if there exists $i \leq n$ such that $v \in Des_{s_i}(u)$. We call σ' a *subsequence* of σ, denoted by $\sigma' \sqsubseteq \sigma$, if there exists k indexes $i_1, i_2 \dots i_k$ such that $1 \leq i_1 < i_2 < \dots i_k \leq n$ and $\sigma' = (v_{i_1}, t_{i_1})(v_{i_2}, t_{i_2})\dots(v_{i_k}, t_{i_k})$.

Remark. In the sequel, when we write "RPN $\mathcal{N}$", we mean $\mathcal{N} = \langle P, T, W^+, W^-, \Omega \rangle$, unless we explicitly write differently. An RPN $\mathcal{N}$ equipped with an initial state s is a *marked RPN* and denoted $(\mathcal{N}, s)$.

For a marked RPN $(\mathcal{N}, s_0)$, let $Reach(\mathcal{N}, s_0) = \{[s] \mid \exists \sigma \in T^* \text{ s.t. } s_0 \xrightarrow{\sigma} s\}$ be its *reachability set*, i.e., the set of all the reachable *abstract* states.

7.2 Modeling capabilities

From a modeling point of view, the RPN formalism have many features which make it more appropriate for specification than Petri nets. In this section, we give three examples of behaviors that are "... are difficult or even impossible to specify with typical models such like Petri nets... " [72]. These examples are directly taken from [72].

7.2.1 Faults

As already described in Section 6.1 faults are a possible feature of many systems, for which Petri nets do not have a "good" modeling pattern. On the contrary, in RPN it is very simple. For example Figure 7.3 depicts an RPN "extending" the Petri net from Figure 6.2 by adding one of the faulty behaviors described in Subsection 6.1.3 (the printer should not receive new information if it did not finish printing). In this RPN we add cut transitions t_{fault}, abstract transition t_{run} and place p_{ready}. The initial state is $s[p_{ready}, r]$. To "turn on" the printing machine one fires t_{run} which creates a new thread. This thread behaves as the Petri net in Figure 6.2, except when it encounters a faulty behavior (the marking of the thread is larger than $p_i + p_r$) where it can fire the cut transition t_{fault}. Firing this transition removes the thread and puts a token in the place p_{ready} of the thread r, from which we can restart the printer.

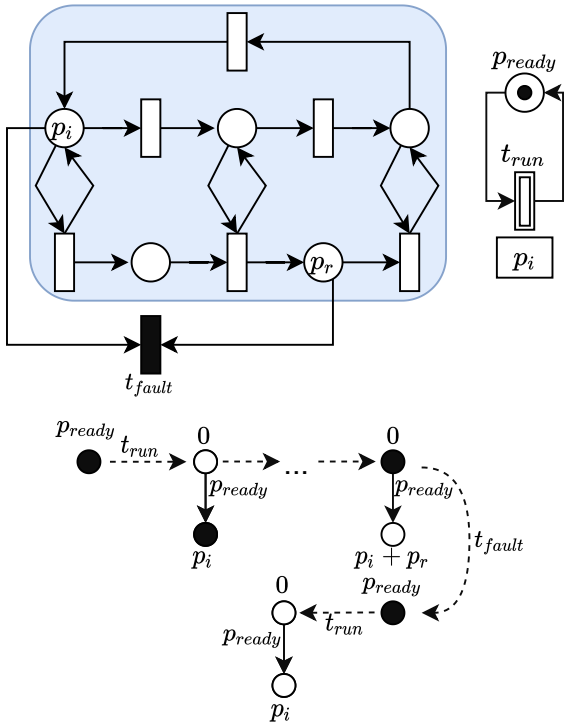


Figure 7.3: Faults using an RPN

This behavior can be easily added to any Petri net. See Figure 7.4 where we add a cut transition to any faulty marking of the net and a mechanism which runs a "fixing" protocol.

7.2.2 Concurrent goal-oriented programs

Goal-oriented programs consist of applying some rules until some predicate is reached. Concurrent goal-oriented programs are systems which run multiple concurrent process trying to fulfill the same goal. Once some process achieves the goal, the entire program terminates. Modeling such systems may be difficult, since, for example, there might exist a final state which is reachable from infinitely many states.

RPN's are well-equipped to model them. For example, let us look the RPN of Figure 7.5. The goal of the system is to reach the empty state. Its initial state consists of a single thread with the marking $p_{1,1} + p_{1,2}$. In order to cut this thread, the thread needs to get at least one token in $p_{1,3}$. For that, it is enough to return after firing one of the abstract transition $t_{1,1}$ or $t_{1,2}$. These transitions can be fired

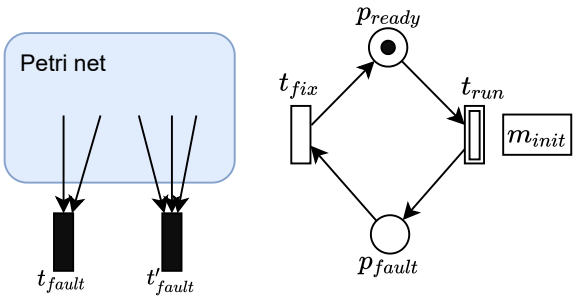


Figure 7.4: Modeling faults with RPN

concurrently, and the goal of the system can be reached once any of them finishes. Note that, firing transition $t_{1,2}$ creates a thread with a token in $p_{1,3}$, which can create an unbounded branch of threads by firing $t_{3,2}$. Therefore, the empty state can be reached from an infinite number of states.

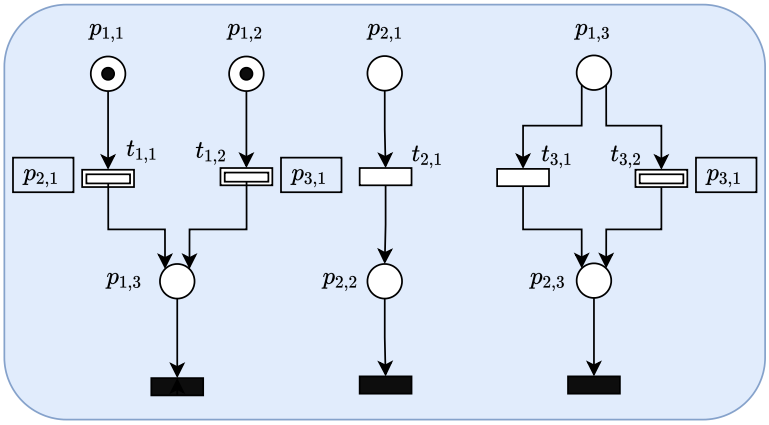


Figure 7.5: Modeling a concurrent goal-oriented program with RPN from [72]

7.2.3 Interruptions

Let us suppose that we have some system for which we want to add interruption functionality. In RPN this functionality can be simply modeled similarly like depicted in Figure 7.6. All the transitions of the original system (in blue) require at least one token in p_{int1}. The interrupt action is realized by firing the abstract transition t_{int1}. This transition takes the token from $pint1$ disabling every transition in the original system until the interrupt is handled. Moreover, the interrupt handling mechanism has the same interrupt system modeled by p_{int2} and t_{int2}, which

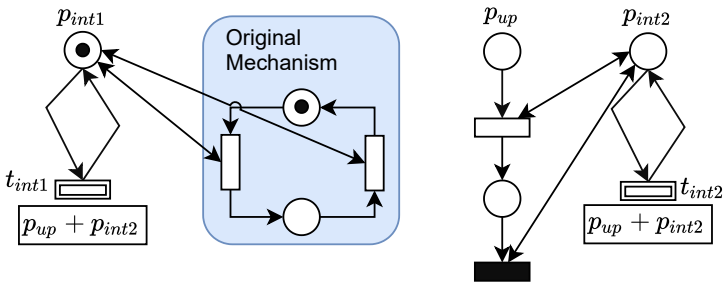


Figure 7.6: Modeling interruption mechanism using an RPN from [72]

creates a recursive interruption process. In comparison, the same modeling with Petri net is rather difficult as it requires keeping each context of the suspended processes.

7.3 Previous results about the analysis and expressiveness of RPN

7.3.1 Decision Problems

As discussed for other extensions of Petri nets in Chapter 6, it is interesting to compare the decidability of relevant properties inherited from Petri nets. Let us quickly recall some of these properties. Let $(\mathcal{N}, s_0)$ be a marked RPN and s_f be a state of $\mathcal{N}$, then:

- The *reachability problem* asks whether there exists a firing sequence σ such that $s_0 \xrightarrow{\sigma} s_f$.
- The *termination problem* asks whether there exists an infinite firing sequence.
- The *boundedness problem* asks whether there exists $B \in \mathbb{N}$ such that for all $s \in Reach(\mathcal{N}, s_0)$ and for all $v \in V_s$, one has $\max(M_s(v)(p))_{p \in P} \leq B$.
- The *finiteness problem* asks whether the reachability set is finite, i.e., $Reach(\mathcal{N}, s_0) < \infty$.

In [71, 70] the authors establish the following results:

Theorem 7.3.1 ([71, 70]). *The reachability, finiteness and boundedness problems for RPN's are decidable*

Contrary to Petri nets, in [70] the authors show that model checking temporal logic becomes undecidable. More precisely, it is undecidable on RTL (Regular Temporal Logic) which is a fragment of LTL.

Theorem 7.3.2 ([70]). *Checking the truth of an RTL formula on an RPN is undecidable.*

In later works, the same authors present in [73] a submodel of RPN, the *sequential recursive Petri nets* (SRPN). Loosely speaking, the limitation imposed by the SRPN is such that the only thread that can fire is the latest created one. This gives the SRPN a limitation on its state, which can only be a branch (where in the RPN it can be any arbitrary rooted tree). This model preserves most of the modeling capabilities of RPN, while model checking LTL becomes decidable.

7.3.2 Languages

Recall that expressiveness of a formalism can be investigated by studying the family of languages that it can generate. In Section 2.3 we defined a few types of languages generated by a Petri net (e.g., the reachability language). Similarly, one can define the family of reachability languages for RPN. Given a marked RPN $(\mathcal{N}, s_0)$, a labelling of the transitions $\lambda(t) \in \Sigma \cup \{\varepsilon\}$ where Σ is a finite alphabet and ε is the empty word, and a finite subset of abstract states S_f, the reachability language $\mathcal{L}_R(\mathcal{N}, s_0, \lambda, S_f)$ is defined by:

$$\mathcal{L}_R(\mathcal{N}, s_0, \lambda, S_f) = \{\lambda(\sigma) \mid \exists\ s_0 \xrightarrow{\sigma} s_f \wedge [s_f] \in S_f\}$$

i.e., the set of labelling for sequences reaching some state of S_f in $\mathcal{N}$. Haddad et

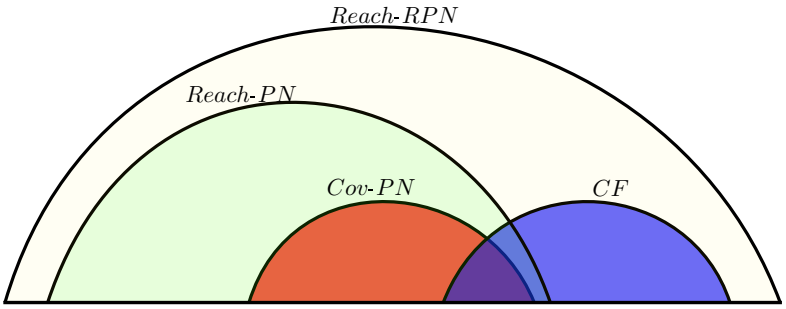


Figure 7.7: Hierarchy of languages.

al. [71] use the reachability languages in order to compare the RPN formalism to other well-known models.

Theorem 7.3.3 ([71]). *The family of reachability languages of RPNs strictly includes the union of the families of Petri net reachability and context-free languages.*

In [70] Haddad et al. show that the family of reachability languages of RPN is quite close to the family of recursively enumerable languages:

Theorem 7.3.4 ([70]). *Let $\mathcal{L}$ be a recursively enumerable language. Then there exists an RPN language $\mathcal{L}'$, a regular language $\mathcal{R}$ and a homomorphism h such that $\mathcal{L} = h(\mathcal{L}' \cap \mathcal{R})$.*

Combining the results above and some well-known results about Petri net languages we get a hierarchy on the languages described in Figure 7.7.

Chapter 8

Expressiveness and Decision Problems in RPN

8.1 Introduction

Till recently, the definition and analysis of the coverability problem for RPNs (similar to the one defined for Petri nets and many of its extensions) was left unexplored. One of the reasons, is the lack of a "good" order between states. One would try to find an order which is well quasi ordered, like for many other extensions of Petri nets. With such a wqo and using the WSTS framework, one would get the decidability of the coverability problem for free. Unfortunately, we show that under very light assumptions, all but the most trivial orders are not wqo. Instead, we present an order for the states of the RPN which while not being wqo is strongly compatible, similar to the one defined on the states of Petri nets.

Equipped with this order, we define and study the family of RPN coverability languages. We show that the RPN coverability languages are quite close to recursively enumerable languages, since the closure under homomorphism and intersection with a regular language is the family of recursively enumerable languages. We also show that RPN coverability (as reachability) languages strictly include the union of context-free languages and Petri net coverability languages. Moreover, we prove that RPN coverability languages and reachability languages of Petri nets are incomparable and that RPN coverability languages are a strict subclass of RPN reachability languages. Studying the operations on languages, we establish that the family of RPN coverability languages is closed under union, Kleene star, and homomorphism but not under intersection with a regular language nor under complementation.

From an algorithmic point of view, we show that, as for Petri nets, coverability, termination, boundedness, and finiteness problems are EXPSPACE-complete.

Thus, the increase of expressive power does not entail a corresponding increase in complexity. In order to solve the coverability problem, we show that if there exists a covering sequence, there exists a 'short' one (i.e., with a length at most doubly exponential w.r.t. the size of the input). In order to solve the termination problem, we consider two cases for an infinite sequence, depending on (informally speaking) whether the depth of the trees corresponding to states are bounded or not along the sequence. For the unbounded case, we introduce the abstract graph that expresses the ability to create threads from some initial state. The decidability of the finiteness and boundedness problems are also mainly based on this abstract graph.

Organization. In section 8.2, we introduce an order between RPNs states and establish basic results related to these notions. In section 8.3, we introduce decision problems and some reductions between them. In section 8.4, we study the expressiveness of coverability languages. Then in sections 8.5, 8.6, and 8.7 we show that the coverability, termination, boundedness, and finiteness problems are EXPSPACE-complete.

Based on. This chapter is mainly based on our works [59, 62].

8.2 An order for Recursive Petri Nets

We now define a qo $\preceq$ between states of an RPN. Given two states s, s' of an RPN $\mathcal{N}$, we say that s is *smaller or equal* than s', denoted by $s \preceq s'$, if there exists a subtree in s', which is isomorphic to s, where markings are greater or equal on all vertices and edges.

Definition 8.2.1. Let $s \neq \emptyset$ and s' be states of an RPN $\mathcal{N}$. Then $s \preceq s'$ if there exists an injective mapping f from V_s to $V_{s'}$ such that for all $v \in V_s$:

1. $M_s(v) \leq M_{s'}(f(v))$, and,
2. for all $v \xrightarrow{m}_s w$, there exists an edge $f(v) \xrightarrow{m'}_{s'} f(w)$ with $m \leq m'$.

In addition, $\emptyset \preceq s$ for all states s.
When $f(r_s)$ is required to be $r_{s'}$, one denotes this relation $s \preceq_r s'$ with $\emptyset \preceq_r s$ if and only if $s = \emptyset$.

Figure 8.1 illustrates these quasi-orders(shown below). We can see that $s \preceq s'$ since taking the injective mapping $f : V_s \mapsto V'_s$ defined by $f(r) = v$ and $f(u) = u'$, we get that both of the conditions are satisfied. On the other hand, $s \not\preceq_r s'$ since $M_s(r) \not\leq M_{s'}(r')$.

While this is irrelevant for the results presented here, let us mention that checking whether $s \preceq s'$ can be done in polynomial time by adapting a standard algorithm for the subtree problem (see for instance [146]).

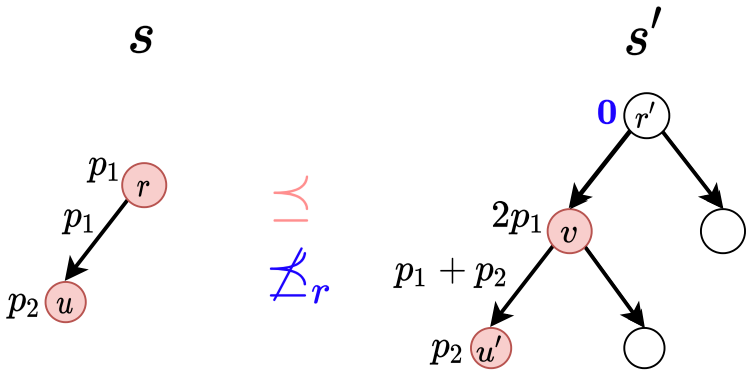


Figure 8.1: Illustration of the two quasi-orders defined in 8.2.1.

Lemma 8.2.2. *The relations $\preceq$ and $\preceq_r$ are quasi-orders.*

Proof. Let, s, s', s'' be states of an RPN $\mathcal{N}$ with $s = (r, m_0, \{(m_i, s_i)\}_{1 \leq i \leq k})$, $s' = (r', m'_0, \{(m'_i, s'_i)\}_{1 \leq i \leq k'})$ and $s'' = (r'', m''_0, \{(m''_i, s''_i)\}_{1 \leq i \leq k''})$. Let us show that the relation $\preceq$ is a qo.

1. Reflexivity: the identity function Id on V_s insures that $s \preceq s$.
2. Transitivity: Given $s \preceq s' \preceq s''$, there exist two injective functions $f : V_s \to V_{s'}$ and $f' : V_{s'} \to V_{s''}$. Let $g : V_s \to V_{s''}$ be defined by $g = f' \circ f$. Then g is injective. For any edge $v \xrightarrow{m}_s w$, there exists an edge $f(v) \xrightarrow{m'}_{s'} f(w)$ with $m \leq m'$ and there exists an edge $f'(f(v)) \xrightarrow{m''}_{s''} f'(f(w))$ with $m \leq m' \leq m''$. For all $v \in V_s$, one has $M_s(v) \leq M_{s'}(f(v)) \leq M_{s''}(f'(f(v))) = M_{s''}(g(v))$. Therefore $s \preceq s''$.

The proof for the relation $\preceq_r$ is similar. □

Consider the equivalence relation $\simeq := \preceq \cap \preceq^{-1}$. Given a set of states A, one denotes by $A/\simeq$ the quotient set by the equivalence relation $\simeq$. Observe that $s \simeq s'$ if and only if their abstract representations are equal and that $\simeq = \preceq_r \cap \preceq_r^{-1}$.

A qo $\leq$ on the states of an RPN is *strongly compatible* (as in [63]) if for all states s, s' such that $s \leq s'$ and for all transition firings $s \xrightarrow{(v,t)} s_1$, there exist a state s'_1 and a transition firing $s' \xrightarrow{(v',t')} s'_1$ with $s_1 \leq s'_1$.

Lemma 8.2.3. *The quasi-orders $\preceq$ and $\preceq_r$ are strongly compatible.*

Proof. Let $s \preceq s'$ and, let f be the mapping associated with the relation $\preceq$ and $s \xrightarrow{v,t} s_1$.
Thus, $s_v \xrightarrow{v,t} s_2$ for some s_2.

We will exhibit some s'_1 such that $s_1 \preceq s'_1$ with some f' as associated mapping. Since $M_s(v) \leq M_{s'}(f(v))$, one has $s'_{f(v)} \xrightarrow{f(v),t} s'_2$ for some s'_2 and by induction $s' \xrightarrow{f(v),t} s'_1$ for some s'_1.
It remains to define f'.

- If $t \in T_{el}$ then $f' = f$;
- If $t \in T_{ab}$ then for all threads u of s, $f'(u) = f(u)$ and if v^* (resp. w^*) is the thread created by the firing (v,t) (resp. $(f(v),t)$) then $f(v^*) = w^*$;
- If $t \in T_\tau$ then f' is equal to f restricted to the remaining vertices.

It is routine to check that the inequalities between corresponding markings of s and s' are fulfilled. The proof for $\preceq_r$ is similar. □

These quasi-orders may contain an infinite set of incomparable states (i.e., an infinite *antichain*). For example, see Figure 8.2 where any two states s_i and s_j are incomparable. Indeed, for any $i < j$: (1) $s_j \not\preceq s_i$ because $|V_{s_j}| > |V_{s_i}|$ there cannot

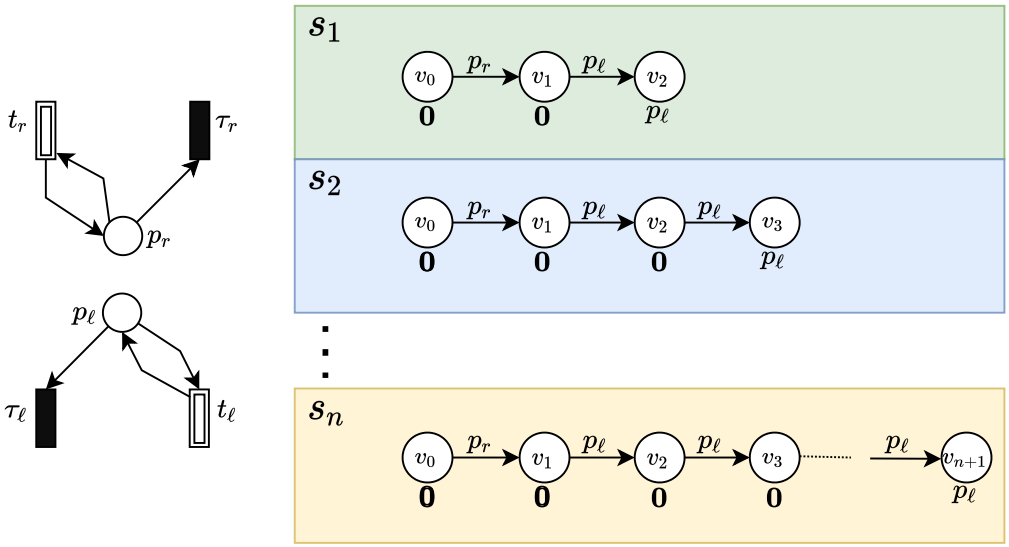


Figure 8.2: An RPN with an antichain of states

be any injective function from V_{s_j} to V_{s_i}, and (2) $s_i \not\preceq s_j$ because for any injective function from V_{s_i} to V_{s_j}, at least one of the edges with the marking p_r would be mapped to an edge with a marking p_ℓ. Since $s \preceq_r s'$ implies $s \preceq s'$, this is also an antichain for $\preceq_r$.

Observe also that these quasi-orders are not only strongly compatible. They are *transition-preserving compatible*, meaning that for all states s, s' such that $s \leq s'$ and for all transition firings $s \xrightarrow{(v,t)} s_1$, there exist s'_1 and a transition firing

$s' \xrightarrow{(v',t)} s'_1$ with $s_1 \leq s'_1$. In Petri net, the standard order on $\mathbb{N}^P$ is a wqo which is transition-preserving compatible. The next proposition establishes that such a qo does not exist in RPN.

Proposition 8.2.4. *There does not exist a wqo on states of RPN which is transition-preserving compatible.*

Proof. Consider the net of Figure 8.2 and the family of states $\{s_n\}_{n\geq 1}$. By a simple examination one gets that for all $n \geq 1$, $s_n \xrightarrow{(v_{n+1},\tau_\ell)\ldots(v_1,\tau_\ell)(v_0,\tau_r)} \emptyset$. Moreover, for all $n' \neq n$, there does not exist a firing sequence from $s_{n'}$ labelled by $\tau_\ell^{n+1}\tau_r$. Thus, for any transition-preserving compatible qo $\leq$, these states are incomparable, establishing that $\leq$ is not a wqo. □

Since $\preceq$ is not a wqo, RPNs with the relation $\preceq$ are not well-structured transition systems, see Section 2.1 or [63], for which coverability is decidable. Therefore, to solve coverability, one needs to find another way.

8.3 Decision problems and reductions

In this section, we introduce the decision problems that we are going to solve and establish reductions to simpler problems in order to shorten the proofs of subsequent sections.

Let $(\mathcal{N}, s_0)$ be a marked RPN and s_f be a state of $\mathcal{N}$.

- The *cut problem* asks whether there exists a firing sequence σ such that $s_0 \xrightarrow{\sigma} \emptyset$?
- The *coverability problem* asks whether there exists a firing sequence σ such that $s_0 \xrightarrow{\sigma} s \succeq s_f$?
- The *termination problem* asks whether there exists an infinite firing sequence?
- The *finiteness problem* asks whether $Reach(\mathcal{N}, s_0)$ is finite?
- The *boundedness problem* asks whether there exists $B \in \mathbb{N}$ such that for all $s \in Reach(\mathcal{N}, s_0)$ and for all $v \in V_s$, one has $\max(M_s(v)(p))_{p\in P} \leq B$?

Observe that contrary to Petri nets, the finiteness, and boundedness problems are different and not equivalent. Indeed, an RPN can be bounded while due to an unbounded number of vertices, its reachability set can be infinite.

For multiple reasons when dealing with marked RPN, it is more convenient to assume that the initial state is $s[r, m]$, i.e., contains a single thread. Therefore, we introduce the "rooted" version of the above problems, i.e., s_0 is required to be some $s[r, m_0]$, to which we show reductions from the general problems. In order to

establish these reductions, given a marked RPN $(\mathcal{N}, s_0)$, we build a marked RPN $(\mathring{\mathcal{N}}, s[r, \mathring{m}_0])$ that in a way simulates the former marked RPN. We do this by adding a place p_v for every vertex $v \neq r$ of s_0, and we add an abstract transition t_v that consumes a token from this place and creates a new vertex with initial marking in $M_{s_0}(v) + \sum_{v \xrightarrow{m_{v'}}_{s_0} v'} p_{v'}$. This will allow creating the children of v in s_0 (see Figure 8.3). In order to similarly proceed in the root, $\mathring{m}_0 = M_{s_0}(r) + \sum_{r \xrightarrow{m_v}_{s_0} v} p_v$.

Definition 8.3.1. Let $(\mathcal{N}, s_0)$ be a marked RPN. Then $(\mathring{\mathcal{N}}, \mathring{s_0})$ is defined by:

- $\mathring{P} = P \cup \{p_v \mid v \in V_{s_0} \setminus \{r_{s_0}\}\}$;
- $\mathring{T}_{ab} = T_{ab} \cup T_V$, $\mathring{T}_\tau = T_\tau$, $\mathring{T}_{el} = T_{el}$ with $T_V = \{t_v \mid v \in V_s \setminus \{r_s\}\}$;
- for all $t \in T$, one has $\mathring{W}^-(t) = W^-(t)$ and all $t \in T_{ab} \cup T_{el}$, $\mathring{W}^+(t) = W^+(t)$;
- for all $t_v \in T_V$ and $u \xrightarrow{m_v}_{s_0} v$, $\mathring{W}^-(t_v) = p_v$ and $\mathring{W}^+(t_v) = m_v$;
- for all $t \in T_{ab}$, $\mathring{\Omega}(t) = \Omega(t)$;
- for all $t_v \in T_V$, $\mathring{\Omega}(t_v) = M_{s_0}(v) + \sum_{v \xrightarrow{m_{v'}}_{s_0} v'} p_{v'}$;
- $\mathring{s_0} = s[r, M_{s_0}(r_s) + \sum_{r_{s_0} \xrightarrow{m_v}_{s_0} v} p_v]$.

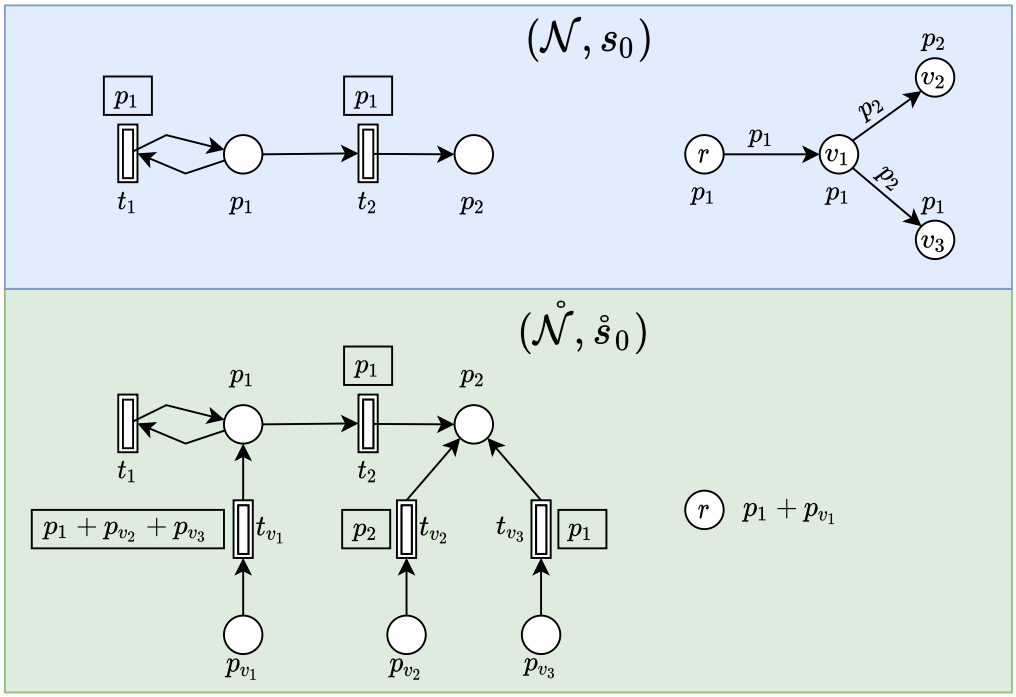


Figure 8.3: From a marked RPN to a rooted one

Let $m \in \mathbb{N}^{\mathring{P}}$, we denote by $m_{|P} \in \mathbb{N}^P$ the projection of m on P. Let s be a

state of $\mathring{N}$, we denote $s_{|P}$ a state of $\mathcal{N}$ obtained by projecting every marking of s on P.

Observations and notations.

1. The encoding size of $(\mathcal{N}, s_0)$ is linear w.r.t. the encoding size of $(\mathring{\mathcal{N}}, \mathring{s_0})$.
2. Let $e := (v_i)_{0 \leq i \leq k}$ be an enumeration of V_{s_0} such that $v_0 = r_{s_0}$ and for all $0 < i \leq k$, $prd(v_i) \in \{v_j\}_{j<i}$. Consider $\sigma^e_{s_0} = (prd(v_i), t_{v_i})_{i=1}^k$. Such an enumeration is called *consistent*. By construction of $\mathring{\mathcal{N}}$, $\mathring{s_0} \xrightarrow{\sigma^e_{s_0}}_{\mathring{\mathcal{N}}} s'_0$ with $s'_{0|P} = s_0$ and all places of P_V unmarked in s'_0.
3. Let $\mathring{s_0} \xrightarrow{\sigma}_{\mathring{\mathcal{N}}} s$, where 1) any abstract transition in σ different from $\{t_v\}$ does not create vertices in V_{s_0}, and 2) when t_v is fired it creates the new vertex is v. Then by construction, for all $v \in V_{s_0} \setminus \{r_{s_0}\}$, there is at most one occurrence of t_v which furthermore is fired in $prd_{s_0}(v)$. Moreover, since these firings consume tokens in P_V that were not used for firings of T, they can be pushed at the beginning of σ (denoted by σ_1) and completed by the missing firings of T_V in σ (denoted by σ_2) getting a consistent enumeration e. Summarizing, denoting by $\sigma_{|\mathcal{N}}$, the sequence σ without the firings of T_V, one gets that:
   (1) $\mathring{s_0} \xrightarrow{\sigma_1\sigma_{|\mathcal{N}}}_{\mathring{\mathcal{N}}} s$,
   (2) $\mathring{s_0} \xrightarrow{\sigma\sigma_2}_{\mathring{\mathcal{N}}} s'$ and
   (3) $s_0 \xrightarrow{\sigma_{|\mathcal{N}}}_{\mathcal{N}} s''$ with $s'_{|P} = s''$ and all places of P_V are unmarked in s'.

Due to observation 2, we immediately get that:

Lemma 8.3.2. *Let $(\mathcal{N}, s_0)$ be a marked RPN and $s_0 \xrightarrow{\sigma}_{\mathcal{N}} s$. Then for every consistent enumeration e, there exists a firing sequence $\mathring{s_0} \xrightarrow{\sigma^e_{s_0}\sigma}_{\mathring{\mathcal{N}}} s'$ with $s'_{|P} = s$ and all places of P_V are unmarked in s'.*

Due to observation 3, we immediately get that:

Lemma 8.3.3. *Let $(\mathcal{N}, s_0)$ be a marked RPN and $\mathring{s_0} \xrightarrow{\sigma}_{\mathring{\mathcal{N}}} s$. Then there exist a consistent enumeration e and a decomposition $\sigma^e_{s_0} = \sigma_1\sigma_2$ such that $\mathring{s_0} \xrightarrow{\sigma_1\sigma_{|\mathcal{N}}}_{\mathring{\mathcal{N}}} s$, $\mathring{s_0} \xrightarrow{\sigma\sigma_2}_{\mathring{\mathcal{N}}} s'$ and $s_0 \xrightarrow{\sigma_{|\mathcal{N}}}_{\mathcal{N}} s''$ with $s'_{|P} = s''$ and all places of P_V are unmarked in s'.*

Due to the previous lemmas, we get that:

Proposition 8.3.4. *The cut (resp. coverability, termination, finiteness, boundedness) problem is polynomially reducible to the rooted cut (resp. coverability, termination, finiteness, boundedness) problem.*

Proof. Let $(\mathcal{N}, s_0)$ be a marked RPN and s_f be a state of $\mathcal{N}$. Define $\mathring{s}_f$ a state of $\mathring{\mathcal{N}}$ be as s_f with in all markings of $\mathring{s}_f$, all places of $\mathring{P} \setminus P$ unmarked.

• Assume that there exists $s_0 \xrightarrow{\sigma}_{\mathcal{N}} \emptyset$. Then by Lemma 8.3.2, $\mathring{s}_0 \xrightarrow{\sigma^e_{s_0}\sigma}_{\mathring{\mathcal{N}}} \emptyset$. Assume that there exists $\mathring{s}_0 \xrightarrow{\sigma}_{\mathring{\mathcal{N}}} \emptyset$, which means that the last transition is fired in the root and is a cut transition. Then by Lemma 8.3.3, $s_0 \xrightarrow{\sigma_{|\mathcal{N}}}_{\mathcal{N}} s''$ for some s'', since the last firing of $\sigma_{|\mathcal{N}}$ is the cut transition fired in the root $s'' = \emptyset$.

• Assume that there exists $s_0 \xrightarrow{\sigma}_{\mathcal{N}} s \succeq s_f$. Then by Lemma 8.3.2, $\mathring{s}_0 \xrightarrow{\sigma^e_{s_0}\sigma}_{\mathring{\mathcal{N}}} \mathring{s}$ with $\mathring{s}_{|P} = s$. Thus $\mathring{s} \succeq \mathring{s}_f$. Assume that there exists $\mathring{s}_0 \xrightarrow{\sigma}_{\mathring{\mathcal{N}}} s \succeq \mathring{s}_f$. Then by Lemma 8.3.3, there exists σ_2 a firing sequence of T_V with $\mathring{s}_0 \xrightarrow{\sigma\sigma_2}_{\mathring{\mathcal{N}}} s'$, $s_0 \xrightarrow{\sigma_{|\mathcal{N}}}_{\mathcal{N}} s''$ and $s'_{|P} = s''$. Since σ_2 only creates vertices and deletes tokens from P_V, $s' \succeq \mathring{s}_f$. Thus $s'' \succeq s_f$.

• Assume that there exists $s_0 \xrightarrow{\sigma}_{\mathcal{N}}$ with σ infinite. Then by Lemma 8.3.2, $\mathring{s}_0 \xrightarrow{\sigma^e_{s_0}\sigma}_{\mathring{\mathcal{N}}}$. Assume that there exists $\mathring{s}_0 \xrightarrow{\sigma}_{\mathring{\mathcal{N}}}$ with σ infinite. Then by Lemma 8.3.3, $s_0 \xrightarrow{\sigma_{|\mathcal{N}}}_{\mathcal{N}}$ with $\sigma_{|\mathcal{N}}$ infinite since there are only a finite number of firings of T_V.

• Assume that $Reach(\mathcal{N}, s_0)$ is infinite. For all $s \in Reach(\mathcal{N}, s_0)$, define $\mathring{s}$ a state of $\mathring{\mathcal{N}}$ as s with all places of P_V in markings of s unmarked. Due to Lemma 8.3.2, $\mathring{s} \in Reach(\mathring{\mathcal{N}}, \mathring{s}_0)$. Since this mapping is injective, $Reach(\mathring{\mathcal{N}}, \mathring{s}_0)$ is infinite. Assume that $Reach(\mathring{\mathcal{N}}, \mathring{s}_0)$ is infinite. Let $s \in Reach(\mathring{\mathcal{N}}, \mathring{s}_0)$. Due to Lemma 8.3.3, consider $s \xrightarrow{\sigma_2}_{\mathring{\mathcal{N}}} s'$ and $s_0 \xrightarrow{\sigma_{|\mathcal{N}}}_{\mathcal{N}} s''$ with $s'_{|P} = s''$ and all places of P_V unmarked in s'. Thus $s'' \in Reach(\mathcal{N}, s_0)$. The mapping from s to s'' is not injective. However, the inverse image of s'' by this mapping is finite since there are a finite number of consistent enumerations and prefixes of such enumerations. Thus $Reach(\mathcal{N}, s_0)$ is infinite.

• Assume that $(\mathcal{N}, s_0)$ is unbounded. For all $s \in Reach(\mathcal{N}, s_0)$, define $\mathring{s}$ a state of $\mathring{\mathcal{N}}$ as s with all places of P_V in markings of s unmarked. Due to Lemma 8.3.2, $\mathring{s} \in Reach(\mathring{\mathcal{N}}, \mathring{s}_0)$. Thus $(\mathring{\mathcal{N}}, \mathring{s}_0)$ is unbounded. Assume that $(\mathring{\mathcal{N}}, \mathring{s}_0)$ is unbounded. By construction, the marking of places in P_V is bounded. Let $s \in Reach(\mathring{\mathcal{N}}, \mathring{s}_0)$. Due to Lemma 8.3.3, consider $s \xrightarrow{\sigma_2}_{\mathring{\mathcal{N}}} s'$ and $s_0 \xrightarrow{\sigma_{|\mathcal{N}}}_{\mathcal{N}} s''$ with $s'_{|P} = s''$ and all places of P_V unmarked in s'. Thus $s'' \in Reach(\mathcal{N}, s_0)$. Since for all vertex v of s, v is also present in s'' and for all $p \in P$, $M_s(v)(p) = M_{s''}(v)(p)$. Then $Reach(\mathcal{N}, s_0)$ is unbounded. □

Let σ be a firing sequence. A thread is *extremal* w.r.t. σ if it is an initial or final thread.

Definition 8.3.5. Let $\mathcal{N}$ be an RPN. Then $T_{ret} \subseteq T_{ab}$, the set of *returning transitions* is defined by:

$$\{t \in T_{ab} \mid \exists \sigma \text{ s.t. } s[r, \Omega(t)] \xrightarrow{\sigma} \emptyset\}$$

For all $t \in T_{ret}$, we define σ_t to be some arbitrary shortest *returning sequence* (i.e., $s[r, \Omega(t)] \xrightarrow{\sigma_t} \emptyset$). We now introduce $\widehat{\mathcal{N}}$, obtained from $\mathcal{N}$ by adding elementary

transitions that mimic the behavior of a returning sequence. Observe that the size of $\widehat{\mathcal{N}}$ is linear w.r.t. the size of $\mathcal{N}$.

Definition 8.3.6. Let $\mathcal{N}$ be an RPN. Then $\widehat{\mathcal{N}} = \left\langle P, \widehat{T}, \widehat{W^+}, \widehat{W^-}, \Omega \right\rangle$ is defined by:

- $\widehat{T}_{ab} = T_{ab}$, $\widehat{T}_\tau = T_\tau$, $\widehat{T}_{el} = T_{el} \uplus \{t^r \mid t \in T_{ret}\}$;
- for all $t \in T$, $\widehat{W^-}(t) = W^-(t)$ and all $t \in T_{ab} \cup T_{el}$, $\widehat{W^+}(t) = W^+(t)$;
- for all $t \in T_{ab}$, $\widehat{\Omega}(t) = \Omega(t)$;
- for all $t \in T_{ret}$, $\widehat{W^-}(t^r) = W^-(t)$ and $\widehat{W^+}(t^r) = W^+(t)$.

Figure 8.4 illustrates this construction.

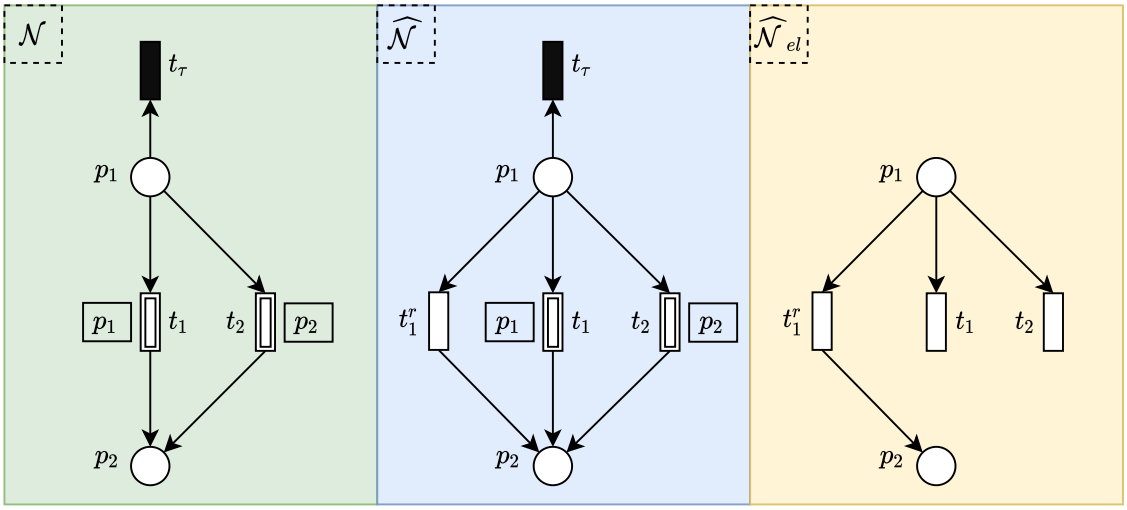


Figure 8.4: From $\mathcal{N}$ to $\widehat{\mathcal{N}}$ and $\widehat{\mathcal{N}}_{el}$

Note that since $\widehat{\mathcal{N}}$ enlarges $\mathcal{N}$ by adding transitions and that any firing of t^r in $\widehat{\mathcal{N}}$ can be replaced by the firing of $t\sigma_t$ in $\mathcal{N}$, we get:

Proposition 8.3.7. *Let $(\mathcal{N}, s_0)$ be a marked RPN. Then $Reach(\mathcal{N}, s_0) = Reach(\widehat{\mathcal{N}}, s_0)$.*

We call a firing sequence σ *omniscient* if any thread created during its firing is a final thread.

Proposition 8.3.8. *Let $(\mathcal{N}, s_0)$ be a marked RPN and $s_0 \xrightarrow{\sigma}_{\mathcal{N}} s$. Then there exists a firing sequence $s_0 \xrightarrow{\widehat{\sigma}}_{\widehat{\mathcal{N}}} s$ such that $\widehat{\sigma}$ is omniscient.*

Proof. Assume that we have an extremal thread u which fires $t \in T_{ab}$, creating a non-final thread v that disappears by a matching cut transition $(v, t_\tau) \in \sigma$ for

$t_\tau \in T_\tau$. One builds σ' by (1) deleting from σ the transition (u,t), (2) deleting all the firings from $Des_\sigma(v)$ in σ, and (3) replacing the transition (v,t_τ) by (u,t^r). We claim that $s \xrightarrow{\sigma} s'$. Indeed, in u the transition (u,t^r) has the same incidence in u as the transition (u,t) followed by (v,t_τ) ('anticipating' (v,t_τ) only add tokens in intermediate states) and the other deleted firings are performed by threads in $Des_\sigma(v)$ which do not exist anymore. By taking $\widehat{\sigma}$ the sequence obtained by iterating the process, we get the omniscient sequence. □

In order to recover from a sequence in $\widehat{\mathcal{N}}$ a sequence in $\mathcal{N}$, for every $t \in T_{ret}$, one has to simulate the firings of a transition t^r by sequence σ_t. Therefore, bounding the length of σ_t is a critical issue. Recall that in [132], Rackoff showed that the coverability problem for Petri nets belongs to EXPSPACE. More precisely, he proved that if there exists a covering sequence, then there exists a 'short' one:

Theorem 8.3.9 (Rackoff [132]). *Let $\mathcal{N}$ be a Petri net, m_{ini}, m_{tar} be markings and σ be a firing sequence such that $m_{ini} \xrightarrow{\sigma} m \geq m_{tar}$. Then there exists a sequence σ' such that $m_{ini} \xrightarrow{\sigma'} m' \geq m_{tar}$ with $|\sigma'| \leq 2^{2^{cn \log n}}$ for some constant c and n being the size of $(\mathcal{N}, m_{tar})$.*

A surprising consequence of Rackoff's proof is that the length of the minimal coverability sequence does not depend on the initial marking of the net. Using this bound, we show a similar one for the length of the returning sequences.

Proposition 8.3.10. *Let $\mathcal{N}$ be an RPN and $t \in T_{ret}$. Then the returning sequence σ_t fulfills $|\sigma_t| \leq 2^{\cdot 2^{dn \log n}}$ for some constant d and $n = size(\mathcal{N})$.*

Proof. Let us enumerate $T_{ret} = \{t_1, \ldots, t_K\}$ in such a way that $i < j$ implies $|\sigma_{t_i}| \leq |\sigma_{t_j}|$. Observe first that the shortest returning sequences do not include firings of abstract transitions not followed by a matching cut transition, since it could be omitted, as it only deletes tokens in the thread. We argue by induction on $k \leq K$ that:

$$|\sigma_{t_k}| < 2^{k \cdot 2^{cn \log n}} \qquad \text{where } c \text{ is the Rackoff constant}$$

For $k = 1$, we know that σ_{t_1} has a minimal length over all returning sequences. Hence, there are no cuts in σ_{t_1} except the last one. Due to the above observation, σ_{t_1} only includes firing of elementary transitions. Thus, the Rackoff bound of Theorem 8.3.9 applies for a covering of some final marking.
Assume that the result holds for all $i < k$. Due to the requirement on lengths, σ_{t_k} only includes cuts from threads created by $t_i \in T_{ret}$ with $i < k$. Thus, by Proposition 8.3.8 we get a sequence $\widehat{\sigma}_{t_k} \cdot (r, t_\tau)$ in $\widehat{\mathcal{N}}$ (where r is the root and $t_\tau \in T_\tau$). The sequence $\widehat{\sigma}_{t_k}$ consists of only elementary transitions and does not contain any transition t_i^r with $i \geq k$. The marking of r reached by $\widehat{\sigma}_{t_k}$ covers

some final marking, hence by Theorem 8.3.9 there exists a covering sequence $\widehat{\sigma}'_{t_k}$ such that $|\widehat{\sigma}'_{t_k}| \leq 2^{2^{cn \log n}}$. Since $\widehat{\sigma}_{t_k}$ does not contain firing of t_i^r with $i \geq k$ this also holds for $\widehat{\sigma}'_{t_k}$. Substituting any firing of t_i^r by σ_{t_i}, one gets a corresponding sequence σ'_{t_k} in $\mathcal{N}$. Using the induction hypothesis, one gets that the length of σ'_{t_k} fulfills:

$$|\sigma'_{t_k}| \leq |\widehat{\sigma}_{t'_k}| 2^{(k-1)\cdot 2^{cn \log n}} \leq 2^{2^{cn \log n}} \cdot 2^{(k-1)\cdot 2^{cn \log n}} = 2^{k \cdot 2^{cn \log n}}$$

From minimality of σ_{t_k}, one gets $|\sigma_{t_k}| \leq |\sigma'_{t_k}| \leq 2^{k \cdot 2^{cn \log n}}$ which concludes the proof since

$$\max_{t \in T_{ret}} \{|\sigma_t|\} \leq 2^{|T_{ret}| \cdot 2^{cn \log n}} \leq 2^{n 2^{cn \log n}} \leq 2^{2^{2cn \log n}}.$$

□

Using the previous proposition, we can compute T_{ret} in exponential space, by enumerating for all abstract transitions, all firing sequences of sufficient length and checking whether they lead to the empty tree.

Below are immediate corollaries from the previous propositions:

Corollary 8.3.11. *Let $\mathcal{N}$ be a marked RPN. Then for all $s \xrightarrow{\widehat{\sigma}}_{\widehat{\mathcal{N}}} s'$, there exists $s \xrightarrow{\sigma}_{\mathcal{N}} s'$ such that $|\sigma| \leq 2^{2^{dn \log n}} |\widehat{\sigma}|$ for some constant d and $n = size(\mathcal{N})$.*

Corollary 8.3.12. *Given an RPN $\mathcal{N}$, one can build $\widehat{\mathcal{N}}$ in exponential space.*

In order to mimic the behavior of a specific thread in a firing sequence (which will be useful later on), we introduce the Petri net $\widehat{\mathcal{N}}_{el}$. The size of $\widehat{\mathcal{N}}_{el}$ is also linear w.r.t. the size of $\mathcal{N}$.

Definition 8.3.13. Let $\mathcal{N}$ be an RPN. Then the Petri net $\widehat{\mathcal{N}}_{el} = \left\langle P, \widehat{T}_{el}, \widehat{W}^+_{el}, \widehat{W}^-_{el} \right\rangle$ is defined by:

- $\widehat{T}_{el} = \widehat{T} \setminus T_\tau$;
- For all $t \in \widehat{T}_{el} \setminus T_{ab}$, $\widehat{W}^-_{el}(t) = \widehat{W}^-(t)$ and $\widehat{W}^+_{el}(t) = \widehat{W}^+(t)$;
- For all $t \in T_{ab}$, $\widehat{W}^-_{el}(t) = \widehat{W}^-(t)$ and $\widehat{W}^+_{el}(t) = 0$.

Figure 8.4 illustrates this definition.

As for $\widehat{\mathcal{N}}$, one can build $\widehat{\mathcal{N}}_{el}$ in exponential space.

Observation. The main (straightforward) property of $\widehat{\mathcal{N}}_{el}$ is the following one. Let $\sigma \in \widehat{T}^*_{el}$ with n_t the number of occurrences of t in σ. Then $m_0 \xrightarrow{\sigma}_{\widehat{\mathcal{N}}_{el}} m$ if and only if $s[r, m_0] \xrightarrow{(r,\sigma)}_{\widehat{\mathcal{N}}} s$ with $V_s = \{r\} \cup \bigcup_{t \in T_{ab}} \{v_{t,1}, \ldots, v_{t,n_t}\}$, $M_s(r) = m$ and for all v_{t_i}, $r \xrightarrow{W^+(t)}_s v_{t_i}$ and $M_s(v_{t_i}) = \Omega(t)$.

8.4 Expressiveness

The expressiveness of a formalism may be defined by the family of languages that it can generate. In [71], the expressiveness of RPNs was studied using reachability languages. However, using reachability languages as specification languages has an inconvenient since the emptiness problem for these languages is Ackermannian-complete [104, 41] for Petri nets, so it is also Ackermannian-Hard, at least, for RPN. We propose to characterize the expressive power of RPN by studying the family of coverability languages which is sufficient to express most of the usual reachability properties since many of them reduce to check that no reachable state may cover a bad marking in a thread.

As for reachability language, we equip any transition t with a *label* $\lambda(t) \in \Sigma \cup \{\varepsilon\}$ where Σ is a finite alphabet and ε is the empty word. The labelling is extended to transition sequences in the usual way. Thus, given a labelled marked RPN $(\mathcal{N}, s_0)$ and a finite subset of states S_f, the (coverability) language $\mathcal{L}_C(\mathcal{N}, s_0, \lambda, S_f)$ is defined by:

$$\mathcal{L}_C(\mathcal{N}, s_0, \lambda, S_f) = \{\lambda(\sigma) \mid \exists\ s_0 \xrightarrow{\sigma} s \succeq s_f \wedge s_f \in S_f\}$$

i.e., the set of words generated by sequences covering some state of S_f in $\mathcal{N}$.

8.4.1 Cut and cover languages equivalence

In this subsection, we show that the family of coverability languages of an RPN is a particular family of reachability languages of an RPN : the family of *cut languages*. A cut language of an RPN is a reachability language with a single final state $\emptyset$.

Proposition 8.4.1. *The family of cut languages of RPNs is included in the family of coverability languages of RPNs.*

Proof. Due to the correspondence between firing sequences of $(\mathcal{N}, s_0)$ and those of $(\mathring{\mathcal{N}}, \mathring{s_0})$, established in the previous section, one can assume w.l.o.g. that the initial markings of the RPNs have a single vertex. Let $\mathcal{L}_R(\mathcal{N}, s[r, m_0], \lambda, \{\emptyset\})$ be such a reachability language.
$\mathcal{N}'$ is obtained by adding places *todo* and *done* and a transition $start \in T'_{ab}$ with:
$\lambda'(start) = \varepsilon$, $W'^{-}(start) = todo$, $W'^{+}(start) = done$, $\Omega'(start) = m_0$.
Then it is routine to check that $\mathcal{L}_C(\mathcal{N}', s[r, todo], \lambda', \{s[r, done]\}) = \mathcal{L}_R(\mathcal{N}, s[r, m_0], \lambda, \{\emptyset\})$.
□

Establishing the converse inclusion is more intricate.

Proposition 8.4.2. *The family of coverability languages of RPNs is included in the family of cut languages of RPNs.*

Proof. Due to the correspondence between firing sequences of $(\mathcal{N}, s_0)$ and those of $(\mathring{\mathcal{N}}, \mathring{s_0})$, established in the previous section, one can assume w.l.o.g. that the initial markings of the RPNs have a single vertex. Let $\mathcal{L}_C(\mathcal{N}, s[r, m_0], \lambda, S_f)$ be a coverability RPN language.

Case $\emptyset \in S_f$. Observe that in this case we can reduce S_f to $\{\emptyset\}$. Then $\mathcal{N}'$ is obtained from $\mathcal{N}$ by adding a place $root$ and a cut transition t_{root} with $\lambda'(t_{root}) = \varepsilon$ and $W'^{-}(t_{root}) = root$. It is routine to check that the reachability language $\mathcal{L}_R(\mathcal{N}', s[r, m_0 + root], \lambda', \{\emptyset\}) = \mathcal{L}_C(\mathcal{N}, s[r, m_0], \lambda, \{\emptyset\})$.

Case $\emptyset \notin S_f$. Consider the net $\mathcal{N}^*$ obtained from $\mathcal{N}$ by adding two places $start$ and run with $m_0^* = start$, transitions $t_{run} \in T_{el}$ and $t_{start} \in T_{ab}$ with $\lambda^*(t_{run}) = \lambda^*(t_{start}) = \varepsilon$ and:
$W^{*-}(t_{run}) = run$, $W^{*+}(t_{run}) = 2run$,
$W^{*-}(t_{start}) = start$, $W^{*+}(t_{start}) = \mathbf{0}$ and $\Omega^*(t_{start}) = m_0 + run$.

- For all $t \in T_{el}$, $W^{*-}(t) = W^{*-}(t) + run$ and $W^{*+}(t) = W^+(t)$;
- For all $t \in T_{ab}$, $\Omega^*(t) = \Omega(t) + run$, $W^{*-}(t) = W^-(t) + run$ and $W^{*+}(t) = W^+(t) + run$;
- For all $t \in T_\tau$, $W^{*-}(t) = W^-(t) + run$.

Let S_f^* be S_f where all markings are increased by run.
Then it is routine to check that: $\mathcal{L}_C(\mathcal{N}^*, s[r, m_0^*], \lambda^*, S_f^*) = \mathcal{L}_C(\mathcal{N}, s[r, m_0], \lambda, S_f)$. Furthermore, (1) the empty tree is not reachable in $(\mathcal{N}^*, s[r, m_0^*])$ and (2) for any coverability sequence $s[r, m_0^*] \xrightarrow{\sigma} s \succeq s_f \in S_f^*$, r does not belong to the image of the corresponding mapping f. Thus, in the rest of the proof, we assume that $(\mathcal{N}, s[r, m_0], \lambda^*, S_f)$ fulfills these properties. We also assume w.l.o.g. that all vertices in S_f are distinct. We denote V_f this set of vertices.

Let $\mathcal{N}'$ obtained as follows.
One adds places $todo, done, cut, \{p_v \mid v \in V_f\}, \{p_{u,v} \mid s \in S_f, u \xrightarrow{m_v}_s v\}$.

- For all $t \in T_{el}$, $W'^{-}(t) = W^-(t)$ and $W'^{+}(t) = W^+(t)$;
- For all $t \in T_{ab}$, $W'^{-}(t) = W^-(t)$, $W'^{+}(t) = W^+(t)$ and $\Omega'(t) = \Omega(t) + cut$;
- For all $t \in T_\tau$, $W'^{-}(t) = W^-(t) + cut$.

For all $t \in T_{ab}$, one adds the following abstract transitions:

- one adds $t_{Br} \in T'_{ab}$ with $\lambda'(t_{Br}) = \lambda(t)$ and $W'^{-}(t_{Br}) = W^-(t) + todo$, $W'^{+}(t_{Br}) = done$, $\Omega'(t_{Br}) = \Omega(t) + todo$;
- For all r_s with $s \in S_f$ one adds $t_{r_s} \in T'_{ab}$ with $\lambda'(t_{r_s}) = \lambda(t)$ and $W'^{-}(t_{r_s}) = W^-(t) + todo$, $W'^{+}(t_{r_s}) = done$, $\Omega'(t_{r_s}) = \Omega(t) + (|\{r_s \xrightarrow{m_w}_s w\}| + 1)p_{r_s}$;
- For all $v \in V_s \setminus \{r_s\}$ with $s \in S_f$ and $u \xrightarrow{m_v}_s v$ such that $W^+(t) \geq m_v$, one adds $t_v \in T'_{ab}$ with $\lambda'(t_v) = \lambda(t)$ and $W'^{-}(t_v) = W^-(t) + p_u$, $W'^{+}(t_v) = p_{u,v}$, $\Omega(t_v) = \Omega(t) + (|\{v \xrightarrow{m_w}_s w\}| + 1)p_v$.

One adds the following cut transitions:

- One adds $\tau_{done} \in T_\tau$ with $W'^{-}(\tau_{done}) = done$ and $\lambda'(\tau_{done}) = \varepsilon$.

- For all $v \in V_s$ with $s \in S_f$, one adds $\tau_v \in T'_\tau$ with $\lambda'(\tau_v) = \varepsilon$ and $W'^-(\tau_v) = M_s(v) + p_v + \sum_{v \xrightarrow{m_w}_s w} p_{v,w}$.

Let us prove that $\mathcal{L}_R(\mathcal{N}', s[r, m_0 + todo], \{\emptyset\}) = \mathcal{L}_C(\mathcal{N}, s[r, m_0], S_f)$.

• $\mathcal{L}_C(\mathcal{N}, s[r, m_0], \lambda, S_f) \subseteq \mathcal{L}_R(\mathcal{N}', s[r, m_0 + todo], \lambda', \{\emptyset\})$. Consider in $\mathcal{N}$ a coverability sequence $s[r, m_0] \xrightarrow{\sigma} s \succeq s_f \in S_f$ with f the mapping from V_{s_f} to V_s. Let Br be the branch in s from r to $f(r_{s_f})$, excluding $f(r_{s_f})$. We build a sequence σ' as follows.

- Let $v \in Br \setminus \{r\}$ and (u, t) be the firing in σ that creates v. Then we substitute (u, t) by (u, t_{Br});
- Let (u, t) be the firing in σ that creates $f(r_{s_f})$. Then we substitute (u, t) by $(u, t_{r_{s_f}})$;
- Let $v \in V_{s_f} \setminus \{r_{s_f}\}$ and (u, t) be the firing in σ that creates $f(v)$. Then we substitute (u, t) by (u, t_v).

Then σ' is a firing sequence of $(\mathcal{N}', s[r, m_0 + todo])$ that leads to s' with the same tree structure (and vertices) as the one of s and where the markings labelling s' are defined as follows.

- For all $v \in V_{s'} \setminus (Br \cup f(V_{s_f}))$, $M_{s'}(v) = M_s(v) + cut$, and all $u \xrightarrow{m'_v}_{s'} v$ and $u \xrightarrow{m_v}_s v$, one has $m'_v = m_v$;
- For all $v \in Br$, $M_{s'}(v) = M_s(v)$. For all $v \xrightarrow{m'_w}_{s'} w$ with $w \in Br \cup \{f(r_{s_f})\}$, $m'_w = done$;
- For all $v \in V_{s_f}$, $M_{s'}(f(v)) = M_s(f(v)) + p_v$. For all $f(v) \xrightarrow{m'_w}_{s'} f(w)$, $m'_w = p_{v,w}$.

Observe that $\lambda(\sigma') = \lambda(\sigma)$. Then one completes σ' by firing $\{(f(v), \tau_v)\}_{v \in V_{s_f}}$ bottom up followed by firing $\{(v, \tau_{done})\}_{v \in Br}$ bottom up leading to $\emptyset$.

• $\mathcal{L}_R(\mathcal{N}', s[r, m_0 + todo], \lambda', \{\emptyset\}) \subseteq \mathcal{L}_C(\mathcal{N}, s[r, m_0], \lambda, S_f)$. Observe that in $(\mathcal{N}', s[r, m_0 + todo])$ the only way to reach $\emptyset$ is to fire, τ_{done} since in r (by induction) only abstract transitions of T_{ab}, $\{t_{Br} \mid t \in T_{ab}\}$ and $\{t_{r_s} \mid t \in T_{ab} \wedge s \in S_f\}$ are fireable and places cut and $\{p_v\}_{v \in V_f}$ are initially unmarked. Furthermore, a single firing $\{t_{Br} \mid t \in T_{ab}\}$ and $\{t_{r_s} \mid t \in T_{ab} \wedge s \in S_f\}$ is at most possible in r, since no transition can produce tokens for $todo$ in r.

So consider in $\mathcal{N}'$ a firing sequence $s[r, m_0 + todo] \xrightarrow{\sigma'} \emptyset$. Due to the previous observation before the firing (r, τ_{done}) ending σ', there has been in σ' a firing of (r, t_{Br}) or (r, t_{r_s}) for some $t \in T_{ab}$ and $s \in S_f$ creating a vertex v_1 followed by the firing of a cut transition in v_1. Since $\Omega'(t_{Br}) = \Omega(t) + todo$, if v_1 has been created by (r, t_{Br}), then the only cut transition that can be fired in v_1 is τ_{done}. Since $\lambda'(\tau_{done}) = \varepsilon$ and $W'^+(t_{Br}) = done$, this firing can be delayed in σ' just before the firing of (r, τ_{done}).

Furthermore, there must have been before this firing, the firing of (v_1, t_{Br}) or

(v_2, t_{r_s}) for some $t \in T_{ab}$ and $s \in S_f$ creating a vertex v_2 followed by the firing of a cut transition in v_2. Since this iterated reasoning must end, there must be some v_k created by the firing of (v_{k-1}, t_{r_s}) (with $v_0 = r$) for some $t \in T_{ab}$ and $s \in S_f$. We denote by $f(r_s)$ the vertex v_k.
Since $\Omega'(t_{r_s}) = \Omega(t) + (|\{r_s \xrightarrow{m_w}_s w\}| + 1)p_{r_s}$, the only cut transition that can be fired in $f(r_s)$ is τ_{r_s}. Since $\lambda'(\tau_{r_s}) = \varepsilon$ and $W'^{+}(t_{r_s}) = \mathit{done}$, this firing can be delayed in σ' just before the firing of $(v_{k-1}, \tau_{\mathit{done}})$. Furthermore, the firing of this cut transition must have been preceded for all $r_s \xrightarrow{m_w}_s w$ by the firing of some abstract transition (v_k, t_w) creating a vertex denoted $f(w)$, followed by the firing of a cut transition in $f(w)$.
Applying the same reasoning for $f(w)$ as the one for $f(r_s)$, one gets that the only cut transition that can be fired in $f(w)$ is τ_w and that all the firings related to these w's can be delayed before the firing $(f(r_s), \tau_r s)$.
Iterating this process, one obtains that σ' can be reordered as $\sigma''\sigma_\tau$ with $\lambda'(\sigma'') = \lambda'(\sigma')$, and σ_τ is a sequence of cut transition firings with $\lambda(\sigma_\tau) = \varepsilon$.
Let s'' be the state of reached by σ'': it includes a branch created by the firings among $\{t_{Br}\}_{t \in T_{ab}}$, followed by a tree whose set of vertices is $f(V_s)$ and every vertex $f(v)$ has been created by the firing of some transition in $\{t_v\}_{t \in T_{ab}}$. Observe that due to our observations on $(\mathcal{N}', s[r, m_0 + \mathit{todo}])$, all other firings of σ'' are firings of transitions in T. By substituting in σ'' all t_{Br} by t and all t_v by t, one gets a firing sequence σ of $(\mathcal{N}, s[r, m_0])$ with $\lambda(\sigma) = \lambda'(\sigma')$ that covers s. □

The transformation presented in the above proof can be performed in polynomial time, and this will be used in the next section.

8.4.2 Language hierarchy

We now turn to investigating the relationship between the family of RPN coverability languages to other language families. More precisely, we consider the families of recursively enumerable languages, context-free languages, RPN reachability languages, and Petri nets languages.

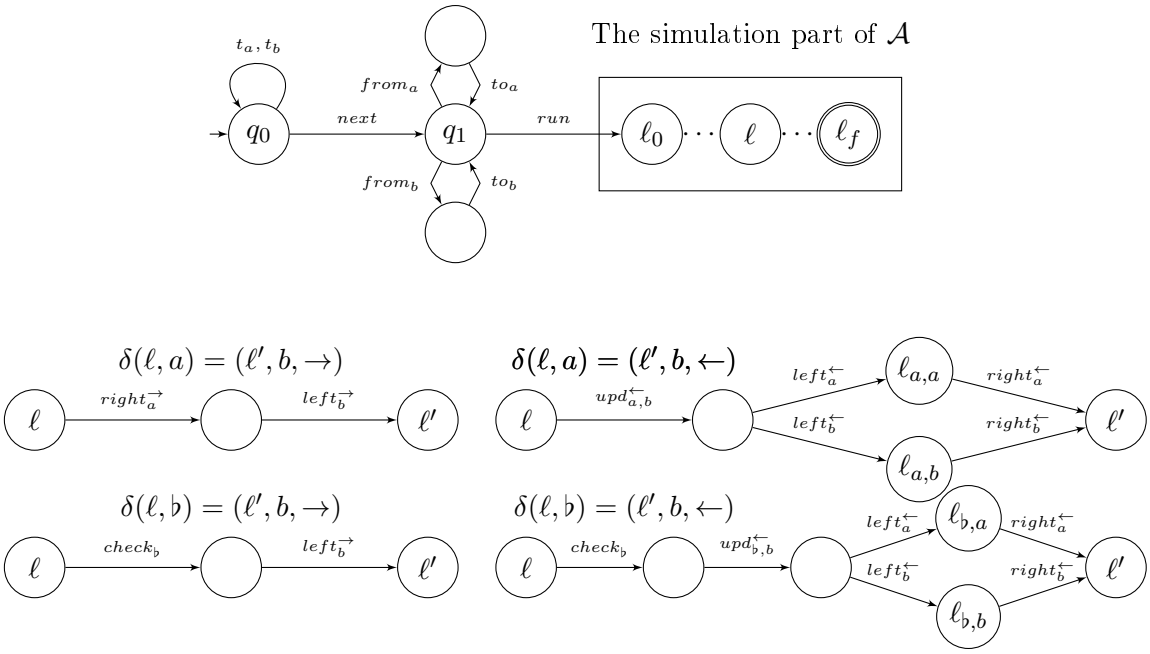
The next theorem has two interesting consequences: the family of RPN coverability languages is not closed under intersection with the family of regular languages. But the family obtained by this intersection is *quite close* to the family of recursively enumerable languages. The result was already stated in Proposition 9 of [70] for the family of RPN reachability languages, but the proof was only sketched.

Theorem 8.4.3. *Let $\mathcal{L}$ be a recursively enumerable language. Then there exist an RPN coverability language $\mathcal{L}'$, a regular language $\mathcal{R}$ and a homomorphism h such that $\mathcal{L} = h(\mathcal{L}' \cap \mathcal{R})$.*

Proof. Let $\mathcal{M} = (\Sigma, L, \delta)$ be a Turing machine with it set of states L including ℓ_0 (resp. ℓ_f) the initial (resp. final) state and its transition function δ from $L \times \Sigma \cup \{\flat\}$ to $L \times \Sigma \times \{\leftarrow, \rightarrow\}$ where $\flat$ is the blank character.

Let us define a labeled marked RPN $\mathcal{N}$ and an automaton $\mathcal{A}$. Their common alphabet is the set of transitions of $\mathcal{N}$, and the labeling of the transitions of the RPN is the identity mapping. The intersection of their languages is thus the language of the synchronized product of the two devices. The single final state of $\mathcal{N}$ (to be covered) is the empty tree.

The automaton $\mathcal{A}$ is depicted below (with $\Sigma = \{a, b\}$). In q_0 it allows $\mathcal{N}$ to generate the representation of any word $w \in \Sigma^*$, input of $\mathcal{M}$. However, this intermediate representation is not suitable for mimicking $\mathcal{M}$. Thus, in q_1, the intermediate representation is translated into an appropriate one. Once this representation is obtained, it mimics any transition of $\mathcal{M}$ by triggering the firing of several transitions of $\mathcal{N}$. We will detail this simulation after the specification of $\mathcal{N}$.



$\mathcal{N}$ is defined as follows. Its set of places is $P = \{p_a \mid a \in \Sigma\} \cup \{root, right, left, start, ret\}$. We now define the set of transitions T. The first subset corresponds to the generation of a representation of the input word of $\mathcal{M}$.

- For all $a \in \Sigma$, $t_a \in T_{ab}$ with $W^-(t_a) = start$, $W^+(t_a) = ret$ and $\Omega(t_a) = start + p_a$;
- $next \in T_{el}$ with $W^-(next) = start$ and $W^+(next) = ret$;
- For all $a \in \Sigma$, $from_a \in T_\tau$ $W^-(from_a) = ret + p_a$;

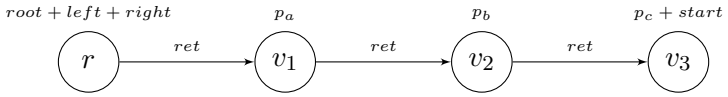
- For all $a \in \Sigma$, $to_a \in T_{ab}$ with $W^-(to_a) = right$, $W^+(to_a) = right$ and $\Omega(to_a) = right + p_a$;
- $run \in T_{el}$ with $W^-(run) = root + ret$ and $W^+(run) = root$

The second subset corresponds to the simulation of $\mathcal{M}$.

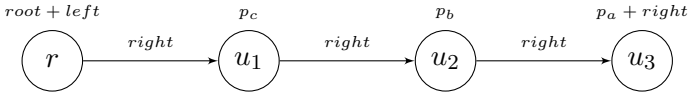
- For all $a \in \Sigma$, $right_a^{\rightarrow} \in T_\tau$ with $W^-(right_a^{\rightarrow}) = right + p_a$;
- For all $a \in \Sigma$, $left_a^{\rightarrow} \in T_{ab}$ with $W^-(left_a^{\rightarrow}) = W^+(left_a^{\rightarrow}) = left$ and $\Omega(left_a^{\rightarrow}) = left + p_a$;
- For all $a, b \in \Sigma$, $upd_{a,b}^{\leftarrow} \in T_{el}$ with $W^-(upd_{a,b}^{\leftarrow}) = right + p_a$ and $W^+(upd_{a,b}^{\leftarrow}) = right + p_b$
- For all $a \in \Sigma$, $left_a^{\leftarrow} \in T_\tau$ with $W^-(left_a^{\leftarrow}) = left + p_a$
- For all $a \in \Sigma$, $right_a^{\leftarrow} \in T_{ab}$ with $W^-(right_a^{\leftarrow}) = W^+(right_a^{\leftarrow}) = right$ and $\Omega(right_a^{\leftarrow}) = right + p_a$
- $check_\flat \in T_{el}$ with $W^-(check_\flat) = W^+(check_\flat) = right + root$;
- For all $b \in \Sigma$, $upd_{\flat,b}^{\leftarrow} \in T_{ab}$ with $W^-(upd_{\flat,b}^{\leftarrow}) = right$, $W^+(upd_{\flat,b}^{\leftarrow}) = right$ and $\Omega(upd_{\flat,b}^{\leftarrow}) = right + p_b$.

The initial state is $s[r, root + start + left + right]$.

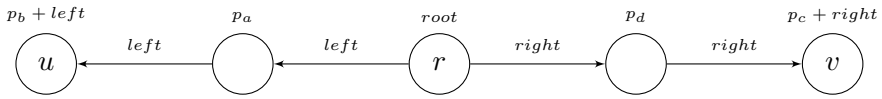
Let us explain how the simulation works. Let abc be the word on the tape of $\mathcal{M}$. Then firing $(r, t_a)(v_1, t_b)(v_2, t_c)$ one gets:



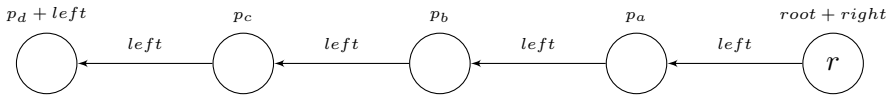
After firing $(v_3, next)(v_3, from_c)(r, to_c)(v_2, from_b)(u_1, to_b)(v_1, from_a)(u_2, to_a)(r, run)$ one gets:



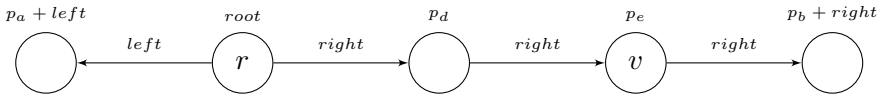
Let us describe the two cases of tape simulation. Assume that the content of the tape is $abcd\flat^\omega$ and that the head of $\mathcal{M}$ is over, c then the corresponding state is the following one. The "left" branch contains the content of the tape on the left of the head while descending to the leaf, and the "right" branch contains the relevant content of the tape on the right of the head (including the cell under the head) while ascending from the leaf. Thus, the token in place $right$ points to the thread corresponding to the cell under the head, while the token in place $left$ points to the thread corresponding to the cell immediately on the left of the head. The state of $\mathcal{M}$ is the state of $\mathcal{A}$.



Assume that the content of the tape is $abcd\flat^\omega$ and that the head of $\mathcal{M}$ is over the first $\flat$, then the corresponding state is the following one.



It is routine to check that the simulation works. Let us illustrate it with one example. Assume that the content of the tape is $abcd\flat^\omega$, the head of $\mathcal{M}$ is over c and the current state is ℓ. Let $\delta(\ell, c) = (\ell', e, \leftarrow)$. Then after firing $(v, upd^{\leftarrow}_{c,e})(u, left^{\leftarrow}_{b})(v, right^{\leftarrow}_{b})$, one gets:



For all $a \in \Sigma$, the homomorphism h maps t_a to a and for all $t \notin \{t_a\}_{a\in\Sigma}$, h maps t to ε. □

Obviously, the family of RPNs coverability languages includes the family of PNs coverability languages. In [71], Proposition 1 establishes that the family of context-free languages is included in family of reachability languages for RPNs. The proof relies on simulating the leftmost derivations of a context-free grammar within particular two places b_X and e_X per nonterminal symbol X, where a token in b_X means that X must derived and a token in e_X means that the derivation of X into a word has been achieved. In order to adapt this result for the family of coverability languages for RPNs, it is enough to consider w.l.o.g. that the initial symbol I never appears on the right-hand side of a rule and to specify $s[r, e_I]$ as final state.

Proposition 8.4.4. *The family of context-free languages is included in the family of coverability languages of RPNs.*

Proof. Let $G = (V, \Sigma, R, S)$ be a context-free grammar. We define a labelled marked net $\mathcal{N}$ as follows.

• The set of places is defined by:

$$P = \{b_v, e_v \mid v \in V\} \cup \{p_{i,j} \mid 1 \leq i \leq n \wedge 0 \leq j < n_i\}$$

• The set of transitions is defined by $T = \bigcup_{i\leq n} T_i \cup T_\tau$ where all abstract and cut transitions

are labelled by ε and:

- $T_\tau = \{t_v\}_{v\in V\setminus S}$ where $W^-(t_v) = e_v$
- If $n_i = 0$ then $T_i = \{t_{i,0}\}$ with $W^-(t_{i,0}) = b_S$ and $W^+(t_{i,0}) = e_S$, $t_{i,0} \in T_{el}$ and $\lambda(t_{i,0}) = \varepsilon$;
- Otherwise $T_i = \{t_{i,0}, t_{i,1}, \ldots, t_{i,n_i}\}$ with:

1. $W^-(t_{i,0}) = b_{v_i}$ and $W^+(t_{i,0}) = p_{i,0}$; $t_{i,0} \in T_{el}$ and $\lambda(t_{i,0}) = \varepsilon$;
2. for all $1 < j < n_i$ $W^-(t_{i,j}) = p_{i,j}$ and $W^+(t_{i,0}) = p_{i,j+1}$;
3. $W^-(t_{i,n_i}) = p_{i,n_i-1}$ and $W^+(t_{i,n_i}) = e_{v_i}$;
4. for all $1 < j \leq n_i$ if $u_{i,j} \in V$ then $t_{i,j} \in T_{ab}$ and $\Omega(t_{i,j}) = b_{u_i[j]}$ else $t_{i,j} \in T_{el}$, $\lambda(t_{i,j}) = u_i[j]$.

• The initial marking $s_0 = s[b_S]$. We want to show that $\mathcal{L}(G) = \mathcal{L}(\mathcal{N}, s_0, \{s_f\})$ where $s_f = s[e_S]$.

The result of a (possibly partial) leftmost derivation, not leading to an empty word, is represented by:

$$(w, u_{i_k}[j_k, n_k], u_{i_{k-1}}[j_{k-1}, n_{k-1}], \ldots, u_{i_1}[j_1, n_1])$$

with (1) for all $\ell \leq k$ $0 < n_\ell$, (2) $1 \leq j_k \leq n_k+1$, and (3) for all $\ell < k$ $1 < j_\ell \leq n_{\ell+1}$ where:

- $w \in \Sigma^*$ is the subword already generated;
- $(i_1, j_1) \cdots (i_k, j_k)$ are the current nested rules with the index of the next symbol to be processed (none when $j_\ell = n_\ell + 1$).

The leftmost derivations are simulated by the net as follows. There is a firing sequence with a trace w leading to a reachable state defined by:

- if $k = 0$ then $s[e_S]$;
- if $k \geq 1$ and $j_k = n_k + 1$ then a single branch of length $k - 1$ labelled by $t_{i_1,j_1-1}, \ldots t_{i_{k-1}-1,j_{k-1}-1}$ with all markings empty except the last one consisting of a token in e_{v_k};
- if $k \geq 1$, $j_k \leq n_k$ and $u_{i_k}[j_k] \in \Sigma$ then a single branch of length $k-1$ labelled by
  $t_{i_1,j_1-1}, \ldots t_{i_{k-1}-1,j_{k-1}-1}$ with all markings empty except the last one consisting of a token in p_{i_k,j_k};
- if $k \geq 1$, $j_k \leq n_k$ and $u_{i_k}[j_k] \in V$ then a single branch of length k labelled by $t_{i_1,j_1-1}, \ldots t_{i_k,j_k-1}$ with all markings empty except the last one consisting of a token in $b_{u_{i_k}[j_k]}$;

Let us describe the simulation.

- If $j_k \leq n_k$ and $u_{i_k}[j_k] \in V$ then the derivation consists in choosing some rule $r_{i_{k+1}} = u_{i_k}[j_k] \to u_{i_{k+1}}$ leading to $(w, u_{i_{k+1}}[1, n_{k+1}], u_{i_k}[j_k+1, n_k], \ldots, u_{i_1}[j_1, n_1])$. This is simulated by the firing of elementary transition $t_{i_{k+1},0}$ possibly followed by the firing of abstract transition $t_{i_{k+1},1}$ when $u_{i_{k+1}}[1] \in V$;
- If $j_k \leq n_k$ and $u_{i_k}[j_k] \in \Sigma$ then the derivation consists in concatenating the letter $u_{i_k}[j_k]$ to the word w leading to $(wu_{i_k}[j_k], u_{i_k}[j_k+1, n_k], u_{i_{k-1}}[j_{k-1}, n_{k-1}], \ldots, u_{i_1}[j_1, n_1])$. This is simulated by the firing of elementary transition t_{i_k,j_k+1} possibly followed by the firing of abstract transition t_{i_k,j_k+1} when $j_k + 1 \leq n_k$ and $u_{i_k}[j_k + 1] \in V$;
- If $j_k = n_k + 1$ then the derivation consists in deleting the empty word $u_{i_k}[j_k, n_k]$ leading to

$(w, u_{i_{k-1}}[j_{k-1}, n_{k-1}], \ldots, u_{i_1}[j_1, n_1])$.
This is simulated by the cut transition in the leaf of the branch producing the token specified by the postcondition of $t_{i_{k-1},j_{k-1}-1}$ possibly followed by the firing of abstract transition $t_{i_{k-1},j_{k-1}}$ when $j_{k-1} \leq n_{k-1}$ and $u_{i_{k-1}}[j_{k-1}] \in V$;
The case of the empty word is straightforward.
Conversely, the simulation from traces of firing sequences to words generated by (possibly partial) leftmost derivations is done similarly. □

Since universality is undecidable for the family of context-free languages, we deduce that universality of the family of RPN coverability languages is undecidable.

Let $\mathcal{L}_1 = \{a^m b^n c^p \mid m \geq n \geq p\}$. Denote by $\mathcal{L}_2 = \{w\tilde{w} \mid w \in \{d, e\}^*\}$, where $\tilde{w}$ is the mirror of w. Observe that given the final marking $\mathbf{0}$ we get that the net in Figure 8.5 has $\mathcal{L}_1$ as its coverability language.

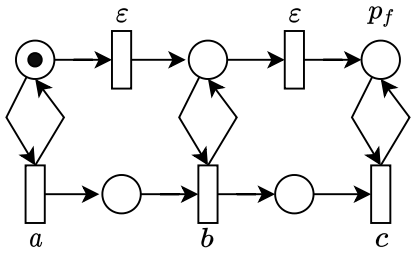


Figure 8.5: A Petri net for the language $\mathcal{L}_1$.

The next proposition witnesses a Petri net language, interesting from an expressiveness point of view. A similar result can be found page 179 in Peterson's book [127].

Proposition 8.4.5. *$\mathcal{L}_1$ is the coverability language of some Petri net, but it is not a context-free language.*

Proof. Let us recall (a weak version of) Ogden lemma [121]. For any context-free language $\mathcal{L}$ there exists an integer N such for any word $w \in \mathcal{L}$ with N marked positions, there exists a decomposition $w = w_1 w_2 w_3 w_4 w_5$ such that $w_2 w_4$ contains at least a marked position and for all $n \geq 0$, $w_1 w_2^n w_3 w_4^n w_5 \in \mathcal{L}$.
Assume that $\mathcal{L}_1$ is a context-free language and consider the word $w = a^N b^N c^N$ with all c positions marked. So let $w = w_1 w_2 w_3 w_4 w_5$ with the decomposition fulfilling the requirements of Ogden lemma. Since $w' = w_1 w_2^2 w_3 w_4^2 w_5 \in \mathcal{L}_1$, w_2 and w_4 are mono-letter words. Furthermore, one of these words is equal to c^q for some $q > 0$. If $w_2 = c^q$ then $w_4 = c^{q'}$ and thus w' contains too much c's to belong to $\mathcal{L}_1$. If

$w_4 = c^q$ then either $w_2 = a^{q'}$, $w_2 = b^{q'}$ or $w_2 = c^{q'}$. Whatever the case, w' misses either a's or b's to belong to $\mathcal{L}_1$.
As mentioned before the coverability language for the net in Figure 8.5 with final marking $\mathbf{0}$ is $\mathcal{L}_1$. □

Using the previous results, the next theorem emphasizes the expressive power of coverability languages of RPNs. Note that, in the following proof, we use Proposition 8.4.11 which shows the closure of the family of coverability languages of RPNs under the union operation. The proof for that proposition comes later in this section.

Theorem 8.4.6. *The family of coverability languages of RPNs strictly includes the union of the family of coverability languages of PNs and the family of context-free languages.*

Proof. The inclusion is an immediate consequence of Proposition 8.4.4. Consider the language $\mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2$.
Since (1) by Proposition 8.4.11, the family of coverability languages of RPNs is closed under union, (2) $\mathcal{L}_1$ is a PN language, and (3) the language of palindromes is a context-free language, we deduce that $\mathcal{L}$ is an RPN language.
PN and context-free languages are closed under homomorphism. Since the projection of $\mathcal{L}$ on $\{a, b, c\}$ is the language of Proposition 8.4.5, $\mathcal{L}$ is not a context-free language. The projection of $\mathcal{L}$ on $\{d, e\}$ is the language of palindromes which concludes the proof since it was seen in Proposition 2.3.15 that the language of (2 letters) palindromes is not a coverability language for any PN. □

The next proposition builds a language which is a reachability language, but not a coverability one. We do it using the language $\mathcal{L}_3 = \{a^n b^m c^m \mid n \geq m;\ m, n \in \mathbb{N}\}$. Observe that given the final marking p_f we get that the net in Figure 8.6 has $\mathcal{L}_3$ as its reachability language. Note that, originally in [59] we showed that the folk language $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ is not a coverability one, using an almost identical proof. By that showing that, as for Petri nets, coverability does not ensure the power of "exact counting". We replace this language, since we can reuse it in Section 9.3 to illustrate the difference between dynamic recursive Petri nets and RPNs. Moreover, the proof of the following proposition is interesting by itself, since it combines an argument based on WSTS (case 1) and an argument *à la* Ogden (case 2).

Proposition 8.4.7. *$\mathcal{L}_3$ is the reachability language of the Petri net of Figure 8.6, but it is not the coverability language of any RPN.*

Proof. Due to Proposition 8.4.2, it is enough to prove that there does not exist $(\mathcal{N}, s[r, m_0], \lambda)$ such that $\mathcal{L}_3 = \mathcal{L}_R(\mathcal{N}, s[r, m_0], \lambda, \{\emptyset\})$. Assume by contradiction

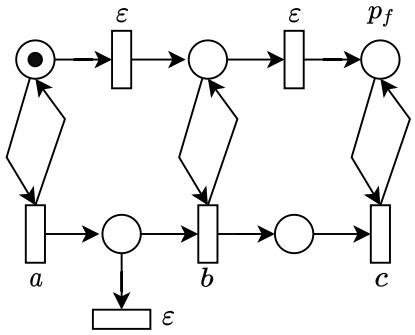
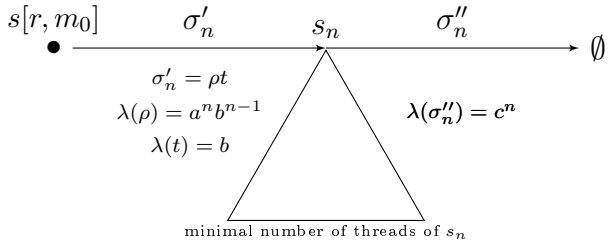


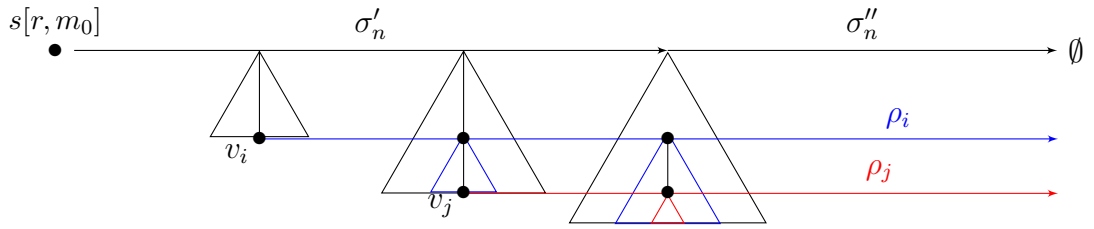
Figure 8.6: A Petri net for the language $\mathcal{L}_3$.

that there exists such $(\mathcal{N}, s[r, m_0]\})$. For all n, let σ_n be a firing sequence reaching $\emptyset$ such that $\lambda(\sigma_n) = a^n b^n c^n$ and σ'_n be the prefix of σ_n whose last transition corresponds to the last occurrence of b. Denote s_n the state reached by σ'_n and the decomposition by $\sigma_n = \sigma'_n \sigma''_n$. Among the possible σ_n, we select one such that s_n has a minimal number of threads. Let $Post$ be the finite set of $\mathbb{N}^P$, defined by: $Post = \{W^+(t)\}_{t \in T_{ab}}$.

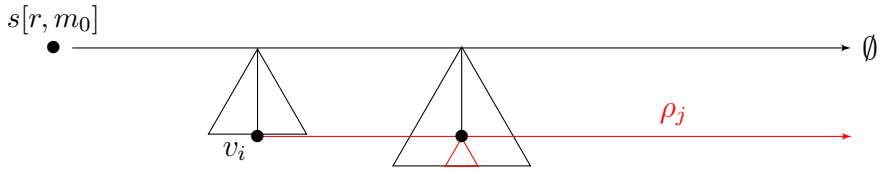


Case 1. There exists a bound B of the depths of the trees, corresponding to $\{s_n\}_{n \in \mathbb{N}}$. Let S_B be the set of abstract states of depth at most B and different from $\emptyset$. Observe that S_0 can be identified to $\mathbb{N}^P$ and S_B can be identified to $\mathbb{N}^P \times \mathsf{Multiset}(Post \times S_{B-1})$. Furthermore, the (component) order on $\mathbb{N}^P$ and the equality on $Post$ are wqos. Since wqo is preserved by the multiset operation and the Cartesian product, S_B is wqo by a qo denoted $<$. By construction, $s \leq s'$ implies $s \preceq_r s'$. Thus, there exist $n < n'$ such that $s_n \preceq_r s_{n'}$ which entails that $\sigma'_{n'} \sigma''_n$ is a firing sequence with trace $a^{n'} b^{n'} c^n$ reaching $\emptyset$, yielding a contradiction.
Case 2. The depths of the trees corresponding to $\{s_n\}_{n \in \mathbb{N}}$ are unbounded. There exists n such that the depth of s_n is greater than $(2|Post| + 1)$. Thus, in s_n for $1 \leq j \leq 3$, there are edges $u_j \xrightarrow{m}_{s_n} v_j$ and denoting i_j the depth of v_j, one has $0 < i_1 < i_2 < i_3$.
For $k \in \{1, 2, 3\}$, consider of the sequence ρ_k performed in the subtree rooted in v_k by the firings of σ_n. Among these three firing sequences, two of them either (1)

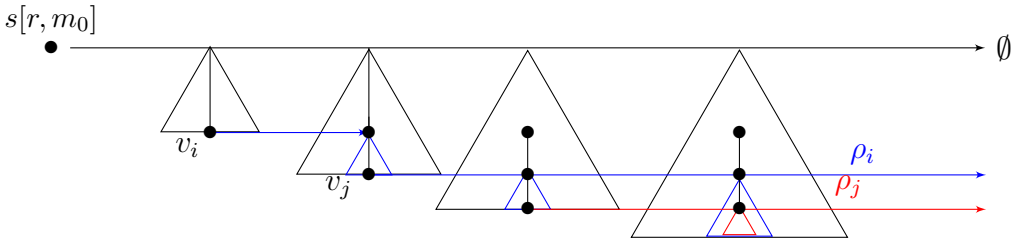
both finish by a cut transition in v_k or (2) both do not finish by a cut transition in v_k. Let us call i, j with $i < j$ the indices of these sequences and w_i and w_j their traces. We have illustrated the situation below.



One can build two firing sequences that still reach $\emptyset$ and thus whose labels belong to the language. The first one consists of mimicking the "behavior" of the subtree rooted in v_j starting from v_i, which is possible due to the choice of i and j, as illustrated below.



The second one consists of mimicking the "behavior" of the subtree rooted in v_i starting from v_j as illustrated below.



Denote by w the trace of the sequence performed in the subtree rooted in v_i without the trace of the sequence performed in the subtree rooted in v_j.

Case $w = \varepsilon$. Then the firing sequence reaching $\emptyset$ obtained by mimicking in v_i the behavior of v_j has trace $a^n b^n c^n$ and leads to another state s_n with fewer threads yielding a contradiction, since s_n was supposed to have a minimal number of threads.

Case $w = a^k$ for $k > 0$. Consider the firing sequence σ reaching $\emptyset$ obtained by mimicking in v_i the behavior of v_j. The trace of this sequence is $a^{n-k} b^n c^n \notin \mathcal{L}$, which yields a contradiction.

Case $w \neq a^k$ for $k \geq 0$. Let us consider the firing sequence σ reaching $\emptyset$ obtained by mimicking in v_j the behavior of v_i. The trace of σ is an interleaving of $a^n b^n c^n$ and w and it belongs to $\mathcal{L}_3$, which implies that $w = a^k b^q c^q$ for some $q > 0$ and $k \geq 0$. Furthermore, σ can be chosen in such a way that the firing subsequences in the subtrees rooted at v_i and v_j are performed in one shot, which implies that its trace is $\ldots a^k a^k w_j b^q c^q b^q c^q \ldots$ yielding a contradiction. □

The following corollary shows that extending the family of coverability languages of PNs by substituting either (1) coverability by reachability or (2) PNs by RPNs is somewhat "orthogonal".

Corollary 8.4.8. *The families of reachability languages of Petri nets and the family of coverability languages of RPNs are incomparable.*

Proof. One direction is a consequence of Proposition 8.4.7 while the other direction is a consequence of Proposition 8.4.4 observing that the language of palindromes is not the reachability language of any Petri net (see Proposition 2.3.15). □

Combining Propositions 8.4.1, 8.4.2 and 8.4.7, one gets the following theorem.

Theorem 8.4.9. *The family of coverability languages of RPNs is strictly included in the family of reachability languages of RPNs.*

Figure 8.7 illustrates the hierarchy of the languages presented in this work.

8.4.3 Closure Properties

We now study the closure properties of the family of RPN coverability languages under several operations. As mentioned in the Peterson book, this analysis is important for two main reasons: It helps to understand the properties of the RPN coverability languages and these compositions may help to design and construct large systems by composing them from smaller ones.

Using proposition 8.4.1 and 8.4.2 we study the closure properties of the family of RPN cut languages instead of the family of RPN coverability languages. Moreover, using Proposition 8.3.4 we assume w.l.o.g. that the initial state of the RPNs has a single vertex.

Lemma 8.4.10. *Given a marked RPN $(\mathcal{N}, s_0)$ with a labeling λ. Then the marked RPN $(\mathring{\mathcal{N}}, \mathring{s}_0)$ from Definition 8.3.1 with the labeling :*

$$\mathring{\lambda}(t) = \begin{cases} \lambda(t) & t \in T \\ \varepsilon & else \end{cases}$$

has the same cut language.

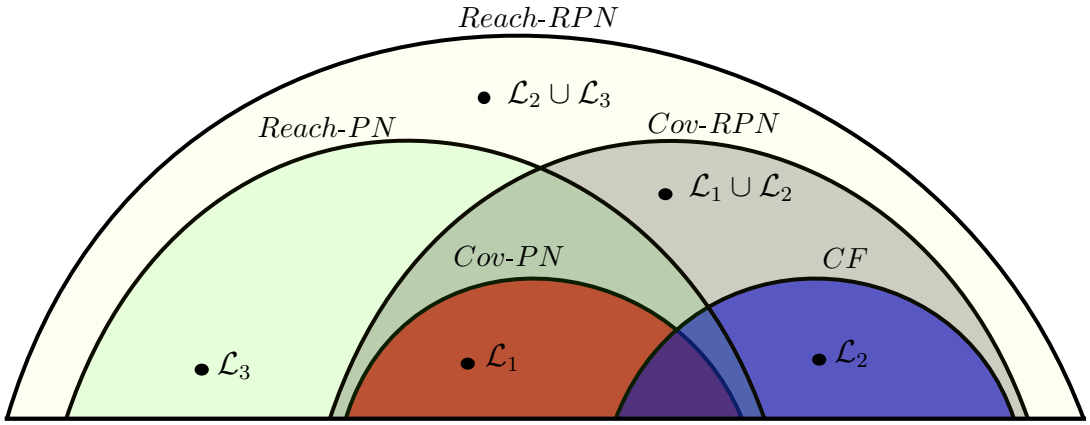


Figure 8.7: $\mathcal{L}_1 = \{a^m b^n c^p \mid m \geq n \geq p\}; \mathcal{L}_2 = \{w \in \{d,e\}^* \mid w = \widetilde{w}\}; \mathcal{L}_3 = \{a^m b^n c^n \mid m \geq n \in \mathbb{N}\}$.

Proof. We show this lemma by a double inclusion.

On the one hand, if $w \in \mathcal{L}_R(\mathcal{N}, s_0, \lambda, \{\emptyset\})$, there exists a cut sequence σ such that $\lambda(\sigma) = w$. From Lemma 8.3.2 we have that $\mathring{s}_0 \xrightarrow{\sigma^e_{s_0}\sigma}_{\mathring{\mathcal{N}}} \emptyset$, and by definition $\mathring{\lambda}(\sigma^e_{s_0}\sigma) = \mathring{\lambda}(\sigma^e_{s_0})\mathring{\lambda}(\sigma) = \varepsilon w = w$. Therefore $w \in \mathcal{L}_R(\mathring{\mathcal{N}}, \mathring{s}_0, \mathring{\lambda}, \{\emptyset\})$.

On the other hand, if $w \in \mathcal{L}_R(\mathring{\mathcal{N}}, \mathring{s}_0, \mathring{\lambda}, \{\emptyset\})$, there exists a cut sequence σ such that $\mathring{\lambda}(\sigma) = w$. From Lemma 8.3.3 we have that $s_0 \xrightarrow{\sigma_{|\mathcal{N}}}_{\mathcal{N}} \emptyset$, moreover $\mathring{\lambda}(\sigma) = \mathring{\lambda}(\sigma_{|\mathcal{N}}) = \lambda(\sigma_{|\mathcal{N}})$. Therefore $w \in \mathcal{L}_R(\mathcal{N}, s_0, \lambda, \{\emptyset\})$. □

Proposition 8.4.11. *The family of RPN coverability/cut languages is closed under union.*

Proof. Consider two labelled marked RPNs $(\mathcal{N}, s[r, m_0], \lambda)$ and $(\mathcal{N}', s[r', m'_0], \lambda')$. Let us define $\widetilde{\mathcal{N}}$ as follows (see Figure 8.8 for an illustration of this construction): $\widetilde{\mathcal{N}}$'s set of places is the disjoint union of P and P' with two additional places p_0 and p_e. Its set of transitions is the disjoint union of T and T' with two additional abstract transitions t, and t' and a cut transition t_e.

- For all $t \in T \cup T'_{ab}$, $\widetilde{W}^-(t) = W^-(t) + p$ and when $t \notin T_\tau \cup T'_\tau$ $\widetilde{W}^+(t) = W^+(t)$
- For all $t \in T_{ab} \cup T'_{ab}$, $\widetilde{\Omega}(t) = \Omega(t)$
- $\widetilde{W}^-(t_{start}) = p_0, \widetilde{W}^+(t_{start}) = p_e$, $\widetilde{\Omega}(t) = m_0$
- $\widetilde{W}^-(t'_{start}) = p_0, \widetilde{W}^+(t'_{start}) = p_e$, $\widetilde{\Omega}(t') = m'_0$
- $\widetilde{W}^-(t_e) = p_e$

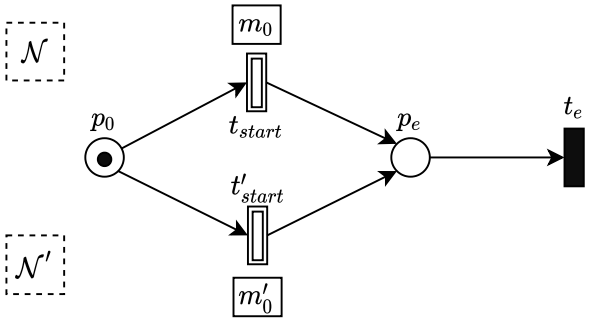


Figure 8.8: Union of two RPN cut languages.

- For all $t \in T$, $\widetilde{\lambda}(t) = \lambda(t)$ and for all $t \in T'$, $\widetilde{\lambda}(t) = \lambda'(t)$
- For all $t \in \{t_{start}, t'_{start}, t_e\}$, $\widetilde{\lambda}(t) = \varepsilon$.
- The initial state of $\widetilde{\mathcal{N}}$ is $s[\tilde{r}, p_0]$.

Let us prove that $\mathcal{L}_R(\mathcal{N}, s[r, m_0], \lambda, \{\emptyset\}) \cup \mathcal{L}_R(\mathcal{N}', s[r', m'_0], \lambda', \{\emptyset\}) \subseteq \mathcal{L}_R(\widetilde{\mathcal{N}}, s[\tilde{r}, \widetilde{\lambda}, p_0], \{\emptyset\})$. Let σ be a cut sequence of $(\mathcal{N}, s[r, m_0])$. The corresponding cut sequence $\widetilde{\sigma}$ of $(\widetilde{\mathcal{N}}, s[\tilde{r}, p_0])$ is built as follows. Initially, one fires $(\tilde{r}, t_{start})$, creating a thread r with a marking m_0. Following it one fires σ which reaches the state $s[\tilde{r}, p_0]$. Finally, one fires $(\tilde{r}, t_e)$ reaching $\emptyset$. The word generated by this cut sequence is $\widetilde{\lambda}((\tilde{r}, t_{start})\sigma(\tilde{r}, t_e)) = \lambda(\sigma)$. The proof for $\mathcal{L}_R(\mathcal{N}', s[r', m'_0], \lambda', \{\emptyset\})$ is similar.

Let us prove that $\mathcal{L}_R(\widetilde{\mathcal{N}}, s[r, p_0], \widetilde{\lambda}, \{\emptyset\}) \subseteq \mathcal{L}_R(\mathcal{N}, s[r, m_0], \lambda, \{\emptyset\}) \cup \mathcal{L}_R(\mathcal{N}', s[r', m'_0], \lambda', \{\emptyset\})$. Observe that any cut sequence of $(\widetilde{\mathcal{N}}, s[\tilde{r}, p_0])$ must start by a firing of t_{start} or t'_{start} and end with t_e. Assume that the cut sequence from $(\widetilde{\mathcal{N}}, s[\tilde{r}, p_0])$ generating w starts by firing t_{start}, i.e., $(\tilde{r}, t_{start})\widetilde{\sigma}(\tilde{r}, t_e)$. The sequence $\widetilde{\sigma}$ is fireable in $(\mathcal{N}, s[r, m_0])$, and the word generated by it is $\lambda(\widetilde{\sigma}) = \widetilde{\lambda}((\tilde{r}, t_{start})\widetilde{\sigma}(\tilde{r}, t_e)) = w \in \mathcal{L}_R(\mathcal{N}, s[r, m_0], \lambda, \{\emptyset\})$. Similarly, we show that if the cut sequence would start by firing t'_{start}, then $w \in \mathcal{L}_R(\mathcal{N}', s[r', m'_0], \lambda', \{\emptyset\})$. □

Proposition 8.4.12. *The family of RPN coverability/cut languages is closed under concatenation.*

Proof. Consider two labelled marked RPNs $(\mathcal{N}, s[r, m_0], \lambda)$ and $(\mathcal{N}', s[r', m'_0]\lambda')$. Let us define $\widetilde{\mathcal{N}}$ as follows, see Figure 8.9. Its set of places is the disjoint union of P and P' with three additional places p_0, p_c and p_e. Its set of transitions is the disjoint union of T and T' with two additional abstract transitions t, and t' and a cut transition t_e.

- For all $t \in T \cup T'_{ab}$, $\widetilde{W^-}(t) = W^-(t)$ and when $t \notin T_\tau \cup T'_\tau$ $\widetilde{W^+}(t) = W^+(t)$
- For all $t \in T_{ab} \cup T'_{ab}$, $\widetilde{\Omega}(t) = \Omega(t)$
- $\widetilde{W^-}(t_{start}) = p_0, \widetilde{W^+}(t_{start}) = p_c$, $\widetilde{\Omega}(t) = m_0$

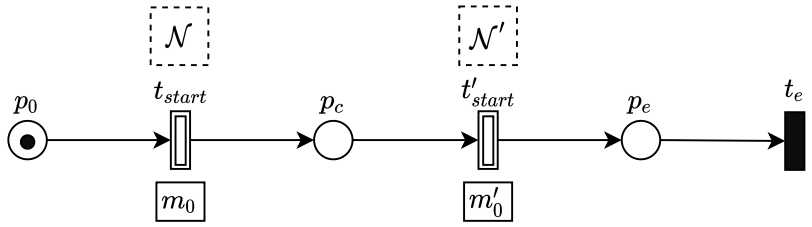


Figure 8.9: Concatenation of two RPN cut languages.

- $\widetilde{W}^-(t'_{start}) = p_c, \widetilde{W}^+(t'_{start}) = p_e$, $\widetilde{\Omega}(t') = m'_0$
- $\widetilde{W}^-(t_e) = p_e$
- For all $t \in T$, $\widetilde{\lambda}(t) = \lambda(t)$ and for all $t \in T'$, $\widetilde{\lambda}(t) = \lambda'(t)$
- For all $t \in \{t_{start}, t'_{start}, t_e\}$, $\widetilde{\lambda}(t) = \varepsilon$.
- The initial state of $\widetilde{\mathcal{N}}$ is $s[\tilde{r}, p_0]$.

Let us prove that $\mathcal{L}_R(\mathcal{N}, s[r, m_0], \lambda, \{\emptyset\}) \cdot \mathcal{L}_R(\mathcal{N}', s[r', m'_0], \lambda', \{\emptyset\}) \subseteq \mathcal{L}_R(\widetilde{\mathcal{N}}, s[\tilde{r}, p_0], \widetilde{\lambda}, \{\emptyset\})$. Let σ be a cut sequence of $(\mathcal{N}, s[r, m_0])$ and σ' be a cut sequence of $(\mathcal{N}', s[r', m'_0])$. The corresponding cut sequence $\widetilde{\sigma}$ of $L(\widetilde{\mathcal{N}}, s[\widetilde{r}, p_0])$ is built as follows. Initially, one fires $(\widetilde{r}, t_{start})$, creating a new thread denoted by r. Then after the creation of a thread r, one fires σ which ends in a cut bringing us to the state $s[\widetilde{r}, p_c])$. Next, one fires $(\widetilde{r}, t'_{start})$, creating a new thread denoted by r'. After the creation of a thread r', one fires σ' which ends in a cut finishing with the state $s[\widetilde{r}, p_e])$. Finally, we finish by firing the cut transition $(\widetilde{r}, t_e)$, which leads us to the $\emptyset$. All together, we get the firing sequence $\widetilde{\sigma} = (\widetilde{r}, t_{start})\sigma(\widetilde{r}, t'_{start})\sigma'(\widetilde{r}, t_e)$, for which $\widetilde{\lambda}(\widetilde{\sigma}) = \varepsilon\, \lambda(\sigma)\, \varepsilon\, \lambda'(\sigma')\, \varepsilon = \lambda(\sigma)\lambda'(\sigma')$.

Let us prove that $\mathcal{L}_R(\widetilde{\mathcal{N}}, s[\tilde{r}, p_0], \widetilde{\lambda}, \{\emptyset\}) \subseteq \mathcal{L}_R(\mathcal{N}, s[r, m_0], \lambda, \{\emptyset\}) \cdot \mathcal{L}_R(\mathcal{N}', s[r', m'_0], \lambda', \{\emptyset\})$. Observe that any cut sequence must be of the type $\widetilde{\sigma} = (\widetilde{r}, t_{start})\sigma(\widetilde{r}, t'_{start})\sigma'(\widetilde{r}, t_e)$, where (1) $(\widetilde{r}, t_{start})$ and $(\widetilde{r}, t'_{start})$ create a new thread r and r' with the marking m_0 and m'_0 respectively, (2) σ and σ' are firing sequences ending by a cut transition fired from r and r' respectively, and (3) σ and σ' are fireable from $s[r, m_0]$, and $s[r', m'_0]$ respectively. Moreover, $\widetilde{\lambda}(\widetilde{\sigma}) = \lambda(\sigma)\lambda'(\sigma')$ which finishes the proof for concatenation. □

Proposition 8.4.13. *The family of RPN coverability/cut languages is closed under Kleene star.*

Proof. Consider a labelled marked RPN $(\mathcal{N}, s[r, m_0], \lambda)$. Let us define $\widetilde{\mathcal{N}}$ as follows, see Figure 8.10. Its set of places is the disjoint union of P with one additional places p_0. Its set of transitions is the disjoint union of T, with one additional abstract transitions t_{start} and a cut transition t_e.

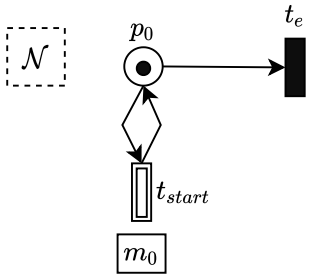


Figure 8.10: Kleene star of an RPN cut language.

- For all $t \in T$, $\widetilde{W}^-(t) = W^-(t)$ and when $t \notin T_\tau$ $\widetilde{W}^+(t) = W^+(t)$
- For all $t \in T_{ab}$, $\widetilde{\Omega}(t) = \Omega(t)$
- $\widetilde{W}^-(t_{start}) = p_0, \widetilde{W}^+(t_{start}) = p_0$, $\widetilde{\Omega}(t_{start}) = m_0$
- $\widetilde{W}^-(t_e) = p_e$
- For all $t \in T$, $\widetilde{\lambda}(t) = \lambda(t)$
- For all $t \in \{t_{start}, t_e\}$, $\widetilde{\lambda}(t) = \varepsilon$.
- The initial state of $\widetilde{\mathcal{N}}$ is $s[\tilde{r}, p_0]$.

Let us prove that $\mathcal{L}_R(\mathcal{N}, s[r, m_0], \lambda, \{\emptyset\})^* \subseteq \mathcal{L}_R(\widetilde{\mathcal{N}}, s[\tilde{r}, p_0], \widetilde{\lambda}, \{\emptyset\})$. Let $\{\sigma_i\}_{i<n}$ for $n \in \mathbb{N}$ be cut sequences of $(\mathcal{N}, s[r, m_0])$. For any $i < n$, construct a sequence $\widetilde{\sigma_i}$ of $(\widetilde{\mathcal{N}}, s[\widetilde{r}, p_0])$ as follows. First we fire $(\widetilde{r}, t_{start})$ creating a new thread r with the marking m_0. Afterwards, we fire σ_i which ends with a cut transition from thread r, and reaches the state $s[\widetilde{r}, p_0]$. Note that $\widetilde{\lambda}(\widetilde{\sigma_i}) = \lambda(\sigma_i)$. Therefore, the sequence $\widetilde{\sigma_1}\widetilde{\sigma_2}\ldots\widetilde{\sigma_n}$ is fireable from $s[\widetilde{r}, p_0])$ ending in the state $s[\widetilde{r}, p_0])$. Finally, we fire $(\widetilde{r}, t_e)$ reaches $\emptyset$. In total, we get a cut sequence generating the word $\widetilde{\lambda}(\widetilde{\sigma_1}\widetilde{\sigma_2}\ldots\widetilde{\sigma_n}(\widetilde{r}, t_e)) = \lambda(\sigma_1)\lambda(\sigma_2)\ldots\lambda(\sigma_n)$.

Let us prove that $\mathcal{L}_R(\widetilde{\mathcal{N}}, s[\tilde{r}, p_0], \widetilde{\lambda}, \{\emptyset\}) \subseteq \mathcal{L}_R(\mathcal{N}, s[r, m_0], \lambda, \{\emptyset\})^*$. Observe that any cut sequence must be of the type $\widetilde{\sigma} = (\widetilde{r}, t)\sigma_1(\widetilde{r}, t)\sigma_2\ldots(\widetilde{r}, t_{start})\sigma_n(\widetilde{r}, t_e)$ for $i \in \mathbb{N}$, where each σ_i is a cut sequence fireable from $(\mathcal{N}, s[r, m_0])$. Since, $\widetilde{\lambda}(t_{start}), \widetilde{\lambda}(t_e) = \varepsilon$, we get that

$$\widetilde{\lambda}((\widetilde{r}, t_{start})\sigma_1(\widetilde{r}, t_{start})\sigma_2\ldots(\widetilde{r}, t_{start})\sigma_n(\widetilde{r}, t_e)) = \lambda(\sigma_1)\lambda(\sigma_2)\ldots\lambda(\sigma_n)$$

which finishes the proof. □

The next proposition exhibits a particular feature of RPNs languages (e.g., Petri nets or context-free languages are closed under intersection with a regular language).

Proposition 8.4.14. *The family of coverability/cut languages of RPNs is not closed under intersection with a regular language not under complementation.*

Proof. Due to Proposition 8.4.7, the family of coverability languages of RPNs is strictly included in the family of recursively enumerable languages. Since the former family is closed under homomorphism, Theorem 8.4.3 implies that it is not closed under intersection with a regular language and a fortiori with another coverability language. Since intersection can be obtained by union and complementation and since the family of RPN coverability languages is closed under union, they are not closed under complementation. □

8.5 Coverability is EXPSPACE-Complete

The section is devoted to the proof that the coverability problem is in EXPSPACE-complete. The EXPSPACE-hardness follows immediately from the EXPSPACE-hardness of the coverability problem for Petri nets [107].

Observe that the coverability problem is equivalent to the emptiness problem of the coverability language of an RPN. In Section 8.4 we have shown that the families of coverability languages and cut languages for RPN are equal and that the transformation from one to another is performed in polynomial time (proposition 8.4.1 and 8.4.2). Therefore, we will establish the complexity result for the cut problem, getting as a corollary the same result for the coverability problem.

Theorem 8.5.1. *The cut problem is* EXPSPACE*-complete.*

Proof. Let $(\mathcal{N}, s_0)$ be a marked RPN and η the accumulated size of the RPN and the initial state. By Proposition 8.3.4 we assume w.l.o.g. that V_{s_0} is a singleton $\{r\}$. Assume there exists a firing sequence $s_0 \xrightarrow{\sigma}_{\mathcal{N}} \emptyset$. Using Proposition 8.3.8 one gets an omniscient sequence $s_0 \xrightarrow{\widehat{\sigma}}_{\widehat{\mathcal{N}}} \emptyset$ such that $\widehat{\sigma} = (r, \sigma_1)(r, t)$ for some $t \in T_\tau$. The (omniscient) sequence (r, σ_1) contains only elementary transitions. Thus, $m_0 \xrightarrow{\sigma_1}_{\widehat{\mathcal{N}}_{el}} m$ with $m \geq W^-(t)$. On the other hand if there exists $m_0 \xrightarrow{\sigma}_{\widehat{\mathcal{N}}_{el}} m \geq W^-(t)$ then there exists a cut sequence in $\mathcal{N}$ from the state s_0.
Therefore, by combing the following two facts 1) one can build $\widehat{\mathcal{N}}_{el}$ in exponential space by constructing $\widehat{\mathcal{N}}$ Corollary 8.3.12, and 2) coverability in Petri nets is EXPSPACE, we are done.

□

The next theorem is an immediate corollary of the previous one.

Theorem 8.5.2. *The coverability problem for RPNs is* EXPSPACE*-complete.*

8.6 Termination is EXPSPACE-Complete

In this section we address the termination problem for RPN. Let $(\mathcal{N}, s_0)$ be a marked RPN. We denote the size of the input of the termination problem by η.

In [132] Rackoff showed that the termination problem for Petri net is solvable in exponential space:

Theorem 8.6.1 (Rackoff's Theorem [107, 132])**.** *The termination problem for Petri nets is* EXPSPACE*-complete.*

We aim to show that the termination problem for RPN is EXPSPACE-complete. EXPSPACE-hardness follows immediately from EXPSPACE-hardness of the termination problem for Petri nets [107]. By Proposition 8.3.4 we assume w.l.o.g. that $V_{s_0} = \{r\}$. Hence, for the rest of the section, we will assume that $s_0 = s[r, m_0]$ for some marking m_0.
A main ingredient of the proof is the construction of an *abstract graph* related to the firing of abstract transitions.

Definition 8.6.2 (abstract graph)**.** Let $(\mathcal{N}, s_0)$ be a marked RPN. Let $G_{\mathcal{N},s_0} = (V_a, E_a, M_a)$ be a labeled directed graph defined inductively as follows:

1. $r \in V_a$ and $M_a(r) = m_0$;
2. For any $v \in V_a$ and $t \in T_{ab}$, if there exists $s[v, M_a(v)] \xrightarrow{\sigma(v,t)}$ then $v_t \in V_a$, $(v, v_t) \in E_a$ and $M_a(v) = \Omega(t)$.

Observe that an edge (v, v_t) means that from state $s[v, M_a(v)]$, the thread v can fire t in the future and by induction that $v_t \in V_a$ if and only if t is fireable in the marked RPN. Observe that the size of $G_{\mathcal{N},s_0}$ is linear w.r.t. the size of $(\mathcal{N}, s_0)$.

Lemma 8.6.3. *Let $(\mathcal{N}, s_0)$ be a marked RPN. Then one can build its abstract graph in exponential space.*

Proof. First, note that $|V_a| \leq |T_{ab}| + 1$. Then for any vertex v already in V_a and any $t \in T_{ab}$ checking whether $s[v, M_a(v)] \xrightarrow{\sigma(v,t)}$ is fireable is equivalent to solving the covering problem $M_a(v) \xrightarrow{\sigma} m \geq W^-(t)$ in $\widehat{\mathcal{N}}_{el}$ (recall Definition 8.3.13) which can be done in exponential space due to Rackoff's coverability theorem for Petri nets. □

While we will not prove it, using a reduction from the Petri net coverability problem, one can show that we cannot use less than an exponential space to build the abstract graph.

Let us illustrate the abstract graph in Figure 8.11 corresponding to the RPN of Figure 7.1. Here the initial state is $s[r, p_{ini}]$. For clarity, we have renamed the abstract transitions as follows: $t := t_{beg}$, $ta := t_{a_2}$, $tb := t_{b_2}$. For instance, the existence of the edge from v_t to v_{ta} is justified by the firing sequence $(v_t, t_{a_1})(v_t, ta)$.

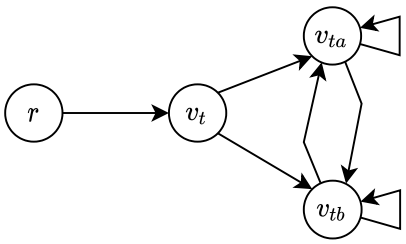


Figure 8.11: An abstract graph for the RPN in Figure 7.1

Let σ be an infinite firing sequence. We say that σ is *deep* if it visits a state s whose depth is strictly greater than $|T_{ab}|$. Otherwise, we say that σ is *shallow*. To solve the termination problem it suffices to decide whether the RPN has such an infinite sequence, either shallow or deep.

The next lemma establishes that *lassos* of the abstract graph are witnesses of deep infinite sequences in an RPN:

Lemma 8.6.4. *Let $(\mathcal{N}, s_0)$ be a marked RPN. Then there is a deep infinite sequence starting from s_0 if and only if there is a cycle in $G_{\mathcal{N},s_0}$.*

Proof.
• Assume that σ is a deep sequence. Hence, it reaches a state $\tilde{s}$ whose tree has a path γ starting from the root, with $|\gamma| > |T_{ab}|$. Let us denote it by $\gamma = (v_i)_{i=1}^{m}$. For all $i \leq m$ denote by t_i the abstract transition that creates v_i. Using γ, one builds a path $\gamma_a = v_1 v_2 \dots v_m$ in $G_{\mathcal{N},s_0}$ as follows. First $v_1 = r$ and $m_r = M_a(r)$. Since along σ the thread r fires t_1 to create v_2, there is an edge between r to v_{t_2} in $G_{\mathcal{N},s_0}$. For any $1 < i \leq m$, the thread v_i is created with the marking $\Omega(t_i) = M_a(v_{t_i})$. Since v_{i+1} is a child of v_i, somewhere in the sequence σ the thread v_i fires t_{i+1}. Therefore, there is an edge from v_{t_i} to $v_{t_{i+1}}$ in $G_{\mathcal{N},s_0}$. The length of the path γ_a strictly greater than $|T_{ab}|$, and since $V_a \leq |T_{ab}| + 1$ there is a cycle in γ_a.
• Conversely, assume that there is a cycle in $G_{\mathcal{N},s_0}$. Then there is an infinite path $\gamma_a = \{v_i\}_{i=0}^{\infty}$ in $G_{\mathcal{N},s}$ starting from r, where for any $i \geq 1$ denote by t_i the abstract transition associated to the vertex v_i. We now translate this infinite path to a deep sequence on $\mathcal{N}$ with initial state s_0. Note that $v_0 = r$ and that $m_r = M_a(r)$. By definition of E_a, there is a sequence $s \xrightarrow{\sigma_1} s'_0$ where the abstract transition t_1 is fireable from v_0 in s'_0. We get $s \xrightarrow{\sigma_1} s'_0 \xrightarrow{(v_0,t_1)} s_2$. Denote by v_1 the thread created by t_1. The thread's marking has $M_{s_1}(v_1) = M_a(v_1)$, therefore one continues translating the path γ_a in the same way as the first edge. Since for any (v_i, v_{i+1}) in γ_a we create a new thread from v_i one gets a deep sequence. □

We now show that for any shallow σ there is a thread v which fires infinitely many times in σ.

Lemma 8.6.5. *Let $(\mathcal{N}, s_0)$ be a marked RPN and σ be a shallow sequence. Then there is a thread v that fires infinitely many times in σ.*

Proof. If the root r fires infinitely often, then we are done. Otherwise, r has finitely many children, and the firing subsequence of σ of the subtree of (at least) one child, say v, must be infinite. If v fires infinitely often, then we are done. Otherwise, we proceed inductively up to $|T_{ab}|$ where some thread must fire infinitely often. □

We now show that given some state $s[r, m_0]$ one can check in exponential space the existence of a shallow sequence in which r fires infinitely many times.

Lemma 8.6.6. *Let $(\mathcal{N}, s_0)$ be a marked RPN. Then one can check in exponential space, whether there exists an infinite sequence starting with r firing infinitely many times.*

Proof. We first show that there is a sequence where r fires infinitely many times if and only if there is a infinite firing sequence in the marked Petri net $(\widehat{\mathcal{N}}_{el}, m_0)$.
• Assume there exists such σ in $(\mathcal{N}, s[r, m_0])$. Then the sequence σ is also fireable in $(\widehat{\mathcal{N}}, s[r, m_0])$. In $\widehat{\mathcal{N}}$, one eliminates in σ the cut transitions by increasing occurrence order as follows. Let (v, t) be a cut transition and (v', t') be the firing that creates v. Then one deletes all the firings performed by the descendants of v and replaces (v', t') by (v', t'^r). Let σ' be the sequence obtained after this transformation. In σ', the root still fires infinitely often, since no firing performed by the root has been deleted (but sometimes substituted by an elementary firing). Moreover, σ' has no more cut transitions. Consider the still infinite firing sequence (r, σ'') where in σ' all firings in other vertices than r have been deleted. Observe now that, by definition, σ'' is also an infinite sequence of $\widehat{\mathcal{N}}_{el}$.
• Conversely, assume there exists an infinite firing sequence σ of $(\widehat{\mathcal{N}}_{el}, m_0)$. Then (r, σ) is an infinite firing sequence of $(\widehat{\mathcal{N}}, s[r, m_0])$ (with only root firings) entailing the existence of an infinite firing sequence of $(\mathcal{N}, s[r, m_0])$.
By Theorem 8.6.1, one can check in exponential space whether there exists an infinite sequence of $(\widehat{\mathcal{N}}_{el}, m_0)$. □

Summing up the results for shallow and deep sequences we get:

Theorem 8.6.7. *The termination problem of RPN is in* EXPSPACE*-complete.*

Proof. The algorithm proceeds as follows. It builds in EXPSPACE (by Lemma 8.6.3) the abstract graph and checks whether there is a deep infinite sequence using the characterization of Lemma 8.6.4. In the negative case, it looks for a shallow infinite sequence. To this aim, it checks in exponential space for any reachable vertex v from r in $G_{\mathcal{N},s_0}$, whether there exists an infinite sequence starting from $s[v, M_a(v)]$ with the root firing infinitely many times. The complexity follows from Lemma 8.6.6 while the correctness follows from Lemma 8.6.5. □

8.7 Finiteness and Boundedness are EXPSPACE-complete

In this section we will show that the finiteness and boundedness problems for RPNs are EXPSPACE-complete w.r.t. $\eta = size(\mathcal{N}, s_0)$, i.e. the accumulated size of the RPN and the initial state. For Petri nets the finiteness problem, which is equivalent to the boundedness problem, has been shown to be EXPSPACE-complete:

Theorem 8.7.1 ([107, 132]). *The finiteness problem for Petri nets is* EXPSPACE*-complete.*

EXPSPACE-hardness follows immediately from EXPSPACE-hardness of the finiteness problem for Petri nets [107].

Moreover, by applying Proposition 8.3.4 like in previous sections we will assume that $s_0 = s[r, m_0]$. Given two vertices u, v in a graph $\mathcal{G}$, the *distance* between them $dist_{\mathcal{G}}(u, v)$ is the length of a shortest path going from u to v.

Lemma 8.7.2. *Let $(\mathcal{N}, s_0)$ be a marked RPN and $G_{\mathcal{N},s_0} = (V_a, E_a, M_a)$ be its abstract graph. Then for all $v \in V_a$, there exists $s \in Reach(\mathcal{N}, s_0)$ and $u \in V_s$ such that $M_s(u) = M_a(v)$.*

Proof. We show the lemma by induction on $dist_{G_{\mathcal{N},s_0}}(r, u)$. If $dist_{G_{\mathcal{N},s_0}}(r, v) = 0$ then $v = r$ and $M_a(r) = m_0$. Assume that we have shown the lemma for any v such that $dist_{G_{\mathcal{N},s_0}}(r, v) < n$, and pick $v \in V_a$ such that $dist_{G_{\mathcal{N},s_0}}(r, v) = n$. Since $dist_{G_{\mathcal{N},s_0}}(v, r) > 0$, $v = v_t$ for some $t \in T_{ab}$. Moreover, there is some $(u, v_t) \in E_a$ such that $dist_{G_{\mathcal{N},s_0}}(r, u) = n - 1$ and by the induction hypothesis there is a sequence $s_0 \xrightarrow{\sigma_u} s_u$ and some $w \in V_{s_u}$ such that $M_{s_u}(w) = M_a(u)$. From the definition of $G_{\mathcal{N},s_0}$, there is a fireable sequence $s[w, M_a(w)] \xrightarrow{\sigma_t(w,t)}$. Combining these sequences, we get $s_0 \xrightarrow{\sigma_u} s_u \xrightarrow{\sigma_t(w,t)} s_{v_t}$, where the newly created thread w' fulfills $M_{s_v}(w') = \Omega(t) = M_a(v_t)$. □

The following lemma shows that we can simulate the behavior of every thread by a Petri net.

Lemma 8.7.3. *Let $(\mathcal{N}, s_0)$ be a marked RPN and $G_{\mathcal{N},s_0} = (V_a, E_a, M_a)$ be its abstract graph. Then:*

$$\bigcup_{s \in Reach(\mathcal{N},s_0)} \{M_s(v)\}_{v \in V_s} = \bigcup_{u \in V_a} Reach(\widehat{\mathcal{N}}_{el}, M_a(u)).$$

Proof.
• Let $m \in \bigcup_{s \in Reach(\mathcal{N},s_0)} \{M_s(u)\}_{u \in V_s}$. There exists $s_0 \xrightarrow{\sigma} s$ with some $v \in V_s$ such that $M_s(v) = m$. By Proposition 8.3.8 there is an omniscient sequence in $s_0 \xrightarrow{\widehat{\sigma}}_{\widehat{\mathcal{N}}} s$. We split $\widehat{\sigma}$ into $s_0 \xrightarrow{\widehat{\sigma}_1}_{\widehat{\mathcal{N}}} s_v \xrightarrow{\widehat{\sigma}_2}_{\widehat{\mathcal{N}}}$ where s_v is the state where the thread

v first appears. By definition of the abstract graph, there is some $u \in V_a$ with $M_{s_v}(v) = M_a(u)$. Let $(v, \widehat{\sigma}_2')$, consisting of all firings of v in $\widehat{\sigma}_2$. $(v, \widehat{\sigma}_2')$ is fireable from s_v since $\widehat{\sigma}_2$ is omniscient implying that there will be not cut transition fired by a child of v. By construction of $\widehat{\mathcal{N}}_{el}$, the sequence $\widehat{\sigma}_2'$ is a firing sequence of $(\widehat{\mathcal{N}}_{el}, M_a(u))$ thus $m \in Reach(\widehat{\mathcal{N}}_{el}, M_a(u))$.

• Let $u \in V_a$ and $m \in Reach(\widehat{\mathcal{N}}_{el}, M_a(u))$, i.e. $M_a(u) \xrightarrow{\sigma}_{\widehat{\mathcal{N}}_{el}} m$ for some $n \in \mathbb{N}$. First by Lemma 8.7.2 there exists $s_0 \xrightarrow{\sigma_u}_{\mathcal{N}} s_u$ where for some $v \in V_{s_u}$ we have $M_{s_u}(v) = M_a(u)$. By construction of $\widehat{\mathcal{N}}$, we also have $s_0 \xrightarrow{\sigma_u}_{\widehat{\mathcal{N}}} s_u$. By construction of $\widehat{\mathcal{N}}_{el}$, we get that $s_u \xrightarrow{(v,\sigma)}_{\widehat{\mathcal{N}}} s$ where $M_s(v) = m$. By Proposition 8.3.7, $s \in Reach(\mathcal{N}, s_0)$, which concludes the proof. □

Using the previous Lemma and Rackoff's Theorem, we establish the complexity of the boundedness problem:

Proposition 8.7.4. *The boundedness problem of RPN is* EXPSPACE*-complete.*

Proof. Hardness of the problem comes from hardness of Petri nets. Let $(\mathcal{N}, s_0)$ be a marked RPN. Due to Corollary 8.3.4 we assume w.l.o.g. that $s_0 = s[r, m_0]$. By Lemma 8.7.3 checking whether $(\mathcal{N}, s_0)$ is bounded is equivalent to whether for $v \in V_a$, $(\widehat{\mathcal{N}}_{el}, M_a(u))$ is bounded which, due to Rackoff, can be performed in exponential space. □

Let $(\mathcal{N}, s_0)$ be a marked RPN. If $s_0 = \emptyset$ then the number of reachable states is 1, hence from now on we assume that $s_0 \neq \emptyset$. Next, if there exists $t \in T_{ab}$ with $W^-(t) = 0$ then there are infinitely many reachable states since one can fire t repeatedly, which provides us with a sequence of states with an unbounded number of threads. Therefore, from now on we assume that for all $t \in T_{ab}$, $W^-(t) > 0$.

We now establish a connection between the boundedness of $\widehat{\mathcal{N}}_{el}$ and the maximal number of children of the root in $\mathcal{N}$:

Lemma 8.7.5. *Let $\mathcal{N}$ be an RPN such that $(\widehat{\mathcal{N}}_{el}, m_0)$ is bounded. Then:*

$$\sup_{s' \in Reach(\mathcal{N}, s[r,m_0])} |\{v \in V_{s'} \mid r_{s'} \rightarrow_{s'} v\}| < \infty$$

Proof. Assume that there exists a family of sequences $\{\sigma_n\}_{n\in\mathbb{N}}$ such that $s[r, m_0] \xrightarrow{\sigma_n}_{\mathcal{N}} s_n$ and the number of children of r in s_n is greater than n. By Proposition 8.3.8 for all σ_n there exists an omniscient sequence $\widehat{\sigma}_n$ in $\widehat{\mathcal{N}}$ from $s[r, m]$ reaching s_n. We remove from $\widehat{\sigma}_n$ all the transitions not fired from the root getting $(r, \widehat{\sigma}_n')$ which is also fireable from $s[r, m]$ and which leads to a state where the root has a number of children greater than n. Since an abstract transition consumes tokens from the root (for all $t \in T_{ab}$, $W^-(t) > 0$) one can remove them from $(r, \widehat{\sigma}_n')$ and get $(r, \widehat{\sigma}_n'')$

for which $s \xrightarrow{(r,\widehat{\sigma}''_n)}_{\widehat{\mathcal{N}}} s''_n$ and $\sum_{p\in P} M_{s''_n}(r)(p) > n$. Since $\widehat{\sigma}''_n$ is fireable from m in $\widehat{\mathcal{N}}_{el}$ this contradicts the hypothesis of the lemma. □

Combining the above results, we get a characterization of the finiteness problem:

Proposition 8.7.6. *Let $(\mathcal{N}, s_0)$ be a marked RPN. Then $Reach(\mathcal{N}, s_0)$ is finite if and only if the following assertions hold:*
1. There is no loop in $G_{\mathcal{N},s_0} = (V_a, E_a, M_a)$;
2. For all $v \in V_a$, $(\widehat{\mathcal{N}}_{el}, M_a(v))$ is bounded.

Proof.
• Assume that assertions 1 and 2 hold. Due to Assertion 1 and Lemma 8.6.4 any reachable state has its depth bounded by some constant. Due to Assertion 2 and Lemmas 8.7.3 and 8.7.5 each thread in any reachable state has a bounded number of children, and a bounded number of different reachable markings. Therefore, $Reach(\mathcal{N}, s_0)$ is finite.
• Assume that Assertion 1 does not hold. By Lemma 8.6.4 there is a deep infinite sequence. Hence, there is an infinite sequence of states with growing depth. Therefore, $Reach(\mathcal{N}, s_0)$ is not finite.
• Assume that Assertion 2 does not hold for some vertex v. By Lemma 8.7.2 there exists a state $s \in Reach(\mathcal{N}, s_0)$ and a vertex $u \in V_s$ such that $M_s(u) = M_a(v)$. By the definition of $\widehat{\mathcal{N}}_{el}$, for any $m \in Reach(\widehat{\mathcal{N}}_{el}, M_s(v))$, there exists a firing sequence (r, σ') in $\widehat{\mathcal{N}}$ such that $s \xrightarrow{(r,\sigma')}_{\widehat{\mathcal{N}}} s'$ with $M_{s'}(v) = m$. Since $Reach(\widehat{\mathcal{N}}, s) \subseteq Reach(\widehat{\mathcal{N}}, s_0)$, $(\widehat{\mathcal{N}}, s_0)$ is unbounded. Due to Proposition 8.3.7, $Reach(\widehat{\mathcal{N}}, s_0) = Reach(\mathcal{N}, s_0)$ and thus $(\mathcal{N}, s_0)$ is unbounded. □

Theorem 8.7.7. *The finiteness problem of RPN is* EXPSPACE*-complete.*

Proof. The algorithm proceeds by checking Assertions 1 and 2 of Proposition 8.7.6. It builds in exponential space (by Lemma 8.6.3) the abstract graph and checks whether there is no loop in $G_{\mathcal{N},s_0}$. In the negative case, for any vertex $v \in V_a$ it checks in exponential space, whether the marked Petri net $(\widehat{\mathcal{N}}_{el}, M_a(v))$ is bounded. □

Chapter 9

Dynamic Recursive Petri Nets

9.1 Introduction

While having a great expressive power, RPN suffer several limitations: (1) the elementary transitions do not include more general features like reset arcs, transfer arcs, ... that preserve the decidability of the coverability problem; (2) the firing rule of an abstract transition is somewhat limited: the initial marking associated with the creation of a thread is constant, i.e. it does not depend on the current marking of the thread that fires it. So we introduce *Dynamic Recursive Petri nets* (DRPN) which address these issues while preserving the decidability of the coverability (and cut) problem.

Dynamic recursive Petri nets. In order to unify the possible extensions that could be imagined, we define in a generic way a DRPN. We associate with a DRPN over the set of places P, a set of non decreasing partial functions $\mathcal{F}$ from $\mathbb{N}^P$ to $\mathbb{N}^P$. The possible effects of a transition on the current marking and the initial marking of a thread created by the firing of an abstract transition are specified by functions of $\mathcal{F}$. Furthermore, the precondition of an elementary or abstract transition is specified by the domain of the function that updates the marking of the thread that fires the transition while the (upward closed) precondition of a cut transition is specified by the minimal elements of this set of markings. Rather than providing a syntactical definition of $\mathcal{F}$, we have chosen to require some effectiveness properties of this set. Such an approach lets open the possibility to choose an appropriate set $\mathcal{F}$ for modelling needs. In particular, we show that Petri nets, Affine nets and Recursive Petri nets can be redefined in terms of DRPN. We also illustrate the modelling power of DRPN through a small but relevant example.

Expressiveness. We generalize the qo defined for RPN to DRPN, and we show that the family of coverability languages of DRPN is strictly greater than the one of RPN. This result is robust in the following sense: we provide two different

languages each one using in a somewhat minimal way a feature specific to DPRNs. The first language is generated by a DRPN without abstract transitions and only using a single 'non-standard' function (which is sublinear, see later on). The second language is generated by a DRPN for which all transitions but a single abstract transition can be specified in the RPN formalism.

Coverability. We show that the coverability problem is decidable. In order to solve the coverability problem, we provide a reduction of this problem to the cut problem of a DRPN without abstract transitions. Since we establish that a DRPN without abstract transitions is a WSTS, the conclusion follows. The complexity of this problem should be studied by fixing some possible $\mathcal{F}$ (as it is done for RPN in the previous chapter) and we let here this open issue as a perspective to this work.

Organization. In section 9.2 we introduce DRPN, illustrate their modelling capabilities and define a qo between states of DRPN. In section 9.3, we study from a theoretical point of view, expressiveness of DRPNs. In section 9.4, we establish that the coverability problem is decidable.

Based on. This chapter is base on our work in [69].

9.2 Dynamic Recursive Petri Nets

Since DRPN is an extension of RPN, let us first point out their similarities and differences.

Similarities. From a syntactical point of view as in RPN, a DRPN includes a set of places P and a set of transitions T, including three types of transitions: elementary, abstract, and cut ones. From a semantical point of view, states are still represented by a tree structure and the effects of firing a transition are similar:

- The firing of an elementary transition updates the marking of the thread firing it;
- The firing of an abstract transition updates the marking of the thread firing it and creates a new child to the firing thread;
- The firing of a cut transition removes the subtree rooted in the thread which fired it, and updates the marking of its parent.

Differences. In DRPN, the initial marking of a new thread not only depends on the abstract transition that created it, but also on the marking of the thread that created it. The functions involved in the marking changes or the marking initialization are picked from a set of non decreasing (partial) functions which is not predefined but must satisfy some properties (see later on). In order to be more precise, we associate with every transition t a finite set of markings Gd_t. Then we

associate (1) with an elementary transition t a mapping (of markings) Up_t whose domain is $\uparrow Gd_t$, (2) with an abstract transition t three mappings Up_t^-, Bg_t and Up_t^+ where Up_t^+ is a total mapping and $\uparrow Gd_t$ is the domain of Up_t^- and Bg_t. The following table exhibits the operational rules for marking changes in RPN and DRPN (the mapping **Id** denotes the identity mapping where the domain should be clear from the context).

	RPN	DRPN
t- *elementary* transition, m- marking of v.	$m \geq W^-(t)$ $m \xrightarrow{v,t} m + W^+(t) - W^-(t)$	$m \in \uparrow Gd_t$ $m \xrightarrow{v,t} Up_t(m)$
t- *abstract* transition, m- marking of v, m'- marking of the new child of v.	$m \geq W^-(t)$ $m \xrightarrow{v,t} m - W^-(t)$ $m' = \Omega(t)$	$m \in \uparrow Gd_t$ $m \xrightarrow{v,t} Up_t^-(m)$ with $Up_t^- \leq \mathbf{Id}$ $m' = Bg_t(m)$
t- *cut* transition, m- marking of v. m'- marking v's parent, t'- transition which created v.	$m \geq W^-(t)$ $m' \xrightarrow{v,t} m' + W^+(t')$	$m \in \uparrow Gd_t$ $m' \xrightarrow{v,t} Up_{t'}^+(m')$ with $Up_{t'}^+ \geq \mathbf{Id}$

Observe that the DRPN expressiveness and modeling power depend strongly on the family of functions it is allowed to use for its transitions. Our interest is to define this family to be as large as possible, while still preserving decidable properties. To this end we will introduce our notion of *effective* functions. Let us first recall a couple of notions from order theory (see Chapter 2.1). Let $Y \subset \mathbb{N}^P$ (for some finite set P), be an upward closed set (with the usual order on $\mathbb{N}^P$). There exists a set of it minimal elements, denoted by $\min(Y)$, i.e. the minimal finite set fulfilling $Y = \uparrow \min(Y)$. Observe that, given a non-decreasing function $f : \mathbb{N}^P \mapsto \mathbb{N}^P$, the set $f^{-1}(Y)$ is upward closed.

Definition 9.2.1 (Effective function)**.** Let $(X, \leq)$ be a wqo, and $f : X \mapsto X$ a non-decreasing recursive partial function. We say that a f is *effective* if there exists an algorithm computing, for any $x \in X$ the set $\min(f^{-1}(\uparrow x))$.

Note that, for an effective function $f : \mathbb{N}^P \mapsto \mathbb{N}^P$ and any $x \in \mathbb{N}^P$, it is decidable whether $x \in dom(f)$, since $dom(f)$ is upward closed, and we have an algorithm computing:

$$\min\left(f^{-1}(\uparrow \mathbf{0})\right) = \min\left(f^{-1}(\uparrow \min(\mathbb{N}^P))\right) = \min\left(f^{-1}(\mathbb{N}^P)\right) = \min(dom(f))$$

We are now ready to give the full definition of DRPN:

Definition 9.2.2 (DRPN)**.** A *Dynamic Recursive Petri Net* is a 6-tuple $\mathcal{N} = \langle P, T, Gd, Up, Up^-, Up^+, Bg \rangle$ where:

- P is a finite set of places;
- T is a finite set of transitions such that $P \cap T = \emptyset$. This set consists of three type of transitions: elementary, abstract, and cut transitions, i.e. $T = T_{el} \uplus T_{ab} \uplus T_{\tau}$;
- $Gd = \{Gd_t\}_{t \in T}$ is a family of finite sets of markings with $Gd_t \subseteq \mathbb{N}^P$;
- $Up = \{Up_t\}_{t \in T_{el}}$ is a family of effective functions with $Up_t \in (\mathbb{N}^P)^{\uparrow Gd_t}$;
- $Up^- = \{Up_t^-\}_{t \in T_{ab}}$ is a family of effective functions with $Up_t^- \in (\mathbb{N}^P)^{\uparrow Gd_t}$;
- $Up^+ = \{Up_t^+\}_{t \in T_{ab}}$ is a family of effective functions with $Up_t^+ \in (\mathbb{N}^P)^{\mathbb{N}^P}$;
- For all $t \in T_{ab}$, we have $Up_t^- \leq \mathbf{Id} \leq Up_t^+$;
- $Bg = \{Bg_t\}_{t \in T_{ab}}$ is a family of effective functions with $Bg_t \in (\mathbb{N}^P)^{\uparrow Gd_t}$.

Most of the operational semantic of a DRPN mimics the one of RPN. However, in order for this chapter to be self content, we repeat the similar parts of the semantic. A concrete state s of a DRPN is a labeled tree representing relations between threads and their associated markings. Every vertex of s is a thread, and edges are labeled by functions from Up^+. We introduce a countable set $\mathcal{V}$ of vertices in order to pick new vertices when necessary.

Definition 9.2.3 (State of a DRPN)**.** A *concrete state (in short, a state)* s of a DRPN is a tree over the finite set of vertices $V_s \subseteq \mathcal{V}$, inductively defined as follows:

- either $V_s = \emptyset$ and thus $s = \emptyset$ is the empty tree;
- or $V_s = \{r_s\} \uplus V_1 \uplus \ldots \uplus V_k$ with $0 \leq k$ and $s = (r_s, m_0, \{(f_i, s_i)\}_{1 \leq i \leq k})$ is defined as follows:
  - r_s is the root of s labeled by a marking $m_0 \in \mathbb{N}^P$;
  - For all $i \leq k$, s_i is a state over $V_i \neq \emptyset$ and there is an edge $r_s \xrightarrow{f_i}_s r_{s_i}$ with $f_i \in Up^+$.

For all $u, v \in V_s$, one denotes $M_s(u)$ the marking labeling u and when $u \xrightarrow{f}_s v$, one writes $\Lambda(u, v) := f$. State s_v is the (maximal) subtree of s rooted in v.

While the set of vertices V_s will be important for analyzing the behavior of a firing sequence in a DRPN, one can omit it and get a more abstract representation of the state. Note that contrary to the previous definition where $\{(f_i, s_i)\}_{1 \leq i \leq k}$ was a set, in the following definition we need a multiset $Child_s$.

Definition 9.2.4 (Abstract state of a DRPN)**.** An *abstract state* s of a DRPN is inductively defined as follows:

- either $s = \emptyset$ is the empty set ;
- or $s = (m_s, Child_s)$ where $m_s \in \mathbb{N}^P$ and $Child_s$ is a finite multiset of pairs (f', s')
  where $f' \in Up^+$ and s' is an abstract state different from $\emptyset$.

Given a concrete state s, we denote by $[s]$ its abstract state. Except if explicitly stated, a state is a concrete state.

In the other direction, given an abstract state s, one recovers its set of concrete states by picking an arbitrary set of vertices $V_s \subseteq \mathcal{V}$ of appropriate cardinality and, inductively, arbitrarily splitting V_s between the root and the pairs (m, s').

One denotes by $Des_s(v)$ (respectively $Anc_s(v)$) the set of descendants (respectively ancestors) of $v \in V$ in the underlining tree of s (including v itself). If $v \neq r$ then $prd(v)$ is the parent of v in the tree. The *depth* of s is the depth of its tree. Given $\mathbf{m} \in \mathbb{N}^P$, $s_{\mathbf{m}}$ denotes a tree consisting of a single vertex r with marking $M(r) = \mathbf{m}$.

Let us formally define the firing of elementary, abstract and cut transitions.

Definition 9.2.5 (Operational semantics)**.** Let $s = (r, m_0, \{(f_i, s_i)\}_{1 \leq i \leq k})$ be a state. Then the firing rule $s \xrightarrow{(v,t)} s'$ where $v \in V_s$ and $t \in T$ is inductively defined as follows:

- Let $t \in T_{el}$ such that $m_0 \in \uparrow Gd(t)$, then one has $s \xrightarrow{(r,t)} (r, Up_t(m_0), \{(f_i, s_i)\}_{i \leq k})$
- Let $t \in T_{ab}$ such that $m_0 \in \uparrow Gd(t)$, then one has $s \xrightarrow{(r,t)} (r, Up_t^-(m_0), \{(f_i, s_i)\}_{i \leq k+1}))$ where $f_{k+1} = Up_t^+$, $s_{k+1} = s[v, Bg_t(M(v))]$ with $v \in \mathcal{V} \setminus V_s$
- Let $t \in T_\tau$ such that $m_0 \in \uparrow Gd(t)$, then one has $s \xrightarrow{(r,t)} \emptyset$
- Let $i \leq k$ such that $s_i \xrightarrow{(v,t)} s_i'$
  if $s_i' = \emptyset$ then $s \xrightarrow{(v,t)} (f_i(m_0), \{(f_j, s_j)\}_{1 \leq j \neq i \leq k})$
  else $s \xrightarrow{(v,t)} (m_0, \{f_j, s_j\}_{1 \leq j \neq i \leq k} \cup \{f_i, s_i'\})$

The transition firing is denoted $s \xrightarrow{(v,t)} s'$ and when there are several nets, $s \xrightarrow{(v,t)}_{\mathcal{N}} s'$. A *firing sequence* is a sequence of transition firings, written in detailed way: $s_0 \xrightarrow{(v_1,t_1)} s_1 \xrightarrow{(v_2,t_2)} \cdots \xrightarrow{(v_n,t_n)} s_n$, or when the context allows it, in a more concise way like $s_0 \xrightarrow{\sigma} s_n$ for $\sigma = (v_1, t_1)(v_2, t_2) \ldots (v_n, t_n)$. The *length* of σ, denoted $|\sigma|$, is n. The *abstract length* of σ, denoted $|\sigma|_{ab}$, is $|\{i \leq n \mid t_i \in T_{ab}\}|$. The *depth* of σ is the maximal depth of states $s_0, \ldots, s_n$. A *cut sequence* is a firing sequence that reaches $\emptyset$. Given a firing sequence that includes the firing of an abstract transition t in vertex v creating vertex w and followed later by the cut transition t_τ in w, we say that $(v, t), (w, t_\tau)$ are *matched* in σ.

Remark. In the sequel, when we write "DRPN $\mathcal{N}$", we mean $\mathcal{N} = \langle P, T, Gd, Up, Up^-, Up^+, Bg \rangle$, unless we explicitly write differently. A DRPN $\mathcal{N}$ equipped with an initial state s is a *marked RPN* and denoted $(\mathcal{N}, s)$.

For a marked DRPN $(\mathcal{N}, s_0)$, let $Reach(\mathcal{N}, s_0) = \{[s] \mid \exists \sigma \in T^* \text{ s.t. } s_0 \xrightarrow{\sigma} s\}$ be its *reachability set*, i.e. the set of all the reachable *abstract* states.

Discussion. The main limitations of the modeling power of DRPN are the requirements that (1) the prerequisite for firing a transition are upward closed sets $\uparrow Gd_t$ and (2) functions like Up_t must be non-decreasing. Despite these limitations, DRPNs include many models like:

Petri nets. Given a Petri net $\mathcal{N} = \langle P, T, \mathbf{Pre}, \mathbf{C} \rangle$ we can be viewed as a DRPN $\mathcal{N}' = \langle P, T, Gd, Up, \emptyset, \emptyset, \emptyset \rangle$, where for all $t \in T = T_{el}$, $Gd_t = \{\mathbf{Pre}(t)\}$ and $Up_t = \mathbf{Id} + \mathbf{C}(t)$.

Affine Petri nets. (defined in Section 6.2) Given an Affine Petri nets $\mathcal{N} = \langle n, F \rangle$ we can simulate it by the DRPN $\mathcal{N}' = \langle P, T, Gd, Up, \emptyset, \emptyset, \emptyset \rangle$ where $P = \{i\}_{0<i\leq n}$, $T = T_{el} = F$. For all $t \in T$, where $t(\mathbf{m}) = A_t\mathbf{m} + B_t$ for $A_t \in \mathbb{N}^{n\times n}$ and $B_t \in \mathbb{Z}^{n\times n}$, we have $Gd_t = \min\{\mathbf{m} \mid A_t\mathbf{m} + B_t \geq 0\}$ and $Up_t(\mathbf{m}) = A_t\mathbf{m} + B_t$.

Recursive Petri nets. Given an Recursive Petri nets $\mathcal{N} = \langle P, T, W^+, W^-, \Omega \rangle$ we can simulate it by the DRPN $\mathcal{N}' = \langle P, T, Gd, Up, Up^-, Up^+, Bg \rangle$, where:

- $t \in T$, $Gd_t = \{W^-(t)\}$
- $t \in T_{el}$, $Up_t = \mathbf{Id} + W^+(t) - W^-(t)$
- $t \in T_{ab}$, $Up_t^- = \mathbf{Id} - W^-(t)$, $Up_t^+ = \mathbf{Id} + W^+(t)$
- Bg_t is constant and equal to $\Omega(t)$

Graphical representation.

For modeling purposes, we equip DRPN with a graphical representation based on net representations. Places (resp. transitions) are depicted by circles (resp. rectangles). However, a transition does not have input arcs, but only output arcs represented by diamond-headed arrows and labeled by expressions where a place represents the current value of its marking. The guard of an elementary transition is represented by the set of markings inside the rectangle (we omit the curly brackets in the figures, for simplicity). For instance, the elementary transition t illustrated in Figure 9.1 is defined by: $Gd_t = \{3p_1\}$ and $Up_t(\mathbf{m}) = (\mathbf{m}(p_1) + \mathbf{m}(p_2))p_1 + \lfloor\sqrt{\mathbf{m}(p_2)}\rfloor p_2$. Note that, in the illustration of DRPNs, we denote the current number of tokens in a place p, i.e. $\mathbf{m}(p)$, by $\overline{p}$.

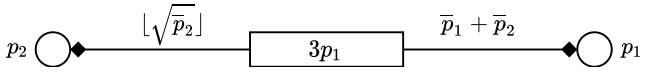


Figure 9.1: An elementary transition in DRPN

The rectangle of an abstract transition t is divided into several parts: on the top corner left $(-)$ starts the edges representing Up_t^-, on the top center Gd_t is represented, on the bottom corner left Bg_t is represented, and on the bottom corner right $(+)$ start the edges representing Up_t^+. We don't plot edges for unchanged

places. For instance, the abstract transition t in 9.2 is defined by: $Gd_t = \{3p_1 + p_3, 2p_2\}$, $Up_t^-(\mathbf{m}) = (\mathbf{m}(p_1)-1)p_1 + \lfloor 0.5\mathbf{m}(p_2)\rfloor p_2 + \mathbf{m}(p_3)p_3$, $Up_t^+(\mathbf{m}) = \mathbf{m}(p_1)p_1 + 2\mathbf{m}(p_2)p_2 + \mathbf{m}(p_3)^2 p_3$, and $Bg_t = \mathbf{m}(p_1)p_2$. Observe that $Up_t^- \leq \mathbf{Id} \leq Up_t^+$.

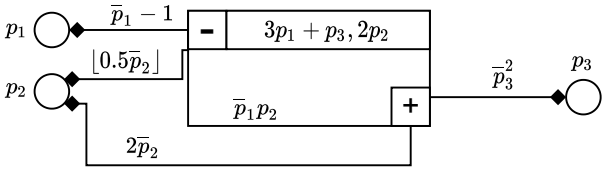


Figure 9.2: An abstract transition in DRPN

Finally, a cut transition τ in DRPN is represented with a rectangle filled with gray surrounding Gd_τ. For example see the cut transition τ in Figure 9.3 where the $Gd(\tau) = \{2p_1, p_2\}$.

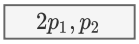


Figure 9.3: A cut transition in DRPN

Example 9.2.6 (Hiring an assassin)**.** In order to illustrate the modeling capabilities of DRPN, we present an example of distributed planning. The DRPN $\mathcal{N}_{Jaqen}$ of Figure 9.4 represents the possible behavior of an assassin hired for a job. The transitions presented with ordinary input and output arcs are RPN (and thus DRPN) transitions.

The assassin is given 3 days (3 tokens in p_{time}), an advance of 20 bitcoins (20 tokens in p_{adv}), and is promised to get a reward of 20 bitcoins after the job is done (20 tokens in p_{reward}). In order to try catching his target, he needs to devote one bitcoin and one day of his time. After this day either the assassin is successful (t_{found}) or fails (t_{lost}) and needs to spend another day. When successful, the assassin can collect the reward ($t_{collect}$). However, the assassin has also another strategy which consists of hiring another assassin by giving him a quarter of his advance money and promise him an equal reward, telling him the number of days left (t_{hire} where $f_{pay}(\mathbf{m}) = \mathbf{m}(p_{time})p_{time} + \lfloor 0.5\lceil 0.5\mathbf{m}(p_{adv})\rceil\rfloor p_{adv} + \lceil 0.5\lceil 0.5\mathbf{m}(p_{adv})\rceil\rceil p_{reward}$). If some hired assassin is successful, he can report his success to the hiring guy by firing the cut transition t_{report}. The state presented on the right of Figure 9.4 consists of three assassins where the last hired one has "found" the target. Observe that as long as an assassin has money he can hire other assassins and that even after hiring he can still try to "find" the target by himself.

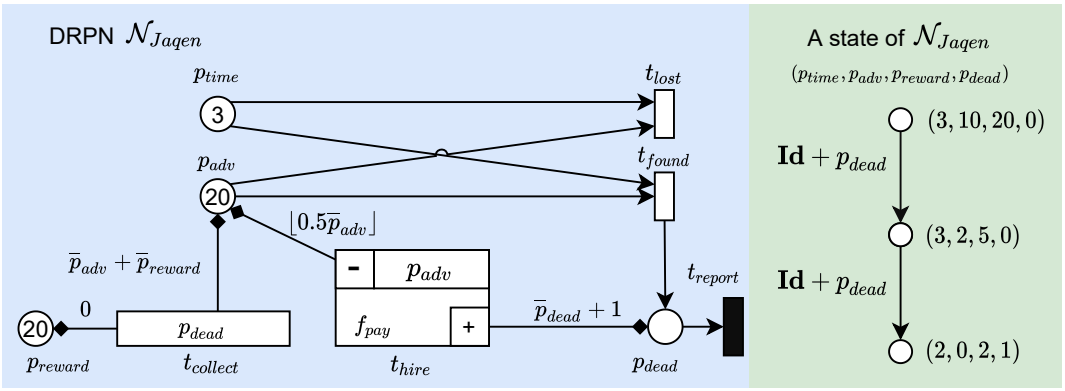


Figure 9.4: The Jaquen DRPN with a state

A firing sequence of $\mathcal{N}_{Jaqen}$ is presented in Figure 9.5 where the vertex that fires the transition is filled. The initial assassin first tries to find the target but fails (i.e. firing (r, t_{lost})), losing one bitcoin and one day. Then he hires another assassin by firing the abstract transition (r, t_{hire}), losing half of his advance money and creating a new vertex v, where the hired assassin has two days, an advance of five bitcoins and a promised reward of five bitcoins for completing the job : $M(v) = 2p_{time} + 5p_{adv} + 5p_{reward}$. This assassin kills the target and collects the reward ((v, t_{lost}) followed by $(v, t_{collect})$). Then using a cut transition he reports it to his employer (v, t_{report}), which removes v and adds one token to p_{dead} in $M(r)$. Finally, the original assassin can collect his money by firing $t_{collect}$.

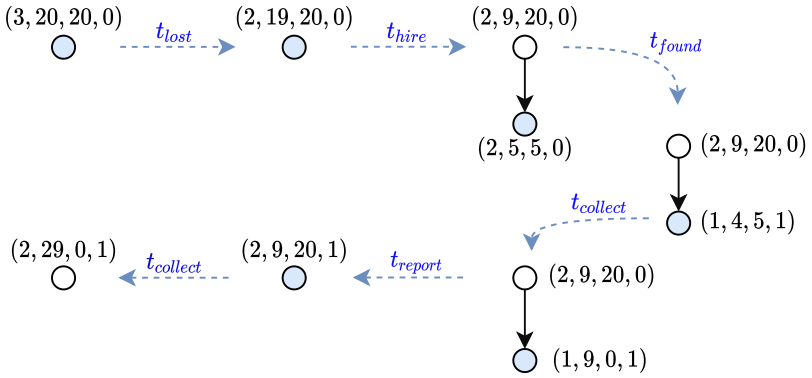


Figure 9.5: A firing sequence in Jaquen DRPN

Ordering states of a DRPN. We now define a qo on the states of a DRPN.

This qo is an extension to the one we had for states of RPN, where instead of comparing markings on the edges, we compare functions. More precisely, we assume a decidable order $\preceq$ between functions of Up^+ such that for all f, f' and $m \in \mathbb{N}^P$ $f \preceq f'$ implies $f(m) \leq f'(m)$. There always exists such an order, namely '=' (syntactical equality). Furthermore, for most of the families Up^+, '$\leq$' is decidable and more appropriate. Given two states s, s' of $\mathcal{N}$, we say that s is smaller or equal than s' (or equivalently that s' *covers* s) if there is a subtree in s' isomorphic to s (by some matching) such that (1) given any pair of matched vertices (u, u'), $M(u) \leq M'(u')$ and (2) given any pair of functions (f, f') labeling matched edges, $f \preceq f'$.

Definition 9.2.7. Let $\preceq$ and $\preceq_\varphi$ be the qo defined as follows. Let $s, s' \neq \emptyset$ be states of a DRPN $\mathcal{N}$ and let φ be an injective mapping from V_s to $V_{s'}$.
If for all $v \in V_s$:

1. $M_s(v) \leq M_{s'}(\varphi(v))$, and,
2. for all $v \xrightarrow{f}_s w$, there exists an edge $\varphi(v) \xrightarrow{f'}_{s'} \varphi(w)$ with $f \preceq f'$.

then $s \preceq_\varphi s'$.

We define $s \preceq s'$ if there exists an injective mapping φ from V_s to $V_{s'}$ such that $s \preceq_\varphi s'$.
In addition, we set $\emptyset \preceq s$ for all states s.

Observe that with our assumption, this order is decidable.

The following lemma shows that the relation defined in 9.2.7 on states of DRPN is a qo and is strongly compatible. We leave out its proofs since they are similar to the ones for lemmas 8.2.2 and 8.2.3.

Lemma 9.2.8. *The relation $\preceq$ is a qo and strongly compatible.*

However, just like in RPN this qo is not a wqo and thus in order to solve the coverability problem, one cannot apply the backward exploration since it would not necessarily terminate.

Recall, the *coverability set* $Cov(\mathcal{N}, s_0)$ is defined as the downward closure of the reachability set:

$$Cov(\mathcal{N}, s_0) = {\downarrow} Reach(\mathcal{N}, s_0)$$

9.3 Expressiveness

In Section 8.4, the expressiveness of RPNs was studied using coverability languages. In order to compare RPN and DRPN expressiveness we recall the definition of the family of coverability languages of DRPN. Equip any transition $t \in T$ with a *label* $\lambda(t) \in \Sigma \cup \{\varepsilon\}$. The labeling is extended to transition sequences in the usual way.

Thus given a labeled marked DRPN $(\mathcal{N}, s_0, \lambda)$ and a finite set of states S_t, the coverability language $\mathcal{L}_C(\mathcal{N}, s_0, \lambda, S_t)$ is defined by:

$$\mathcal{L}_C(\mathcal{N}, s_0, \lambda, S_t) = \{\lambda(\sigma) \mid \exists s \in S_t, \exists s' \in S_t;\ s_0 \xrightarrow{\sigma} s \wedge s \succeq s'\}$$

In Section 8.4, we showed that the family of RPN coverability languages is equal to the family of RPN cut languages (propositions 8.4.1 and 8.4.2). The proof also works for DRPN (using the RPN simulations).

Proposition 9.3.1. *The family of cut languages of DRPNs is equal to the family of coverability languages of DRPNs.*

Therefore, in this section, we will be talking about coverability and cut languages for DRPN interchangeably. Moreover, due to Proposition 8.4.10 we can assume that the initial markings of the RPNs have a single vertex.

In order to establish the difference between RPN and DRPN languages, we need to define a new type of RPN languages, in which we are only allowed to only fire a bounded number of abstract transitions. Formally, given $B \in \mathbb{N}$, let $\mathcal{L}_R(\mathcal{N}, s_0, \lambda, \emptyset)$ be the *B-bounded cut language*, i.e.

$$\mathcal{L}_B(\mathcal{N}, s_0, \lambda, \emptyset) = \{\lambda(\sigma) \mid \exists \sigma;\ s_0 \xrightarrow{\sigma} \emptyset \wedge |\sigma|_{ab} \leq B\}$$

We say that $\mathcal{L} \subseteq \Sigma^*$ is a *B-bounded cut language* if $\mathcal{L} = \mathcal{L}_B(\mathcal{N}, s_0, \lambda, S_t)$ for some $\mathcal{N}$, s_0, λ and S_t.

9.3.1 Bounded RPN and Petri net languages equivalence

In this subsection, we show how to build a Petri net with same language as a B-bounded cut language of an RPN.

Let $\mathcal{L} = \mathcal{L}_B(\mathcal{N}, [r, \mathbf{m}], \lambda, \emptyset)$ be a B-bounded cut language with the RPN $\mathcal{N} = \langle P, T, W^+, W^-, \Omega\rangle$. Without loss of generality, we assume that for any $t, t' \in T_{ab}$ we have $W^+(t) \neq W^+(t')$. We build a Petri net $\mathcal{N}^B = (P^B, T^B, \mathbf{Pre}^B, \mathbf{C}^B)$, with an initial marking $\mathbf{m}_B$, a target marking $p_\emptyset$ and a labeling function for its transitions λ^B, such that $\mathcal{L} = \mathcal{L}_C(\mathcal{N}^B, \mathbf{m}_B, \lambda^B, p_\emptyset)$. $\mathcal{N}^B$ has a structure which represents the super set of all possible states reachable by firing sequences in $\mathcal{N}$ containing at most B abstract transitions. This is achieved by copying the structure of the RPN in a tree like structure. Each vertex in this tree represents a possible thread in the RPN. To each of these vertices we match a set of places reflecting the threads marking, its connection to its predecessor and its control state. Each vertex can be in one of three control states "sleeping", "running", or "dead". Note that a "running" vertex cannot become "sleeping" vertex and a "dead" one cannot become "running". We now describe in details the construction of $\mathcal{N}^B$. Figure 9.6 illustrates such a construction for $B = 2$.

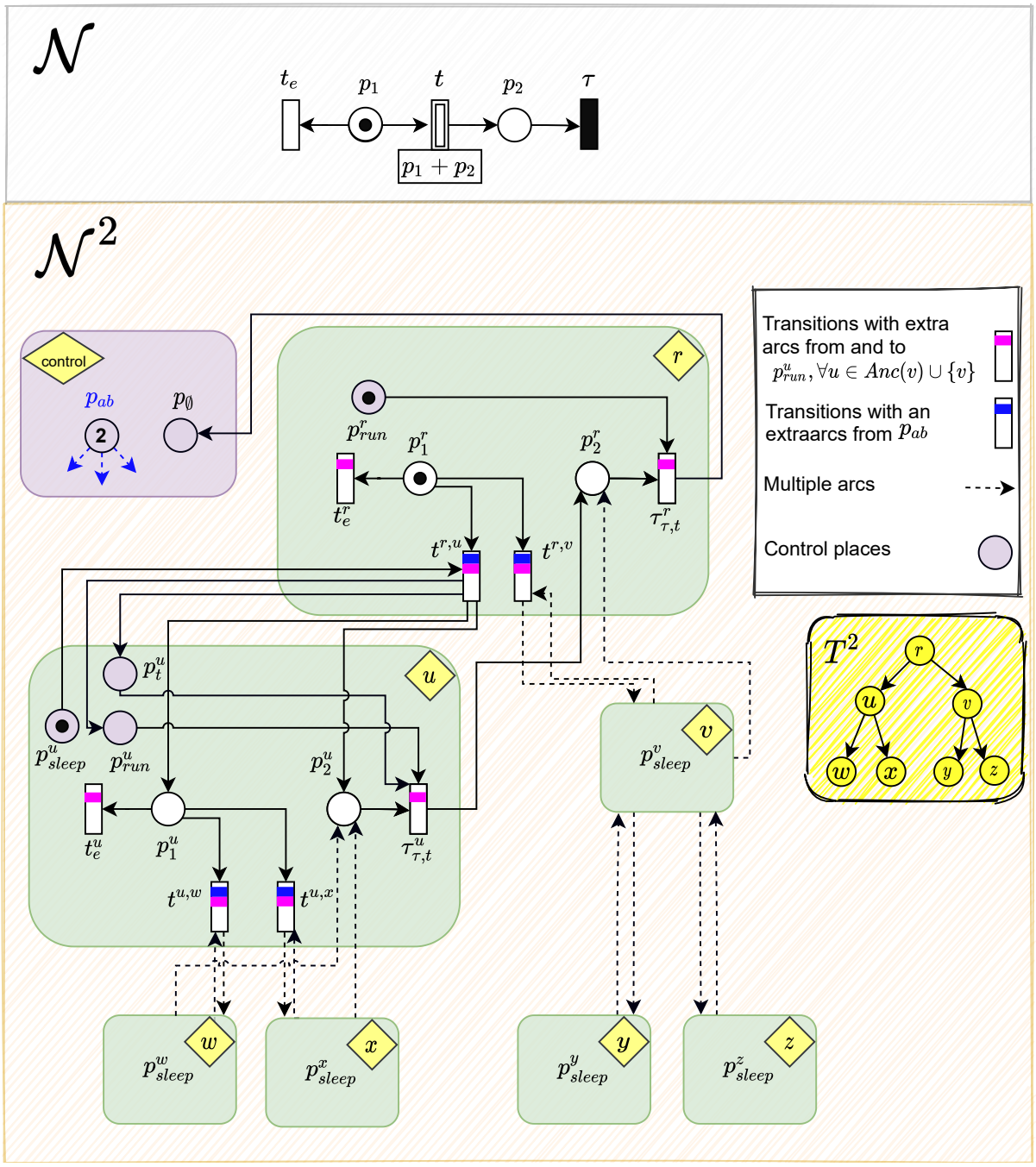


Figure 9.6: The RPN $\mathcal{N}$ and it Petri net $\mathcal{N}^B$ for $B = 2$. $\mathcal{N}^B$ is split into two types of blocks: a control block and a block per possible thread. The second type of block includes two types of places: control places (e.g. p^u_{run} and p^u_t) and places (e.g. p^u_1 and p^u_2) simulating the marking of the thread.

Let $T^B = (E^B, V^B)$ be a rooted and directed tree of depth B, for which every internal node has B children. Denote its root by r_B. For each node, $v \in V^B$ define a set of places P^v and a set of transitions T^v. The set P^v for $v \neq r_B$ includes four types of places:

$$P^v = \{p^v \mid p \in P\} \uplus \{p^v_t \mid t \in T_{ab}\} \uplus \{p^v_{sleep}\} \uplus \{p^v_{run}\}$$

- p^v for every place p in the RPN $\mathcal{N}$
- p^v_t for every abstract transition t (marked if v was "woken up" by the transition t)
- p^v_{sleep} (marked if this vertex is "still sleeping")
- p^v_{run} (marked if this vertex is "running")

For $v = r_B$ we have $P^{r_B} = \{p^{r_B} \mid p \in P\} \uplus \{p^{r_B}_{run}\}$ since r_B corresponds to the initial thread. For an example, see the set of places for the vertex u in Figure 9.6:

$$P^u = \{p^u_1, p^u_2\} \uplus \{p^u_t\} \uplus \{p^u_{sleep}\} \uplus \{p^u_{run}\}$$

For any node $v \neq r_B$, the set T^v includes three types of transitions:

$$T^v = \{t^v \mid t \in T_{el}\} \uplus \{t^{v,w} | t \in T_{ab}, prd(w) = v\} \uplus \{\tau^v_{\tau,t} \mid \tau \in T_\tau, t \in T_{ab}\}$$

Note that, for any leaf v, there is no transition of the second type, i.e. $T^v = \{t^v \mid t \in T_{el}\} \uplus \{\tau^v_{\tau,t} \mid \tau \in T_\tau, t \in T_{ab}\}$.

For $v = r_B$, we have a difference for the cut transitions, since it was not "created" by a transition. We have

$$T^{r_B} = \{t^{r_B} \mid t \in T_{el}\} \uplus \{t^{r_B,w} | t \in T_{ab}, prd(w) = r_B\} \uplus \{\tau^{r_B}_\tau \mid \tau \in T_\tau\}$$

For an example, the set of transitions for vertex u in Figure 9.6 is:

$$T^u = \{t^u_e\} \uplus \{t^{u,w}, t^{u,x}\} \uplus \{\tau^u_{\tau,t}\}$$

Lastly, we have two extra special places, $p_\emptyset$ denoting the empty tree, and p_{ab} in charge of limiting the number of "abstract transition firings" in $\mathcal{N}^B$. Combining all we get:

$$P^B = \{p_\emptyset, p_{ab}\} \uplus \bigcup_{v \in V^B} P^v \; ; \quad T^B = \bigcup_{v \in V^B} T^u$$

Before describing $\mathbf{Pre}^B$ and $\mathbf{C}^B$ let us introduce a few useful notations. Given $v \in V^B$ and a marking (of a thread of $\mathcal{N}$) $\mathbf{m} = \sum_{p \in P} a_p \cdot p$, denote by $[\mathbf{m}]^v = \sum_{p \in P} a_p \cdot p^v$, the translation of the marking to vertex v of $\mathcal{N}^B$. Denote by $\mathbf{m}_{\mathsf{Anc}}(v) = \sum_{u \in \mathsf{Anc}(v)} p^u_{run}$, such that if $\mathbf{m} \geq \mathbf{m}_{\mathsf{Anc}}(v)$, then v and all its ancestors are "running".

We are now ready to construct $\mathbf{Pre}^B$ and $\mathbf{C}^B$ for each $t \in T^B$. Note that, all transitions in $\mathcal{N}^B$ belong to some vertex v. First, A general rule for all transitions is that if the vertex they "belong" or any of its ancestors (including itself) are not running (i.e. $\mathbf{m} \geq \mathbf{m}_{\mathsf{Anc}}(v)$) then it is not fireable. Now, we describe in details the four types of transitions:

- The transition t^v is fireable if it is fireable in v (i.e. $\mathbf{m} \geq [W^-(t)]^v$). If fired, it only updates the marking of places in vertex v according to W^+ and W^-:

$$\mathbf{Pre}^B(t^v) = \mathbf{m}_{\mathsf{Anc}}(v) + \left[W^-(t)\right]^v \ ; \ \mathbf{C}^B(t^v) = \left[W^+(t) - W^-(t)\right]^v$$

  See for example transitions t^u_e in Figure 9.6.
- The transition $t^{v,w}$ is fireable if (1) t is fireable in v (i.e. $\mathbf{m} \geq [W^-(t)]^v$), (2) $\mathcal{N}^B$ can still fire "abstract transitions" (i.e. $p_{ab} > 0$), and (3) the vertex w was not "woken up" yet (i.e. $p^w_{sleep} > 0$). Firing it (1) "wakes up" w by putting a token in p^w_{run}, removing the token in p^w_{sleep} and putting a token in p^w_t meaning that it was created by t, 2) removes a token from p_{ab} since it used an "abstract transitions", 3) puts tokens in vertex w according to $\Omega(t)$, and 4) updates the marking on the vertex v according to $W^-(t)$:

$$\begin{aligned}\mathbf{Pre}^B(t^{v,w}) &= \mathbf{m}_{\mathsf{Anc}}(v) + \left[W^-(t)\right]^v + p^w_{sleep} + p_{ab} \ ; \\ \mathbf{C}^B(t^{v,w}) &= p^w_{run} + p^w_t - p^w_{sleep} - p_{ab} + [\Omega(t)]^w - [W^-(t)]^v\end{aligned}$$

  See for example transition $t^{u,w}$ in Figure 9.6.
- The transition $\tau^v_{\tau,t}$ represents the cut steps of v. It is fireable if v was created by the transition t (i.e. $p^v_t > 0$), and has marking greater or equal than $W^-(\tau)$. Its firing removes the token from p^v_{run} ("killing" v) and puts tokens in the places of its parent according to $W^+(t)$:

$$\mathbf{Pre}^B(\tau^v_{\tau,t}) = \mathbf{m}_{\mathsf{Anc}}(v) + [W^-(\tau)]^v + p^v_t \ ; \ \mathbf{C}^B(\tau^v_{\tau,t}) = [W^+(t)]^{prd(v)} - p^v_{run}$$

  See for example transitions $t^u_{\tau,t}$ in Figure 9.6.
- The transition $\tau^{r_B}_{\tau}$ represents a cut step of r_B. It is fireable if the vertex r_B has a marking greater or equal than $W^-(\tau)$. It removes the token in $p^{r_B}_{run}$ making all transitions not fireable, and puts a token in $p_\emptyset$ signifying we reached the "empty state":

$$\mathbf{Pre}^B(\tau^{r_B}_{\tau,t}) = p^{r_B}_{run} + [W^-(\tau)]^{r_B} \ ; \ \mathbf{C}^B(\tau^{r_B}_{\tau,t}) = p_\emptyset - p^{r_B}_{run}$$

  See for example transition $t^{r_B}_{\tau}$ in Figure 9.6.

For translations from a state of $\mathcal{N}$ to a marking of $\mathcal{N}^B$, we need a way to translate a marking on an edge of the state to the transition that created it. Hence, given

a state s of $\mathcal{N}$ and $v \in V_s \setminus \{r\}$, denote by $\Upsilon_s(v) \in T_{ab}$ the transition such that $W^+(\Upsilon_s(v)) = \Lambda_s(prd(v), v)$. Let s be a state of the RPN $\mathcal{N}$ and ϕ an isomorphism from its underlining tree structure to a rooted subtree of T^B. Denote by $M(\phi, s)$ the "translation" of the state s to a marking of $\mathcal{N}^B$, such that if $s \neq \emptyset$:

$$M(\phi, s) = \sum_{u \in V_s} [M_s(u)]^{\phi(u)} + \sum_{u \in \phi(V_s)} p^u_{run} + \sum_{v \in V_s \setminus \{r\}} p^{\phi(v)}_{\Upsilon_s(v)}$$

and $M(\phi, \emptyset) = p_\emptyset$. Recall that $[r, \mathbf{m}_0]$ is the initial state of the marked RPN $\mathcal{N}$, let the initial marking of $\mathcal{N}^B$ be:

$$\mathbf{m}_B = [\mathbf{m}]^{r_B} + p^{r_B}_{run} + \sum_{u \in V^B \setminus \{r_B\}} p^u_{sleep} + B\, p_{ab}$$

Finally, we define the transition labeling of $\mathcal{N}^B$ as follows. $\lambda^B(t^v) = \lambda(t)$, $\lambda^B(t^{v,w}) = \lambda(t)$, $\lambda^B(\tau^v_{\tau,t}) = \lambda(\tau)$, and $\lambda^B(\tau^{r^B}_\tau) = \lambda(\tau)$.
For any marking $\mathbf{m}$ of $\mathcal{N}^B$, we define the projection of $\mathbf{m}$ to a specific node $u \in V_B$, denoted by $[\mathbf{m}]_u$, as follows. Let $\mathbf{m}$ be a marking of $\mathcal{N}^B$ defined by:

$$\mathbf{m} = \sum_{u \in V_B} \left(a^u_{run} \cdot p_{run} + a^u_{sleep} \cdot p^u_{sleep} + \sum_{p \in P} a^u_p \cdot p^u + \sum_{t \in T_{ab}} a^u_t \cdot p^u_t \right) + a_\emptyset \cdot p_\emptyset + a_{ab} \cdot p_{ab}$$

then:

$$[\mathbf{m}]_u = a^u_{sleep} \cdot p^u_{sleep} + \sum_{p \in P} a^u_p \cdot p^u + \sum_{t \in T_{ab}} a^u_t \cdot p^u_t$$

We are now in position to show the inclusion of the family of B-bounded RPN cut languages in the family of PN coverability languages. The inverse inclusion is immediate: the family of PN coverability languages is included in the family of B-bounded RPN cut languages.

Proposition 9.3.2. *The family of B-bounded RPN cut languages is included to the family of PN coverability languages.*

Proof. We establish that for any $B \in \mathbb{N}$ we have $\mathcal{L}_B(\mathcal{N}, [r, \mathbf{m}], \lambda, \emptyset) = \mathcal{L}_C(\mathcal{N}^B, \mathbf{m}_B, \lambda^B, \{p_\emptyset\})$. We do it by showing the double inclusion.

Showing $\mathcal{L}_B(\mathcal{N}, [r, \mathbf{m}], \lambda, \emptyset) \subseteq \mathcal{L}_C(\mathcal{N}^B, \mathbf{m}_B, \lambda^B, \{p_\emptyset\})$

Given $\omega \in \mathcal{L}_B(\mathcal{N}, [r, \mathbf{m}], \lambda, \emptyset)$ there exists a cut sequence $[r, \mathbf{m}] \xrightarrow{\sigma} \emptyset$ where:

$$\sigma = s_0 \xrightarrow{(v_1, t_1)} s_1 \xrightarrow{(v_2, t_2)} \cdots \xrightarrow{(v_\ell, t_\ell)} s_\ell = \emptyset,$$

$\lambda(\sigma) = \omega$, and $|\sigma|_{ab} \leq B$.

If cut steps would instead of removing vertices would only forbid firing in them and their descendants, then after firing the sequence σ we would get a state with an underlining rooted tree T. T is isomorphic to a subtree of T_B rooted in r_B, since each inner node can not have more than B children and its depth is smaller then B. Denote this isomorphism by ϕ.
We now show by induction that there exists a firing sequence

$$\mathbf{m}_B = \mathbf{m}_0 \xrightarrow{t'_1} \mathbf{m}_1 \xrightarrow{t'_2} \cdots \xrightarrow{t'_\ell} \mathbf{m}_\ell$$

in $\mathcal{N}^B$ such that $\mathbf{m}_i \geq M(\phi, s_i)$ and $\lambda_B(t'_i) = \lambda(t_i)$.

For $i = 0$, $\mathbf{m}_0 = \mathbf{m}_B$ and the induction holds from the definition of $\mathbf{m}_B$. Assume we have shown the induction hypothesis for $i < n$, we turn to show it for $i = n$. We split the proof into four cases according to t_i (where the last two are for cut transitions). For each type we show: a) the transition simulating t_i in $\mathcal{N}^B$ with the same label, b) this transition is fireable from $\mathbf{m}_{i-1}$, and c) the new reached marking $\mathbf{m}_i \geq M(\phi, s_i)$:

1. $t_i \in T_{el}$.
   (a) **Replacement** - $t'_i = t_i^{\phi(v_i)}$ and by definition their label is the same.
   (b) **Fireable from $\mathbf{m}_{i-1}$** - From the induction hypothesis we know that:
   $\mathbf{m}_{i-1} \geq M(\phi, s_{i-1}) = \mathbf{m}_{\mathsf{Anc}}(\phi(v_i)) + [M(v_i)]^{\phi(v_i)} \geq \mathbf{m}_{\mathsf{Anc}}(\phi(v_i)) + [W^-(t_i)]^{\phi(v_i)} = \mathbf{Pre}^B(t^{\phi(v_i)})$
   hence $t^{\phi(v_i)}$ is fireable from $\mathbf{m}_{i-1}$.
   (c) **Marking reached** - Since $\mathbf{C}^B(t_i^{\phi(v_i)})$ only changes the marking of places of the type $p^{\phi(v_i)}$ for $p \in P$ and t_i only changes the marking of the vertex, v_i we get that:
   $M(\phi, s_i) - M(\phi, s_{i-1}) = [W^+(t_i) - W^-(t_i)]^{\phi(v_i)} = \mathbf{C}^B(t_i^{\phi(v_i)})$
   hence $\mathbf{m}_i = \mathbf{m}_{i-1} + \mathbf{C}^B(t^{\phi(v_i)}) \geq M(\phi, s_i)$
2. $t_i \in T_{ab}$, which creates a new vertex w in s_i.
   (a) **Replacement** - $t'_i = t_i^{\phi(v_i),\phi(w)}$ and by definition their label is the same.
   (b) **Fireable from $\mathbf{m}_{i-1}$** - By the same reasoning as for the elementary transitions, plus the fact that the vertex w is fresh (i.e. $p_{sleep}^{\phi(w)} = 1$) and we fired less than B abstract transitions (i.e. $p_{ab} > 0$) we have that $\mathbf{m}_{i-1} \geq \mathbf{Pre}^B(t_i^{\phi(v_i),\phi(w)})$.
   (c) **Marking reached** - Note that the changes in s_i compared to s_{i-1} are a new edge to a new vertex w marked by $\Omega(t_i)$, and that the marking of v_i changes according to $W^-(t_i)$. We get that:

$$\begin{aligned} M(\phi, s_i) - M(\phi, s_{i-1}) &= p_{run}^{\phi(w)} + p_{t_i}^{\phi(w)} - p_{sleep}^{\phi(w)} + [\Omega(t_i)]^{\phi(w)} - [W^-(t_i)]^{\phi(v_i)} \\ &= \mathbf{C}^B(t_i^{\phi(v_i),\phi(w)}) \end{aligned}$$

hence $\mathbf{m}_i = \mathbf{m}_{i-1} + \mathbf{C}^B(t_i^{\phi(v_i),\phi(w)}) \geq M(\phi, s_i)$

3. $t_i \in T_\tau$ and $v_i \neq r$. Denote by $\hat{t}$ the transition that created v_i.
   (a) **Replacement** - $t'_i = \tau_{t_i,\hat{t}}^{\phi(v_i)}$ and by definition their label is the same.
   (b) **Fireable from $\mathbf{m}_{i-1}$** - By the same reasoning as for the elementary transitions, plus the fact that *hatt* created it (i.e. $p_h att\{phi(v_i)\} = 1$) we get $\mathbf{m}_{i-1} \geq \mathbf{Pre}^B(\tau_{t_i,\hat{t}}^{\phi(v_i)})$.
   (c) **Marking reached** - The changes in s_i compared to s_{i-1} are that the vertex v_i is removed and the marking of $prd(v_i)$ changes according to $W^+(\hat{t})$. We get that:

$$M(\phi, s_i) - M(\phi, s_{i-1}) \leq [W^+(\hat{t})]^{\phi(prd(v_i))} - p_{run}^{\phi(v_i)} = \mathbf{C}^B(t_{t_i,\hat{t}}^{\phi(v_i)})$$

   Hence, $\mathbf{m}_i = \mathbf{m}_{i-1} + \mathbf{C}^B(t_{t_i,\hat{t}}^{\phi(v_i)}) \geq M(\phi, s_i)$

4. $t_i \in T_\tau$ and $v_i = r$.
   (a) **Replacement** - $t'_i = \tau_{t_i}^{\phi(v_i)}$, and by definition their label is the same.
   (b) **Fireable from $\mathbf{m}_{i-1}$** - By the same reasoning as for the elementary transitions $\mathbf{m}_{i-1} \geq \mathbf{Pre}^B(\tau_{t_i}^{\phi(v_i)})$.
   (c) **Marking reached** - We have that $s_i = \emptyset$ and firing $\tau_{t_i}^{\phi(v_i)}$

$$M(\phi, s_i) - M(\phi, s_{i-1}) \leq p_\emptyset - p_{run}^{r_B} = \mathbf{C}^B(t_{t_i,\hat{t}}^{r_B})$$

   Hence, $\mathbf{m}_i = \mathbf{m}_{i-1} + \mathbf{C}^B(t_{t_i,\hat{t}}^{r_B}) \geq M(\phi, s_i)$

Combining all the above we get the covering sequence $\sigma' = t'_1 t'_2 \dots t'_\ell$ such that:

$$\mathbf{m}_B \xrightarrow{\sigma'} \mathbf{m}_\ell \geq p_\emptyset$$

Therefore $\omega = \lambda(\sigma') \in \mathcal{L}_C(\mathcal{N}^B, \mathbf{m}_B, \lambda^B, \{p_\emptyset\})$.

Showing $\mathcal{L}_C(\mathcal{N}^B, \mathbf{m}_B, \lambda^B, \{p_\emptyset\}) \subseteq \mathcal{L}_B(\mathcal{N}, [r, \mathbf{m}], \lambda, \emptyset)$

Given $\omega \in \mathcal{L}_C(\mathcal{N}^B, \mathbf{m}_B, \lambda^B, \{p_\emptyset\})$, there exists a covering sequence σ:

$$\mathbf{m}_B = \mathbf{m}_0 \xrightarrow{t_1} \mathbf{m}_1 \xrightarrow{t_2} \cdots \xrightarrow{t_\ell} \mathbf{m}_\ell \geq p_\emptyset$$

in $\mathcal{N}^B$ such that $\lambda_B(\sigma) = \omega$. We will show that there exists a cut sequence σ':

$$[r, \mathbf{m}] = s_0 \xrightarrow{(v_1,t'_1)} s_1 \xrightarrow{(v_2,t'_2)} \cdots \xrightarrow{(v_\ell,t'_\ell)} s_\ell = \emptyset$$

in $\mathcal{N}$ such that for any $0 \leq i \leq \ell$ we have:

1. There is an isomorphism ϕ_i from a subtree of T^B to the underling tree of the state s_i
2. For $U_i = \{v \in V_B \mid \mathbf{m}_i \geq \mathbf{m}_{\mathsf{Anc}}(v)\}$, we have $\phi_i(U_i) = V_{s_i}$
3. For all vertices $v \in V_{s_i} \setminus r$, and $u = \phi_i^{-1}(v)$:
$$0 \leq [\mathbf{m}_i]_u - p^u_{\Upsilon_{s_i}(v)} \leq [M_{s_i}(v)]^u$$
where for $v = r$ then $[\mathbf{m}_{r_B}]_u \leq [M_{s_i}(r)]^u$
4. $\lambda_B(t_i) = \lambda(t'_i)$.

We do this by induction on $0 \leq i \leq \ell$. For $i = 0$ since $[r, \mathbf{m}]$ has only one vertex we let $\phi_0(r_B) = r$ satisfying assertion 1. From the definitions we have $\mathbf{m}_0 = \mathbf{m}_B$ which satisfies the assertions 2-3. Finally, there is no t_0 so assertion 4. does not apply to this base case. Assume we have shown this for every $n < i$ and show it for $n = i$. We split the proof into three cases according to t_i:

- $t_i = x^u$ for $u \in V_B$ and $x \in T_{el}$. First, note that since t^u was fired from $\mathbf{m}_{i-1}$ we have $\mathbf{m}_{i-1} > p^u_{run}$. Let $v_i = \phi_{i-1}(u) \in V_{s_{i-1}}$, which exists by assertion 2. We show that that letting $t'_i = x$ we would get that (v_i, t'_i) is fireable from s_{i-1} to an appropriate s_i. First notice that by induction:
$$[M_{s_{i-1}}(v_i)]^u \geq [\mathbf{m}_{i-1}]_u - p^u_{\Upsilon_{s_{i-1}}(v)} \geq \mathbf{Pre}^B(x^u) - \mathbf{m}_{\mathsf{Anc}}(u) = [W^-(x)]^u$$
hence $M_{s_{i-1}}(v_i) \geq W^-(t)$ and (v_i, t'_i) is fireable from s_{i-1}. Since firing x^u does not change the tokens in any p^w_{run} for $w \in V_B$ and firing (v_i, t'_i) does not change the structure of the underlining tree of s_i, letting $\phi_i = \phi_{i-1}$ satisfies assertions 1 and 2.
Since (v_i, t'_i) only change the marking of v_i, we get that for any vertex $w \neq u$ assertion 3 holds. For u we get that :
$$[m_i]_u = [m_{i-1}]_u + \mathbf{C}^B(x^u) \leq [M_{s_{i-1}}(v_i)]^u + [W^+(t'_i) - W^-(t'_i)]^u$$
and $M_{s_i}(v_i) = M_{s_{i-1}}(v_i) + W^+(t'_i) - W^-(t'_i)$ which satisfies assertion 3 for the node u. Finally, by definition $\lambda(t'_i) = \lambda(x) = \lambda_B(x^u) = \lambda_B(t_i)$.
- $t_i = x^{u,u'}$ for $u, u' \in V_B$, $u = prd(u')$ in T_B, and $x \in T_{ab}$. By the same reasoning as in the previous case $v_i = \phi_{i-1}(u) \in V_{s_{i-1}}$ and letting $t'_i = x$ we get that (v_i, t'_i) is fireable from s_{i-1}.
Since $t'_i \in T_{ab}$, firing (v_i, t'_i) creates for v_i a new child v'_i. Denote by:
$$\phi_i(w) = \begin{cases} \phi_{i-1}(w) & w \neq u' \\ v'_i & w = u' \end{cases}$$
ϕ_i is an isomorphism from a subtree of T_B to the underlining tree of s_i, hence assertion 1 holds. Moreover, since the transition $t_i = x^{u,u'}$ adds a token to

$p^{u'}_{run}$ and for any node $w \neq u'$ does not decrease the number of tokens in p^w_{run}, we get that $\phi_i(U_i) = \phi_i(U_{i-1} \cup u') = V_{s_{i-1}} \cup \{v'\} = V_{s_i}$ satisfying assertion 2. For any $w \in V_{s_i}$ such that $w \neq v_i, v'_i$ we have $M_{s_i}(w) = M_{s_{i-1}}(w)$ since firing (v_i, t'_i) does not change their marking. For any $w \in V_B$ such that $w \neq u, u'$ we have $[\mathbf{m}_i]_w = [\mathbf{m}_{i-1}]_w$ since firing $x^{u,u'}$ does not change their marking. Therefore, to show assertion 3 holds we only need to show it for the nodes u, u', and we get:

$$\begin{aligned}[\mathbf{m}_i]_u &= [\mathbf{m}_{i-1}]_u + \left[\mathbf{C}^B\left(x^{u,u'}\right)\right]_u \leq \\ &\leq [M_{s_{i-1}}(v_i) - W^-(t'_i)]^u + p^u_{\Upsilon_{s_{i-1}}(v)}\end{aligned}$$

$$\begin{aligned}[\mathbf{m}_i]_{u'} =& [\mathbf{m}_{i-1}]_{u'} + [\Omega(x)]^{u'} + p^{u'}_{run} + p^u_x - p^{u'}_{sleep} \\ =& [\Omega(x)]^{u'} + p^u_x\end{aligned}$$

While in $\mathcal{N}$ we have $M_{s_i}(v) = M_{s_{i-1}}(v_i) - W^-(t'_i)$, $M_{s_i}(v'_i) = \Omega(t'_i)$ and $\Upsilon_{s_i}(v') = t$ which satisfies assertion 3 for the nodes u and u'. Finally, by definition $\lambda(t'_i) = \lambda(x) = \lambda_B(x^{u,u'}) = \lambda_B(t_i)$.

- $t_i = \tau^u_{x,y}$ or $\tau^{r_B}_x$ for $y \in T_{ab}$ and $x \in T_\tau$. Denote by u' the parent of u in T_B. By the same reasoning as in the previous case $v_i = \phi_{i-1}(u) \in V_{s_{i-1}}$ and letting $t'_i = y$ we get that (v_i, t'_i) is fireable from s_{i-1}.
  If $u = r_B$ then $t_i = \tau^{r_B}_x$ and $U_i = V_{s_i} = \emptyset$ and assertion 1-3 are fulfilled.
  Assume that $u \neq r_b$, then $t_i = \tau^u_{x,y}$. Firing (v_i, x) removes v_i and its descendants, hence $V_{s_i} = V_{s_{i-1}} \setminus (Des_{s_{i-1}}(v_i))$. Denote by $\phi_i = \phi_{i-1}|_{\phi^{-1}(V_{s_i})}$ this is an isomorphism from a subtree of T_B to the underlining tree of s_i, hence assertion 1 holds. Moreover, since the transition $t_i = \tau^u_{x,y}$ does not change the number of tokens in p^w_{run} for any $w \neq u$ and decrees the tokens in p^u_{run} to 0, we have $U_i = U_{i-1} \setminus (Des_{T_B}(u))$ and assertion 2 holds.
  Let v'_i be the parent of v_i in s_{i-1}. For any $w \in V_{s_i}$ such that $w \neq v'_i$, $M_{s_i}(w) = M_{s_{i-1}}(w)$ since firing (v_i, x) does not change their marking. For any $w \in U_i$ such that $w \neq u'$ we have $[\mathbf{m}_i]_w = [\mathbf{m}_{i-1}]_w$ since firing $\tau^u_{x,y}$ does not change their marking. Therefore, to show assertion 3 we only need to show it for the node u'. From assertion 3 (of the previous step), we know that $\Upsilon_{s_{i-1}}(v_i) = y$. We get that:

$$\begin{aligned}[\mathbf{m}_i]_u &= [\mathbf{m}_{i-1}]_u + \left[\mathbf{C}^B\left(\tau^u_{x,y}\right)\right]_u = \\ &= [M_{s_{i-1}}(v'_i) + W^+(y)]^u + p^u_{\Upsilon_{s_{i-1}}(v'_i)}\end{aligned}$$

and $M_{s_i}(v'_i) = M_{s_{i-1}}(v'_i) + W^+(y)$ which satisfies assertion 3 for the node u'. Finally, by definition $\lambda(t'_i) = \lambda(x) = \lambda_B(\tau^u_{x,y}) = \lambda_B(t_i)$.

Recall that $\mathbf{m}_B = \mathbf{m}_0 \xrightarrow{\sigma} \mathbf{m}_\ell \geq p_\emptyset$ hence $t_\ell = \tau_\tau^{r_B}$. This transition has to appear since it is the only transition adding a token in $p_\emptyset$. It is the final one since, after firing it there are zero tokens in $p_{run}^{r_B}$ and $\mathbf{Pre}^B(t) \geq p_{run}^{r_B}$ for any $t \in T^B$. Therefore, $s_0 \xrightarrow{\sigma'} s_\ell = \emptyset$, and from assertion 4. we get that $\omega = \lambda_B(\sigma) = \lambda(\sigma')$. Moreover, $|\sigma'|_{ab} = |\{t^{u,u'} \in \sigma \mid u, u' \in V^B, t \in T_{ab}\}| \leq B$ since $\mathbf{C}(t^{u,u'})(p_{ab}) = -1$ for $t^{u,u'} \in T^B$, $\mathbf{C}^B(t)(p_{ab}) = 0$ for $t \in T^B$, and $\mathbf{m}_0(p_{ab}) = B$. So we conclude that $\omega \in \mathcal{L}_B(\mathcal{N}, [r, \mathbf{m}], \lambda, \emptyset)$. □

9.3.2 Language Hierarchy

In this subsection we show that the family of coverability languages for DRPN is strictly larger than the family of coverability languages for RPN. We will do it in two ways, using each time a different feature of DRPN not present in RPN. The first feature we are going to use is the ability of the DRPN to create threads whose initial marking depends on the marking of the thread which created them. Recall that in Proposition 8.4.7 we have shown that the language $\mathcal{L}_3 = \{a^n b^m c^m \mid n \geq m\}$ is not an RPN coverability language.

Proposition 9.3.3. *$\mathcal{L}_3$ is a DRPN cut language.*

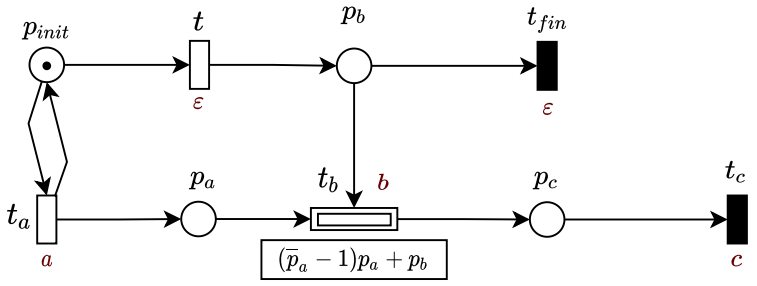


Figure 9.7: A DRPN with a coverability language $\mathcal{L}_3$.

Proof. Let $\mathcal{N}$ be the DRPN in Figure 9.7, we show that $\mathcal{L}_R(\mathcal{N}, [r, p_{init}], \lambda, \emptyset) = \mathcal{L}_3$.
On one hand, given $w = a^n b^m c^m \in \mathcal{L}_3$ i.e. $n \geq m$. The sequence $\sigma = t_a^n\, t\, t_b^m\, t_{fin}\, t_c^m$ is a cut sequence and $\lambda(\sigma) = a^n b^m c^m$.
On the other hand, let σ be a cutting sequence such that $\lambda(\sigma) \in \mathcal{L}_R(\mathcal{N}, [r, p_{init}], \lambda, \emptyset)$. We note that any cut sequence has to be of type $\sigma_{x,y,z} = t_a^x\, t\, t_b^y\, t_{fin}\, t_c^z$ for $x, y, z \geq 0$. We get that $x \geq y$ since t_b consumes a token from t_b. Furthermore, $y = z$ since a thread firing t_b consumes the token from p_b and the only way to cut that thread would be to fire t_c. Therefore, $\lambda(\sigma) = a^x b^y c^z$ with $x \geq y = z$ and $\lambda(\sigma) \in \mathcal{L}_3$. □

It is important to note that, the DRPN in figure 9.7 is almost a regular RPN, where the only difference is that Bg_{t_b} is not a constant marking but an additive function depending on the marking of the thread firing it.

The second feature we are going to use is the ability of transitions in DRPN to perform "arbitrary" functions. We achieve this by extending the result in [102] about *weakly computable* function by Petri nets to RPNs.

Definition 9.3.4. A function $f : \mathbb{N} \to \mathbb{N}$ is called *weakly computable by a Petri net* if there exists a Petri net $\mathcal{N}$ which has four designated places p_{beg}, p_{in}, p_{out} and p_{fin} and fulfills the following properties:

1. For any $n \in \mathbb{N}$ there exists a sequence σ_n such that $p_{beg} + n \cdot p_{in} \xrightarrow{\sigma_n} \mathbf{m} \geq p_{end}$ and the number of tokens in p_{out} is exactly $f(n)$.
2. For any $n \in \mathbb{N}$ and any sequence σ such that $p_{beg} + n \cdot p_{in} \xrightarrow{\sigma} \mathbf{m} \geq p_{end} + x \cdot p_{out}$, we have $x \leq f(n)$.

In [102] the authors show that *slow* functions are not weakly computable by Petri nets.

Definition 9.3.5 (slow/sublinear function)**.** We say that a function $f : \mathbb{N} \to \mathbb{N}$ is *sublinear* if $\lim_{n\to\infty} \frac{f(n)}{n} = 0$. We call the function f *slow* if it is sublinear, non decreasing and $\lim_{n\to\infty} f(n) = \infty$.

For example $\lfloor \log(1+n) \rfloor$ and $\lfloor \sqrt{n} \rfloor$ are slow functions.

Theorem 9.3.6 ([102])**.** *A slow function cannot be weakly computed by a Petri net.*

Given a slow function f and the alphabet $\{a, b\}$ we denote the language $\mathcal{L}_f = \{a^k b^m \mid k \geq 0 \wedge f(k) \geq m\}$. As an immediate consequence of the properties of f, one defines by induction the strictly increasing sequence $(\alpha(n))_{n\in\mathbb{N}}$: $\alpha(0) = 0$ and $\alpha(n+1) = \min(m \mid \alpha(n) < m \wedge f(\alpha(n)) < f(m))$. Note that α depends on f, but for sake of readability, and where it is clear from context, we write α instead of α_f. The next proposition establishes that $\mathcal{L}_f$ is a DRPN coverability language.

Proposition 9.3.7. *For any slow function f, $\mathcal{L}_f$ is a DRPN cover language.*

Proof. Let $\mathcal{N}$ be the DRPN in Figure 9.8, we show that $\mathcal{L}_C(\mathcal{N}, [r, p_{w_a}], \lambda, [r, p_{w_b}]) = \mathcal{L}_f$.
On one hand, given $a^k b^m \in \mathcal{L}_f$, from the definition of the language $m \leq f(k)$. This word is also in
$\mathcal{L}_C(\mathcal{N}, [r, p_1], \lambda, [r, p_2])$, where the appropriate covering sequence is $t_a^k t_b^m$.

On the other hand, given a firing sequence σ it can take the following forms: 1) t_a^k for $k \geq 0$, or 2) $t_a^k t_{w_b} t_b^m$ for $k, m \geq 0$ for which $m \leq f(k) + 1$. The first form does not cover p_2. The second form, if $m = f(k) + 1$ then the reached marking does not cover p_2. Therefore, firing sequences $t_a^k t_{w_b} t_b^m$ with $m \leq f(k)$ are the all covering sequences, and $\lambda(t_a^k t_{w_b} t_b^m) = a^k b^m \in \mathcal{L}_f$. □

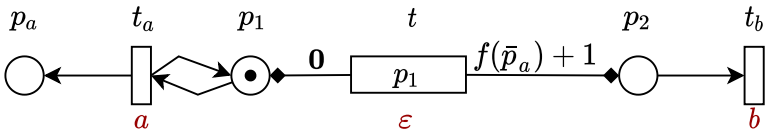


Figure 9.8: A DRPN with a coverability language $\mathcal{L}_f$

Let us pick an arbitrary slow function f. The remainder of this section consists in showing that the language $\mathcal{L}_f$ is not an RPN cut language. We will establish that every RPN cut language which includes $\mathcal{L}_f$ strictly includes it. Let us pick an arbitrary labeled RPN $\mathcal{N}$ with an initial state $[r, \mathbf{m}_0]$, such that $\mathcal{L}_f \subseteq \mathcal{L}_R(\mathcal{N}, [r, \mathbf{m}_0], \lambda, \emptyset)$. Let $\{\sigma_n\}_{n \in \mathbb{N}}$ be a family of cut sequences such that $\lambda(\sigma_n) = a^{\alpha(n)} b^{f(\alpha(n))}$ where among the possible σ_n's, we pick one with the minimal depth and among those one with the minimal length (i.e. $\min |\sigma_n|$). The skeleton of the proof is organized in three propositions:

- Proposition 9.3.8 establishes that $\mathcal{L}_f$ is a not a PN coverability language. Using Proposition 9.3.2 we conclude that there does not exist an RPN $\mathcal{N}$ and $B \in \mathbb{N}$ such that $\mathcal{L}_f$ is its B-bounded language.
- Proposition 9.3.9 establishes that if the depth of $\{\sigma_n\}_{n \in \mathbb{N}}$ is bounded then $\mathcal{L}_R(\mathcal{N}, [r, \mathbf{m}_0], \lambda, \emptyset)$ is a B-bounded language. This shows that if the depth of $\{\sigma_n\}_{n \in \mathbb{N}}$ is bounded then $\mathcal{L}_f \neq \mathcal{L}_R(\mathcal{N}, [r, \mathbf{m}_0], \lambda, \emptyset)$.
- Proposition 9.3.10 concludes the proof by showing that if the depth of $\{\sigma_n\}_{n \in \mathbb{N}}$ is unbounded, $\mathcal{L}_R(\mathcal{N}, [r, \mathbf{m}_0], \lambda, \emptyset)$ contains words that do not belong to $\mathcal{L}_f$. This shows that $\mathcal{L}_f$ is strictly included in $\mathcal{L}_R(\mathcal{N}, [r, \mathbf{m}_0], \lambda, \emptyset)$, i.e. $\mathcal{L}_f \subsetneq \mathcal{L}_R(\mathcal{N}, [r, \mathbf{m}_0], \lambda, \emptyset)$.

Note that, the family of cut sequences $\{\sigma_n\}_{n \in \mathbb{N}}$ is used in the rest of this subsection.

We first show that $\mathcal{L}_f$ is not a Petri net coverability language, using Theorem 9.3.6:

Proposition 9.3.8. *$\mathcal{L}_f$ is not a PN coverability language.*

Proof. Assume for contradiction that there is a Petri net $\tilde{\mathcal{N}} = \langle P, T, \mathbf{Pre}, \mathbf{C} \rangle$ with an initial marking $\mathbf{m}_0$, a set of final markings M_t, and a labeling function

$\lambda : T \to \{a, b, \varepsilon\}$ for which $\mathcal{L}(\tilde{\mathcal{N}}, \mathbf{m}_0, \lambda, M_t) = \mathcal{L}_f$. Let us define a Petri net $\mathcal{N}' = \langle P', T', \mathbf{Pre}', \mathbf{C}' \rangle$ that weakly computes f. Let $P' = P \cup \{p_{beg}, p_{run}, p_{fin}, p_{in}, p_{out}\}$ and $T' = T \cup \{t_{\mathbf{m}} \mid \mathbf{m} \in M_t\} \cup \{t_{run}\}$. For any $t \in T$ we set:

$$\mathbf{Pre}'(t) = \begin{cases} \mathbf{Pre}(t) + p_{in} + p_{run} & \lambda(t) = a \\ \mathbf{Pre}(t) + p_{run} & else \end{cases} ; \mathbf{C}'(t) = \begin{cases} \mathbf{C}(t) - p_{in} & \lambda(t) = a \\ \mathbf{C}(t) + p_{out} & \lambda(t) = b \\ \mathbf{C}(t) & else \end{cases}$$

For every $\mathbf{m} \in M_t$, we set $\mathbf{Pre}(t_{\mathbf{m}}) = \mathbf{m} + p_{run}$ and $\mathbf{C}(t_{\mathbf{m}}) = p_{fin} - p_{run}$. Last we set $\mathbf{Pre}(t_{run}) = p_{beg}$ and $\mathbf{C}(t_{run}) = p_{run} + \mathbf{m}_0$.
For all $n \in \mathbb{N}$, there is a sequence $\tilde{\sigma}_n$ in the original net $\tilde{\mathcal{N}}$, such that $\lambda(\tilde{\sigma}_n) = a^n b^{f(n)}$ and for some $\mathbf{m} \in M_t$ we have that $\mathbf{m}_{ini} \xrightarrow{\tilde{\sigma}_n} \mathbf{m}' \geq \mathbf{m}$. Therefore, the sequence $t_{run}\tilde{\sigma}_n t_{\mathbf{m}}$ is fireable in the Petri net $\mathcal{N}'$, from the marking $p_{beg} + np_{in}$ reaching a marking greater or equal than p_{end} with exactly $f(n)$ tokens in p_{out}.
Let $\mathbf{m}_n = p_{beg} + np_{in}$ be an initial marking and σ be a sequence such that $\mathbf{m}_n \xrightarrow{\sigma} \mathbf{m} > xp_{out} + p_{end}$ for some x. The first transition in this sequence has to be t_{run}, since it is the only one fireable from $\mathbf{m}_n$. The final marking is greater or equal than p_{end}, hence there has to be a firing of $t_{\mathbf{m}}$, for some $\mathbf{m} \in M_t$, in the sequence but after firing $t_{\mathbf{m}}$ no other transition can be fired. Combining these two facts, we get that $\sigma = t_{run}\sigma' t_{\mathbf{m}}$ where $\sigma' \in T^*$ is fireable in the original net $\tilde{\mathcal{N}}$, where $\mathbf{m}_0 \xrightarrow{\sigma'} \mathbf{m}' \geq \mathbf{m} \in M_t$. Therefore $\lambda(\sigma') = a^m b^\ell \in \mathcal{L}_f$, where $m \leq n$ since the transitions labeled with a can be fired at most n times (the initial marking of $\mathcal{N}'$ had n tokens in p_{in}). Since $m \leq n$, we get that $\ell \leq f(m) \leq f(n)$. Therefore, looking back to $\mathcal{N}'$, we have that after firing σ we have $x \leq f(n)$. □

Given a labeled RPN $(\mathcal{N}, \lambda)$ and an abstract transition t (i.e. $t \in T_{ab}$), we introduce the following languages:

$$\mathcal{L}_{\mathcal{N},\lambda}(t) = \{\lambda(\sigma) \mid s[r, Beg_t] \xrightarrow{\sigma}\} \; ; \; \mathcal{L}^{\perp}_{\mathcal{N},\lambda}(t) = \{\lambda(\sigma) \mid s[r, Beg_t] \xrightarrow{\sigma} \emptyset\}$$

These are the languages can appear in a subtree created by t (closed or not). Recall that $\{\sigma_n\}_{n\in\mathbb{N}}$ is the family of cut sequences such that $\lambda(\sigma_n) = a^{\alpha(n)} b^{f(\alpha(n))}$ defined in the skeleton of the proof.

Proposition 9.3.9. *Let $(\mathcal{N}, \lambda)$ be a labeled RPN, s_0 be its initial state such that $\mathcal{L}_f \subseteq \mathcal{L}_R(\mathcal{N}, s_0, \lambda, \{\emptyset\})$. Assume that the depths of $\{\sigma_n\}_{n\in\mathbb{N}}$ are bounded. Then $\mathcal{L}_f \subsetneq \mathcal{L}_R(\mathcal{N}, s_0, \lambda, \{\emptyset\})$.*

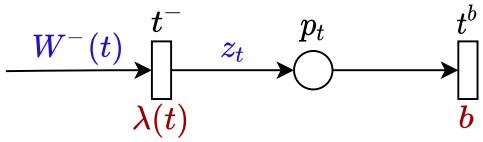
Proof. Let $D \geq 0$ be the maximum of the depths of the family of $\{\sigma_n\}_{n=1}^{\infty}$. For clarity, we let $\mathcal{L}_{\mathcal{N}}$ denote $\mathcal{L}_R(\mathcal{N}, s_0, \lambda, \{\emptyset\})$ more concisely.

We are going to build an RPN $\widehat{\mathcal{N}}$ such that $\mathcal{L}_f \subseteq \mathcal{L}_B(\widehat{\mathcal{N}}, p_{ini}) \subseteq \mathcal{L}_{\mathcal{N}}$. This would finish the proof since by Proposition 9.3.2 and 9.3.8, $\mathcal{L}_f \subsetneq \widehat{\mathcal{L}}$.

We first build the labeled RPN $\bar{\mathcal{N}} = \langle \bar{P}, \bar{T}, \bar{W}^+, \bar{W}^-, \bar{\Omega} \rangle$ with label $\bar{\lambda}$ by adding places and transitions to $(\mathcal{N}, \lambda)$ as follows. For all $t \in T_{ab}$, one adds new places and transitions according to $\mathcal{L}_{\mathcal{N},\lambda}(t)$ and $\mathcal{L}^{\perp}_{\mathcal{N},\lambda}(t)$:

- If $a^+b^+ \cap \mathcal{L}_{\mathcal{N},\lambda}(t) = \emptyset$ and $z_t = \max\{m \mid b^m \in \mathcal{L}_{\mathcal{N},\lambda}(t)\} < \infty$ then one adds two elementary transitions t^-, t^b and a place p_t (see figure below), where:

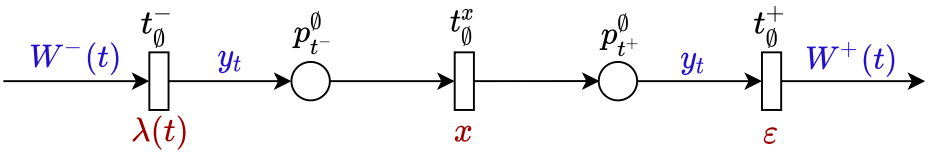
$$\begin{array}{lll} \bar{W}^-(t^-) = W^-(t), & \bar{W}^+(t^-) = z_t \cdot p_t, & \bar{\lambda}(t^-) = \lambda(t); \\ \bar{W}^-(t^b) = p_t, & \bar{W}^+(t^b) = 0, & \bar{\lambda}(t^b) = b; \end{array}$$



- Otherwise, if $a^+b^+ \cap \mathcal{L}^{\perp}_{\mathcal{N},\lambda}(t) = \emptyset$ and $\max\{m \mid b^m \in \mathcal{L}^{\perp}_{\mathcal{N},\lambda}(t)\} < \infty$ then one adds, three elementary transitions $t^-_{\perp}$,$t^x_{\perp}$, $t^+_{\perp}$ and two places $p^{\perp}_{t^-}, p^{\perp}_{t^+}$ with y_t and x are defined below:

$$\begin{array}{lll} \bar{W}^-(t^-_{\perp}) = W^-(t), & \bar{W}^+(t^-_{\perp}) = y_t \cdot p^{\perp}_{t^-}, & \bar{\lambda}(t^-_{\perp}) = \lambda(t); \\ \bar{W}^-(t^x_{\perp}) = p^{\perp}_{t^-}, & \bar{W}^+(t^x_{\perp}) = p^{\perp}_{t^+} - p^{\perp}_{t^-}, & \bar{\lambda}(t^x_{\perp}) = x; \\ \bar{W}^-(t^+_{\perp}) = y_t \cdot p^{\perp}_{t^+}, & \bar{W}^+(t^+_{\perp}) = W^-(t) - y_t \cdot p^{\perp}_{t^+}, & \bar{\lambda}(t^+_{\perp}) = \varepsilon \end{array}$$

  - If $b^m \in \mathcal{L}^{\perp}_t$ for $m > 0$ then $y_t = \max\{m \mid b^m \in \mathcal{L}^{\perp}_t\}$ and $x = b$;
  - Else if $a^\ell \in \mathcal{L}^{\perp}_t$ for $\ell > 0$ then $y_t = \min\{\ell \mid a^\ell \in \mathcal{L}^{\perp}_t\}$ and $x = a$;
  - Otherwise, $y_t = 1$ and $x = \varepsilon$.



- Otherwise, nothing is changed.

On the one hand $\mathcal{L}_{\mathcal{N}} \subseteq \mathcal{L}_R(\bar{\mathcal{N}}, s_0, \bar{\lambda}, \emptyset)$ since any firing sequence in $\mathcal{N}$ can be performed in $\bar{\mathcal{N}}$ and for all transitions $t \in T$ the labeling functions agree, i.e $\lambda(t) = \bar{\lambda}(t)$. On the other hand, the new transitions are built according to $\mathcal{L}_{\mathcal{N}}(t)$ and $\mathcal{L}^{\perp}_{\mathcal{N}}(t)$ in such a way that every firing of a new transition can be replaced by a firing of a sequence of transitions with the same produced label. Hence $\mathcal{L}_R(\bar{\mathcal{N}}, s_0, \bar{\lambda}, \emptyset) = \mathcal{L}_{\mathcal{N}}$.
We now show that there exists some B, such for all $n \in \mathbb{N}$ there is a firing sequence σ'_n in $\bar{\mathcal{N}}$ with $|\sigma'_n|_{ab} \leq B$ and $\bar{\lambda}(\sigma'_n) = \lambda(\sigma_n)$. Pick an arbitrary $n \in \mathbb{N}$ and denote

more explicitly the cut sequence $s_0 \xrightarrow{\sigma_n} \emptyset$. Assume there is an occurrence $t \in T_{ab}$ by the vertex u in σ_n creating a vertex v. Recall that, D is the maximal depth of the family $\{\sigma_n\}_{n\in\mathbb{N}}$. We transform σ_n depending on whether the firing (u,t) has a matching cut transition (v,τ) in σ_n:

The firing (u,t) has a matching cut.

1. Assume that $a^+b^+ \cap \mathcal{L}^{\perp}_{\mathcal{N}}(t) \neq \emptyset$. If there are more than D occurrences of t with a matching cut in σ_n then there are two occurrences (w_1,t) and (w_2,t) where w_2 is neither a descendant nor an ancestor of w_1. So one could build a cut sequence σ with $\ell_i, m_i > 0$ such that $\lambda(\sigma) = \ldots a^{\ell_1}b^{m_1}a^{\ell_2}b^{m_2}\ldots \in \mathcal{L}_{\mathcal{N}}$. So, $\mathcal{L}_f \subsetneq \mathcal{L}_{\mathcal{N}}$ and we are done.
2. If $\max\{m \mid b^m \in \mathcal{L}^{\perp}_{\mathcal{N}}(t)\} = \infty$. Let m be the number of occurrences of b in σ_n produced at the subtree rooted in v then there exists $m' > m$ such that one can build a cut sequence σ with $m > f(\alpha(n))+1$ such that $\lambda(\sigma) = a^{\alpha(n)}b^{f(\alpha(n))+m'-m} \in \mathcal{L}_{\mathcal{N}}$ So $\mathcal{L}_f \subsetneq \mathcal{L}_{\mathcal{N}}$ and we are done.
3. Otherwise (i.e. $a^+b^+ \cap \mathcal{L}^{\perp}_{\mathcal{N}}(t) = \emptyset$ and $\max\{m \mid b^m \in \mathcal{L}^{\perp}_{\mathcal{N}}(t)\} < \infty$), we replace the firing of (u,t) by the sequence below and remove all firings from $Des(v)$,

$$(u,t^-_{\perp})\overbrace{(u,t^x_{\perp})\ldots(u,t^x_{\perp})}^{y_t \text{ times}}(u,t^+_{\perp})$$

and obtain a cut sequence σ'_n such that $\bar{\lambda}(\sigma'_n) = a^{\ell}b^m$ with $\ell \leq \alpha(n)$ and $m \geq f(\alpha(n))$. If $\ell < \alpha(n)$, and $m > f(\alpha(n))$ then $\mathcal{L}_f \subsetneq \mathcal{L}_{\mathcal{N}}$ and we are done. Otherwise $\bar{\lambda}(\sigma'_n) = \lambda(\sigma_n)$ and $|\sigma'_n|_{ab} < |\sigma_n|_{ab}$. We can repeat this process until either one concludes that $\mathcal{L}_f \subsetneq \mathcal{L}_{\mathcal{N}}$ or there are at most D firings of t with a matching cut transition in σ'_n.

The firing (u,t) does not have a matching cut.

We again have the same three cases only with $\mathcal{L}_{\mathcal{N},\lambda}(t)$ instead of $\mathcal{L}^{\perp}_{\mathcal{N},\lambda}(t)$. Case 3 is slightly different. Here we replace the firing of (u,t) by:

$$(u,t^-)\overbrace{(u,t^b)\ldots(u,t^b)}^{z_t \text{ times}}$$

We have built a sequence σ'_n with $\bar{\lambda}(\sigma'_n) = \lambda(\sigma_n)$ and $|\sigma'_n|_{ab} \leq 2D|T_{ab}|$. So we choose $B = 2D|T_{ab}|$.

We now enlarge the labeled RPN $(\bar{\mathcal{N}},\bar{\lambda})$ to $(\widehat{\mathcal{N}},\widehat{\lambda})$ in such a way that for all $a^{\ell}b^m \in \mathcal{L}_f$ there is a cut sequence σ with $\widehat{\lambda}(\sigma) = a^{\ell}b^m$ and $|\sigma|_{ab} \leq B$. We observe that the properties of α imply that: $\mathcal{L}_f = \{a^{\ell}b^m \mid \exists n, \delta^-, \delta^+ \ \ell = \alpha(n) + \delta^+ \wedge m = f(\alpha(n)) - \delta^-\}$. Due to the observation about RPN languages, the initial state of $\bar{\mathcal{N}}$ only consists of one vertex, whose initial marking is denoted $\mathbf{m}_{ini}$. So one builds

the RPN $\widehat{\mathcal{N}}$ (for an example see Figure 9.9) where its initial state only consists of one vertex with initial marking p_{ini} as follows.

• Add elementary transitions t_a, t_{run}, t_{ab} and places p_{ini}, p_{run} such that:

$$\begin{array}{lll} \widehat{W}^-(t_a) = p_{ini}, & \widehat{W}^+(t_a) = p_{ini}, & \widehat{\lambda}(t_a) = a; \\ \widehat{W}^-(t_{run}) = p_{ini}, & \widehat{W}^+(t_{run}) = \mathbf{m}_{ini} + p_{run}, & \widehat{\lambda}(t_{run}) = \varepsilon; \\ \widehat{W}^-(t_{ab}) = p_{run}, & \widehat{W}^+(t_{ab}) = 2p_{run}, & \widehat{\lambda}(t_{ab}) = \varepsilon. \end{array}$$

• For all $t \in T$, we set:

$$\widehat{W}^-(t) = \bar{W}^-(t) + p_{run}, \quad \widehat{W}^+(t) = \bar{W}^+(t), \quad \widehat{Bg}(t) = \bar{B}g(t) + p_{run} \text{ when } t \in T_{ab}$$

• For all $t \in T$ with $\bar{\lambda}(t) = b$, we add transition t_ε which is a copy of t with $\widehat{\lambda}(t_\varepsilon) = \varepsilon$.

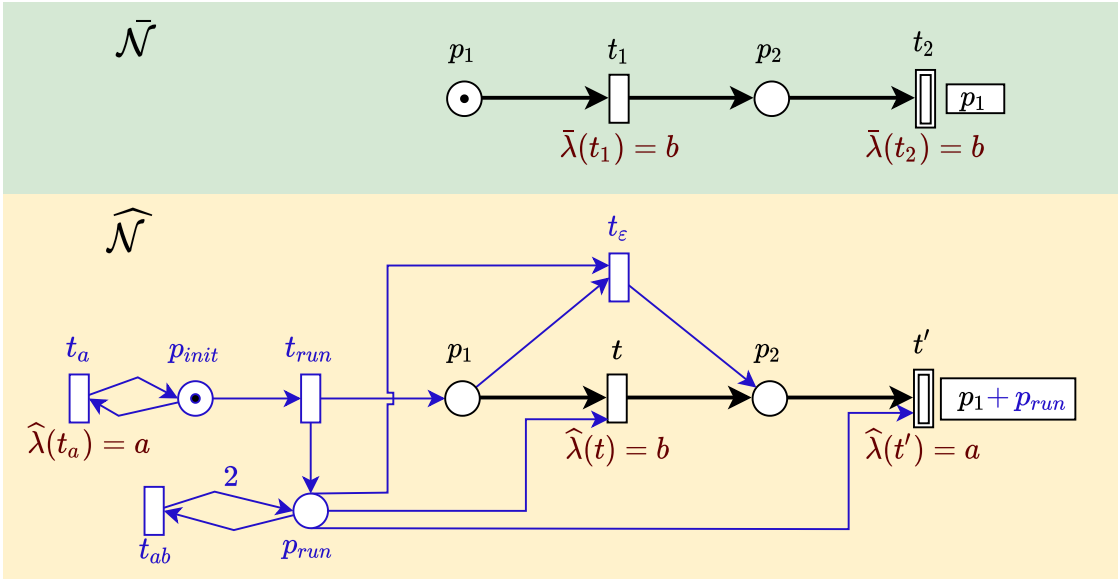


Figure 9.9: From $\bar{\mathcal{N}}$ to $\widehat{\mathcal{N}}$

We now prove that $\mathcal{L}_f$ is included in the bounded language of $(\widehat{\mathcal{N}}, \widehat{\lambda})$. Let $a^\ell b^m \in \mathcal{L}_f$. Then there exist n, δ^-, δ^+ such that $\ell = \alpha(n) + \delta^+$ and $m = f(\alpha(n)) - \delta^-$. Let σ_n be a cut sequence in $\bar{\mathcal{N}}$ such that $\lambda'(\sigma_n) = a^{\alpha(n)} b^{f(\alpha(n))}$. We define σ to be the cut sequence of $\widehat{\mathcal{N}}$ as follows.

σ starts by $(r, t_a)^{\delta^+}(r, t_{run})(r, t_{ab})^{|\sigma_n|}$. Then σ is completed by $\widehat{\sigma}_n$ where $\widehat{\sigma}_n$ is obtained from σ_n by:

- changing δ^- occurrences of transitions with label b by their copy;

- whenever σ_n creates a new vertex v, one inserts $(v, t_{ab})^{|\sigma_n|}$ firings.

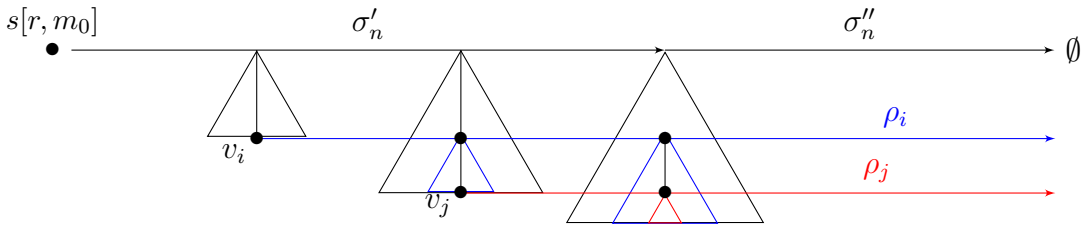
Observe that $\widehat{\lambda}(\sigma) = a^\ell b^m$. Let w be a word. Define $w_{\downarrow b}$ as the set of words obtained from w by omitting some occurrences of b. Define $\mathcal{L}'' = \{w \mid \exists w' \in \mathcal{L}_\mathcal{N}\ w \in a^* w'_{\downarrow b}\}$. Therefore $\mathcal{L}_f \subseteq \mathcal{L}_B(\widehat{\mathcal{N}}, p_{ini}, \widehat{\lambda}, \emptyset) \subseteq \mathcal{L}''$. Since $\mathcal{L}_f$ is not a B-bounded language, by propositions 9.3.2 and 9.3.8, $\mathcal{L}_f \subsetneq \mathcal{L}''$. Finally, we get that if $\mathcal{L}_\mathcal{N} = \mathcal{L}_f$ then $\mathcal{L}'' = \mathcal{L}_f$ which concludes the proof. □

We are now in position to conclude that $\mathcal{L}_f$ is not an RPN cut language. Note that the following proof is very similar to the second part of the proof of Proposition 8.4.7.

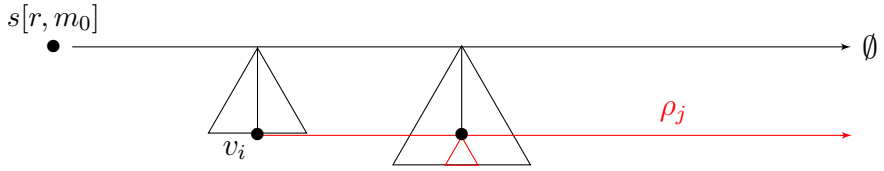
Proposition 9.3.10. *$\mathcal{L}_f$ is not an RPN cut language.*

Proof. Let $(\mathcal{N}, s_0, \lambda)$ be a marked RPN such that $\mathcal{L}_f \subseteq \mathcal{L}_R(\mathcal{N}, s_0, \lambda, \emptyset)$. Let us denote more concisely $\mathcal{L}_R(\mathcal{N}, s_0, \lambda, \emptyset)$ by $\mathcal{L}_\mathcal{N}$. By Proposition 9.3.9 if the depths of $\{\sigma_n\}_{n\in\mathbb{N}}$ are bounded then $\mathcal{L}_f \subsetneq \mathcal{L}_\mathcal{N}$ and we are done.
So assume that the depths of $\{\sigma_n\}_{n\in\mathbb{N}}$ are unbounded. There is some n with $\sigma_n = \sigma'_n\sigma''_n$ such that $s_0 \xrightarrow{\sigma'_n} s'$ where the depth of s' is greater than $(2|T| + 1)$. Thus, in s_n, for all j such that $1 \leq j \leq 3$, there are edges $u_j \xrightarrow{m}_{s_n} v_j$ and denoting i_j the depth of v_j, one has $0 < i_1 < i_2 < i_3$.

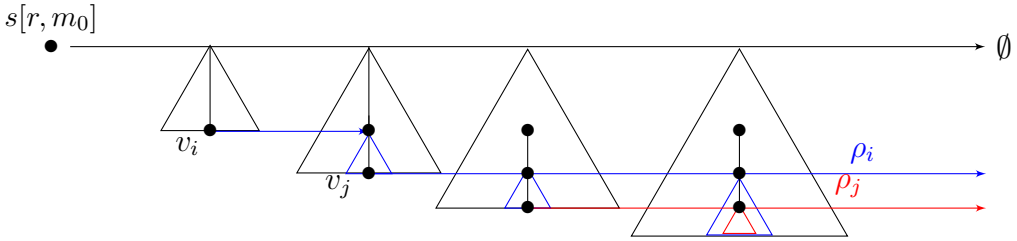
For $k \in \{1, 2, 3\}$, consider of the sequence ρ_k performed in the subtree rooted in v_k by the firings of σ_n. Among these three firing sequences, two of them either (1) both finish by a cut transition in v_k or (2) both do not finish by a cut transition in v_k. Let us call i, j with $i < j$ the indices of these sequences and w_i and w_j their traces. We have illustrated the situation below.



One can build two firing sequences that still reach $\emptyset$ and thus whose labels belong to the language. The first one consists of mimicking the "behavior" of the subtree rooted in v_j starting from v_i, which is possible due to the choice of i and j, as illustrated below.



The second one consists of mimicking the "behavior" of the subtree rooted in v_i starting from v_j as illustrated below.



Denote by w the trace of the sequence performed in the subtree rooted in v_i without the trace of the sequence performed in the subtree rooted in v_j. Let us examine the possible case for word w:

- $w = \varepsilon$. Then one can build another covering sequence with the trace $a^{\alpha(n)}b^{f(\alpha(n))}$ where we mimic the behavior of v_j in v_i. By doing so we get a covering sequence σ'_n not deeper than σ_n but which is shorter then σ_n, i.e. $|\sigma_n| > |\sigma'_n|$ which is a contradicts to our assumption about σ_n being the minimal one.
- $w = a^\ell$, for $\ell > 0$. Then one can build another covering sequence by mimicking the behavior of v_j in v_i. But then the trace of the new covering sequence will be $a^{\alpha(n)-\ell}b^{f(\alpha(n))}$ and since $\ell > 0$ we get that $a^{\alpha(n)k-\ell}b^{f(\alpha(n))} \notin \mathcal{L}_f$, from which we conclude that $\mathcal{L}_f \subsetneq \mathcal{L}_\mathcal{N}$.
- $w = b^m$, for $m > 0$. Then one can build another covering sequence by mimicking the behavior of v_i in v_j. But then the trace of the new covering sequence will be $a^{\alpha(n)}b^{f(\alpha(n))+m} \notin \mathcal{L}_f$ from which we conclude that $\mathcal{L}_f \subsetneq \mathcal{L}_\mathcal{N}$.
- $w = a^\ell b^m$, for $\ell, m > 0$. Then one can build a family of covering sequences $\{\widehat{\sigma}_x\}_{x\in\mathbb{N}}$ by mimicking the behavior of v_i from v_j recursively x times. We would get that $\lambda(\widehat{\sigma}_x) = a^{\alpha(n)+x\ell}b^{f(\alpha(n))+xm}$ for any $x \in \mathbb{N}$. But that would give us that:

  $$\frac{f(\alpha(n)) + xm}{\alpha(n) + x\ell} \xrightarrow{x\to\infty} \frac{m}{\ell} > 0$$

  Since f is sublinear there exists $x \in \mathbb{N}$ such that $\lambda(\widehat{\sigma}_x) \notin \mathcal{L}_f$, from which we conclude that $\mathcal{L}_f \subsetneq \mathcal{L}_\mathcal{N}$.

□

9.3.3 Closure Properties

We now study the closure properties of the family of DRPN coverability languages under several forms of operations. The proof of this subsection are omitted since the proofs from section 8.4 and 8.3 are also valid here. The proof of the next proposition is based on the proof of Proposition 8.4.10, and Lemmas 8.3.2 and 8.3.3.

Proposition 9.3.11. *Given a labelled marked DRPN $(\mathcal{N}, s, \lambda)$ there exists a marked DRPN $(\mathring{\mathcal{N}}, s[r, \mathbf{m}], \mathring{\lambda})$ with the same cut language.*

Using this equivalence and similar proofs as those for Propositions 8.4.11, 8.4.12 and 8.4.13, where the same control parts described in Figures 8.8, 8.9, and 8.10 can be used for DRPN, it is straightforward to show the following closure properties:

Proposition 9.3.12. *The family of DRPN coverability languages is closed under:*

1. *Union*
2. *Concatenation*
3. *Kleene star*

Using the same proof as for the one for Proposition 8.4.14, since any RPN language is also a DRPN language, we get:

Proposition 9.3.13. *The family of coverability languages of DRPNs is not closed under intersection with a regular language and under complementation.*

9.4 Decidability of the coverability problem

In order to prove that the coverability problem for DRPN is decidable, we are missing one important ingredient. Recall that in Section 8.3 we developed some reductions and techniques to handle covering sequences in RPN. We now develop similar techniques for DRPN. Specifically, given a DRPN $\mathcal{N}$ we build a kind of $\widehat{\mathcal{N}}_{el}$ (Definition 8.3.13) for DRPN, i.e. a DRPN which is unable to fire abstract transitions but which is still able to simulate the markings of a single thread of the DRPN $\mathcal{N}$. In order to achieve this for RPN, we had to generate the set $T_{ret} \subseteq T_{ab}$, the set of abstract transitions whose newly created thread can be cut. In DRPN we encounter a difficulty: the initial marking of a created thread is not constant but depends on the marking of the thread creating it. Therefore, computing this set is a bit more involved and we need the following definitions:

Definition 9.4.1. Let $\mathcal{N}$ be a DRPN. Then $\mathit{Cuttable}(\mathcal{N}) \subseteq \mathbb{N}^P$ is defined by:

$$\mathit{Cuttable}(\mathcal{N}) = \min\{\mathbf{m} \mid \exists \sigma \; s[r, \mathbf{m}] \xrightarrow{\sigma} \emptyset\}$$

and for all $t \in T_{ab}$, $Closed_t$ is defined by:

$$Closed_t = \min(Bg_t^{-1}(Cuttable(\mathcal{N}))$$

Due to the properties of DRPN transition firing rules, these sets are upward closed.

Example 9.4.2. Consider the DRPN of Figure 9.4. With one token in both p_{time} and in p_{adv} one fires t_{found} producing a token in p_{dead} which allows firing a cut transition. Furthermore, firing the abstract transition t_{hire} does not help to reach $\emptyset$ since in the new vertex, the marking of p_{time} is equal to the marking of p_{time} in its parent vertex and the marking p_{adv} is smaller than the marking of p_{adv} in in its parent vertex. Thus $Cuttable(\mathcal{N}) = \{p_{time} + p_{adv}\}$. The marking of a vertex created by t_{hire} is greater or equal than $p_{time}+p_{adv}$ if in its parent vertex there is at least one token in p_{time} and three tokens in p_{adv}. Thus $Closed_{t_{hire}} = \{p_{time}+3p_{adv}\}$.

If we are able to compute $Cuttable$ and thus $\{Closed_t\}_{t\in T_{ab}}$ we still encounter an issue building the DRPN $\widehat{\mathcal{N}}_{el}$. Recall that for RPN in order to get $\widehat{\mathcal{N}}_{el}$ we took the transition in $t \in T_{ret}$ and replaced them by elementary transition t_r. Firing these transitions had the same effect on the state as of a sequence starting by creating a new thread by t and then cutting it. In DRPN it is impossible to replace such a sequence by a single transition since the effect of the cut depends on the marking of the thread, hence the timing of the cut transition is crucial. Therefore, we replace the single t_r transitions by $T_r = \{t^-, t^+ \mid t \in T_{ab}\}$. To ensure the sequentiality between these firings, the set of places is extended with $P_r = \{p_t \mid t \in T_{ab}\}$ with one token produced (resp. consumed) in p_t by t^- (resp. t^+). To ensure that the firing of t^- is performed when the corresponding firing of t can be matched later by the firing of the cut transition, the guard of t^- is the guard of t intersected with $Closed_t$.

In order to formally define $\widehat{\mathcal{N}}$ and to exhibit relations between states and thus markings of $\mathcal{N}$ and $\widehat{\mathcal{N}}$, we introduce the projection Proj from $\mathbb{N}^{\widehat{P}}$ to $\mathbb{N}^P$ where $\widehat{P} = P \cup P_r$.

Definition 9.4.3. Let $\mathcal{N}$ be a DRPN.
Then $\widehat{\mathcal{N}}_{el} = \langle \widehat{P}, \widehat{T}, \widehat{Gd}, \widehat{Up}, \widehat{Up^-}, \widehat{Up^+}, \widehat{Bg}\rangle$ is a DPRN defined by: $\widehat{Up^+} = \widehat{Bg} = \emptyset$

- $\widehat{P} = P \cup P_r$ and $\widehat{T} = T_{el} \cup T_\tau \cup T_r$;
- for all $t \in T_{el} \cup T_\tau$, $\widehat{Gd}_t = Gd_t$
  for all $t^- \in T_r$, $\widehat{Gd}_{t^-} = \min(\uparrow Gd_t \cap \uparrow Closed_t)$
  for all $t^+ \in T_r$, $\widehat{Gd}_{t^+} = \{p_t\}$;
- for all $t \in T_{el}$, all $p \in P$ and all $p_{t'} \in P_r$,
  $\widehat{Up}_t(p) = Up_t(p) \circ \mathsf{Proj}$ and $\widehat{Up}_t(p_{t'}) = p_{t'}$;
- for all $t^- \in T_r$, all $p \in P$ and all $p_{t'} \in P_r$,
  $\widehat{Up}_{t^-}(p) = Up_t^-(p) \circ \mathsf{Proj}$ and $\widehat{Up}_t(p_{t'}) = p_{t'} + 1_{t=t'}$;

- for all $t^+ \in T_r$, all $p \in P$ and all $p_{t'} \in P_r$,
  $\widehat{Up}_{t^+}(p) = Up_t^+(p) \circ \mathsf{Proj}$ and $\widehat{Up}_t(p_{t'}) = p_{t'} - \mathbf{1}_{t=t'}$.

Let us denote by $M_{=0} = \{\mathbf{m} \in \mathbb{N}^{\widehat{P}} \mid \forall t \in T_{ab}\ \mathbf{m}(p_t) = 0\}$ the set of markings with no token in every p_t.

Before showing that given $\mathcal{N}$ we can compute $\widehat{\mathcal{N}}_{el}$, we show that there exists a cut sequence from an initial state with one thread in $\mathcal{N}$ if and only if there exists one in $\widehat{\mathcal{N}}_{el}$. Otherwise stated:

Lemma 9.4.4. *Let $\mathcal{N}$ be a DRPN, then $Cuttable(\mathcal{N}) = \mathsf{Proj}(Cuttable(\widehat{\mathcal{N}}_{el}))$*

Proof. We show this lemma by double inclusion.
On one hand, let $\mathbf{m} \in Cuttable(\mathcal{N})$, i.e. there exist a cutting sequence $s[r, m] \xrightarrow{\sigma}_{\mathcal{N}} \emptyset$. Observe that if in σ, there exists a firing of an abstract transition creating some vertex v not followed later by a cut transition in v, then one can omit this firing and all firings in the subtree rooted at v and still reaches $\emptyset$. Thus, we assume that every vertex v created by the firing of an abstract transition is later deleted by a matching cut transition in v. For any abstract transition t fired from the root r in σ creating the thread v_t. Denote by $\sigma_t(\tau, v_t)$ the firing sub-sequence fired in the subtree rooted in v_t. We get the firing sequence $\widehat{\sigma}$ from σ as follows. For every firing of an abstract transition t from the root, modify σ as follows: (1) replace (t, r) firing by (t^-, r), (2) remove the sub-sequence σ_t from σ, and (3) replace (τ, v_t) by (t^+, r). The firing sequence $\widehat{\sigma}$ is firable in $\widehat{\mathcal{N}}_{el}$ and has that $s[r, m] \xrightarrow{\widehat{\sigma}}_{\widehat{\mathcal{N}}_{el}} \emptyset$.
On the other hand, given $\mathbf{m} \in \mathsf{Proj}(Cuttable(\widehat{\mathcal{N}}_{el}))$, i.e. there exists a cut sequence $s[r, m] \xrightarrow{\widehat{\sigma}}_{\widehat{\mathcal{N}}_{el}} \emptyset$. Observe that like in the previous case if in σ, there exists a firing of some t^- not paired with a later firing of transition in t^+, then one can omit t^- and still reaches $\emptyset$. By the construction of t^- and t^+, if t^- is fireable from $s[r, \mathbf{m}']$ then there exists a sequence σ_t in $\mathcal{N}$ such that $Bg_t(\mathbf{m}') \xrightarrow{\sigma_t}_{\mathcal{N}} \emptyset$. We get the firing sequence σ from $\widehat{\sigma}$ as follows: Given a firing of t^- in $\widehat{\sigma}$ and its matching t^+ replace (1) (t^-, r) by the abstract transition (t, r) which creates a new thread v_t, and (2) replace the firing of transition t^+ by the sequence σ_t fired from v_t. The firing sequence σ is firable in $\mathcal{N}$ and has that $s[r, m] \xrightarrow{\sigma}_{\mathcal{N}} \emptyset$. □

We are left with proving that there exists an algorithm which given a DRPN $\mathcal{N}$ builds $\widehat{\mathcal{N}}_{el}$. To show that this algorithm exists it is enough to show that an algorithm computing *Cuttable* and $\{Closed_t\}_{t \in T_{ab}}$ exists. Let us describe how Algorithm 7 does exactly that. In the lines 2-8, it builds a version of $\widehat{\mathcal{N}}_{el}$ where for every t, $Closed_t$ is replaced by $Closed[t]$. Since $Closed[t]$ will be updated during the loop of lines 9-18, Gd_{t^-} is updated several times at line 10. Still in this loop, using a standard backward exploration (see Section 2.1), during lines 11-15, it computes in variable X, *Cuttable* for this version of $\widehat{\mathcal{N}}_{el}$. Afterwards, still in this

loop, it updates Y by restricting X to the markings with no token in P_r and then projecting it to P. Then using Y, it updates for every t, *Closed*[t]. The algorithm terminates when Y is no more enlarged.

We now show that the Algorithm 7 is effective. We begin by showing the following lemma:

Lemma 9.4.5. *Let U, V be finite subsets of $\mathbb{N}^P$. Then there is an algorithm computing* $\min(\uparrow U \cap \uparrow V)$ *and* $\min(\uparrow U \cup \uparrow V)$.

Proof. The lines 2-4 of Algorithm 8 compute a set W such that $\uparrow W = \min(\uparrow U \cap \uparrow V)$. It achieves this by computing $\max(u, v)$ defined by:

$$\forall p \in P, \max(u, v)(p) = \max(u(p), v(p))$$

for any two elements $u \in U$ and $v \in V$, and adding it to W. This set is not necessarily minimal, therefore in lines 5-5 removes any item which is not minimal.

Since $\min(\uparrow U \cup \uparrow V) = \min(U \cup V)$ and U, V are finite we are done. □

Algorithm 8: Computing $\min\{\uparrow U \cap \uparrow V\}$

Input: Finite sets U, V

1 $W \leftarrow \emptyset$;
2 **for** $u, v \in U \times V$ **do**
3 | $W \leftarrow W \cup \{\max(u, v)\}$
4 **end**
5 **for** $w, w' \in W$ **do** **if** $w \leq w'$ **then** $W \leftarrow W \setminus \{w'\}$;
6 **return** W;

Using this algorithm we are ready to show that Algorithm 7 is effective.

Proposition 9.4.6. *Algorithm 7 is effective.*

Proof. First note that lines 1–8 are effective since T_{ab} is finite and we only perform initialization of variables.

The sets $X, oldX, Y, oldY$, and Gd_t for all $t \in T$ are finite set at the end of Line 8, and we will show that at any point of the algorithm between 10 - 17 they stay finite. For any $t \in T$ by the definition of DRPN the functions Up_t, Bg_t are effective. `Proj` is also effective (we omit the trivial proof). Therefore, the Line 17 is effective. First, the Line 10, if effective from Lemma 9.4.5 and the previous remark on `Proj`. The sets Gd_t are kept finite since the function min produces only finite sets. Similarly, the lines 11 and 14 are effective and keep all the sets finite. □

Proposition 9.4.7. *Algorithm 7 terminates and it returns $Y = \uparrow Cuttable$ and for all $t \in T_{ab}$, $Closed[t] = Closed_t$.*

Proof. In the sequel *Closed*[t] denotes the value of this variable at some execution point. Let us denote $\mathcal{N}'$ the version of $\widehat{\mathcal{N}}_{el}$ built by the algorithm and updated at every iteration of loop of lines 9-18.

• **Termination.** Denote by Y_n , $Closed_n[t]$ for all t, the sets Y and *Closed*[t] at the beginning of iteration n of the **repeat** loop(lines 9-18). We prove by induction on n that the sequence of sets $\uparrow Y_n$ and for all t, $\uparrow Closed_n[t]$ is an increasing sequence of upward closed sets of $\mathbb{N}^P$ and $\mathbb{N}^{\widehat{P}}$, respectively. Since $\mathbb{N}^P$ and $\mathbb{N}^{\widehat{P}}$ are well-ordered, these sets must stabilize after a finite number of iterations. Since T is finite, this will establish termination of the algorithm.

The basis of the induction is correct since $Y_0, = Closed_0[t] = \emptyset$ for all $t \in T$. It remains to prove that for $n > 1$, $\uparrow Y_n \subseteq \uparrow Y_{n+1}$, $\uparrow Closed_n[t] \subseteq \uparrow Closed_{n+1}[t]$. We start by showing that the **while** loop terminates(lines 12-15). We also prove it by induction on n. Consider the n^{th} iteration of the **repeat** loop, and let us prove the sequence of sets $\uparrow X_n^k$ at the beginning of the iteration k of the **while** loop is an increasing sequence of upward closed sets. This will establish the termination of this loop. Observe that $X_n^1 = \min\{\mathtt{Proj}^{-1}(Gd_t) \mid t \in T_\tau\}$, and that at every iteration

$$\uparrow X_n^{k-1} \subseteq \uparrow X_n^k = (Up_t^{-1}(\uparrow X) \cap \uparrow Gd_t) \cup \uparrow X_n^{k-1}.$$

Denote by $X_n = X_n^k$, where k is the minima value for which $\uparrow X_n^k = \uparrow X_n^{k+1}$. Note that, $X_{n-1} \subseteq X_n$, since $\uparrow Closed_{n-1}[t] \subseteq \uparrow Closed_n[t]$. Finally, $Y_n = \mathtt{Proj}(X_n \cap M_{=0})$ for which we know that:

$$\uparrow Y_{n-1} = \uparrow \mathtt{Proj}(X_{n-1} \cap M_{=0}) \subseteq \uparrow \mathtt{Proj}(X_n \cap M_{=0}) = \uparrow Y_n,$$

and for all $t \in T$ we have:

$$\uparrow Closed_{n-1}[t] = Bg_t^{-1}(\uparrow Y_{n-1}) \subseteq Bg_t^{-1}(\uparrow Y_n) = \uparrow Closed_n[t].$$

• **Consistency.** We establish by induction on the iterations of the **repeat** loop that $\uparrow Y \subseteq \uparrow Cuttable(\mathcal{N})$ and for all $\mathbf{m} \in \uparrow Closed[t]$, there is a sequence $s_{Bg(t)(\mathbf{m})} \xrightarrow{\sigma_{\mathbf{m}}} \emptyset$, implying $\uparrow Closed[t] \subseteq \uparrow Closed_t$. Consider an arbitrary iteration of the **repeat** loop. Thus, the **while** loop computes X which is the set $Cuttable(\mathcal{N}')$. Since by induction, $\uparrow Closed[t] \subseteq \uparrow Closed_t$ one deduces that $\uparrow X \subseteq \uparrow Cuttable(\widehat{\mathcal{N}}_{el})$. Applying Lemma 9.4.4, one deduces that
$Y \subseteq \uparrow Cuttable(\mathcal{N})$ and so that at the end of the iteration $\uparrow Closed[t] \subseteq \uparrow Closed_t$.

• **Completeness.** Let $\mathbf{m} \in \uparrow Closed_t$. Consider a sequence $s_{Bg(t)(\mathbf{m})} \xrightarrow{\sigma}_{\mathcal{N}} \emptyset$. Observe that if in σ, there exists a firing of and abstract transition creating some vertex v not later followed by a cut transition in v, then one can omit this firing and all firings in the subtree rooted at v and still reaches $\emptyset$. Thus, we assume that every vertex v created by the firing of abstract transition is later deleted by a matching cut transition in v.

We establish the completeness of the algorithm by recurrence on the depth of σ. If the depth is null, it means that σ only includes firing of elementary transitions in r ended by the cut transition. So $\widehat{s}_{Bg(t)(\mathbf{m})} \xrightarrow{\sigma}_{\mathcal{N}_{el}} \emptyset$. Furthermore, since $\sigma \in (\{r\} \times T_{el})^*(r, \tau)$, $\widehat{s}_{Bg(t)(\mathbf{m})} \xrightarrow{\sigma}_{\mathcal{N}'} \emptyset$ for $\mathcal{N}'$ built at the beginning of the first iteration of the **repeat** loop. During every iteration of the **repeat** loop, the **while** loop computes in X the set of markings from which a sequence of transitions of $\mathcal{N}'$ leads to some marking greater than a marking in $\{\mathtt{Proj}^{-1}(Gd_t) \mid t \in T_\tau\}$. So $\widehat{Bg_t(\mathbf{m})} \in \uparrow X$ at the end of the iteration and after the **for** loop at line 17, $\mathbf{m} \in \uparrow Closed[t]$.
Assume that σ has depth $h > 0$. So every $\mathbf{m}'$ in the root from which there is a firing of an abstract transition t' belongs to $\uparrow Closed[t']$ since the subsequence in the created vertex up to the cut transition has depth strictly less than h. Consider the last iteration of the **repeat** loop for which such t' is added to $\uparrow Closed[t']$. Then either at this iteration $\mathbf{m}$ already belongs to $\uparrow Closed[t]$ or it will be added at the next iteration (which exists since $\uparrow Y$ is enlarged) due to execution of the **while** loop. Indeed, consider a closing subsequence of σ for a child of the root created by some transition t' and substitute the firing of t' by t'^{-}, delete the closing subsequence and substitute the cut step by the firing of t^+ in r. Doing this transformation (and omitting the cut step in r) one obtains a closing firing sequence in $\mathcal{N}'$ as described above from $\widehat{s}_{Bg(t)(\mathbf{m})}$. Thus, at the beginning of the last iteration of the **while** loop, $\mathcal{N}' = \widehat{\mathcal{N}}_{el}$. Using Lemma 9.4.4, one gets that at this end of this iteration, $Y = Cuttable(\mathcal{N})$. □

Finally, we are ready to prove that coverability in DRPN is decidable. This goal is achieved with a set of reductions. We show that (see Figure 9.10): (1) the DRPN coverability problem is equivalent to rooted DRPN cut problem in $\widehat{\mathcal{N}}_{el}$, i.e. a DRPN with no abstract transitions with an initial state consisting of one state and (2) showing that $\widehat{\mathcal{N}}_{el}$ is a WSTS.

Proposition 9.4.8. *The following problems are reducible to each other:*

1. *DRPN coverability problem*
2. *DRPN cut problem*
3. *Rooted DRPN cut problem*
4. *rooted DRPN cut problem in* $\widehat{\mathcal{N}}_{el}$

Proof. First notice, the coverability problem is equivalent to the emptiness problem of the coverability language of an DRPN. The same is true for cut problem. In Proposition 9.3.1 we have shown that these language families are equivalent, and the translation of the problem from one to the other can be done in polynomial space. Therefore, problems (1) and (2) are equivalent. Using Lemma 9.3.11 we get that problems (2) and (3) are equivalent. Finally, from Lemma 9.4.4 we get that (3) and (4) are equivalent, which concludes the proof. □

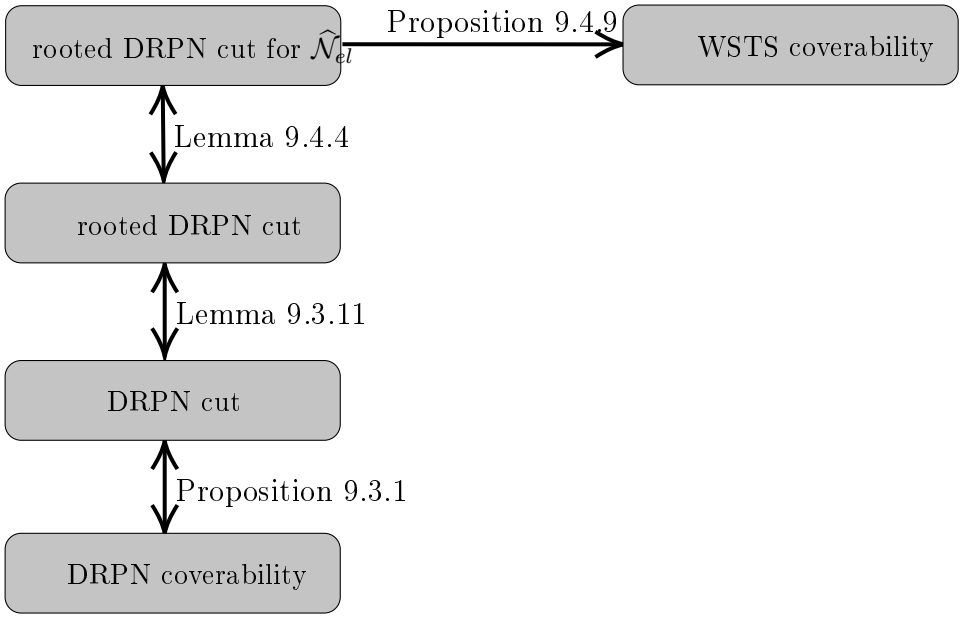


Figure 9.10: The reductions used to prove the decidability of DRPN coverability

Proposition 9.4.9. *For any marked DRPN* $(\mathcal{N}, s[r, \mathbf{m}])$ *the marked DRPN* $(\widehat{\mathcal{N}}_{el}, s[r, \mathbf{m}])$ *is a WSTS.*

Proof. The DRPN $(\widehat{\mathcal{N}}_{el}, s[r, \mathbf{m}])$ is unable to create any new threads, hence the reachable states can only own a single thread. The order on a single state is a wqo since it is equivalent to the usual order on $\mathbb{N}^P$. Since the order is wqo and we already know that it is strongly compatible by Lemma 9.2.8, we get that $(\widehat{\mathcal{N}}_{el}, s[r, \mathbf{m}])$ is a WSTS. □

Theorem 9.4.10. *The DRPN coverability problem is decidable.*

Proof. By Proposition 9.4.8 the cover problem in DRPN is equivalent to the rooted DRPN cut problem in $\widehat{\mathcal{N}}_{el}$. □

Algorithm 7: Computing the closure of abstract transitions

Input: $\mathcal{N}$ a DRPN

// $\widehat{P} = P \cup \{p_t \mid t \in T_{ab}\}$, $\widehat{T}_{el} = T_{el} \cup \{t^-, t^+ \mid t \in T_{ab}\}$

// Proj is the projection from $\mathbb{N}^{\widehat{P}}$ to $\mathbb{N}^P$

Data: $X, oldX$ finite subsets of $\mathbb{N}^{\widehat{P}}$; $oldY, Y$ finite subsets of $\mathbb{N}^P$; t a transition

Output: $Closed$ an array indexed by T_{ab} of finite subsets of $\mathbb{N}^P$

$Y \leftarrow \emptyset$; **for** $t \in T_{ab}$ **do** $Closed[t] \leftarrow \emptyset$

// Initializing $\widehat{N}_{el}$:

for $t \in T_{el}$ **do** $\widehat{Up}_t(\widehat{m}) := \begin{cases} Up_t(\mathtt{Proj}(\widehat{m}(p))) & \text{for all } p \in P \\ \widehat{m}(p) & \text{for all } p \in P_r \end{cases}$

for $t \in T_{el} \cup T_\tau$ **do** $\widehat{Gd}_t := Gd_t$

for $t \in T_{ab}$ **do**

$\widehat{Up}_{t^-}(\widehat{m}) := \begin{cases} Up_{t^-}(\mathtt{Proj}(\widehat{m}(p))) & \text{for all } p \in P \\ \widehat{m}(p) & \text{for all } p \in P_r \setminus \{p_t\} \\ \widehat{m}(p_t) + 1 & \text{when } p = p_t \end{cases}$

$\widehat{Gd}_{t^+}(\widehat{m}) := \{p_t\}$

$\widehat{Up}_{t^+}(\widehat{m}) := \begin{cases} Up_{t^+}(\mathtt{Proj}(\widehat{m}(p))) & \text{for all } p \in P \\ \widehat{m}(p) & \text{for all } p \in P_r \setminus \{p_t\} \\ \widehat{m}(p_t) - 1 & \text{when } p = p_t \end{cases}$

end

// $\widehat{N}_{el}$ is initialized except Gd_{t^-} for all $t \in T_{ab}$.

// Computing $Closed_t$:

repeat

  for $t \in T_{ab}$ **do** $Gd_{t^-} \leftarrow \min(\mathtt{Proj}^{-1}(\uparrow Gd_t \cap \uparrow Closed[t]))$

  $oldY \leftarrow Y$; $X \leftarrow \min(\mathtt{Proj}^{-1}(\bigcup_{t \in T_\tau} \uparrow Gd_t))$; $oldX \leftarrow \emptyset$

  while $X \neq oldX$ **do**

    $oldX \leftarrow X$

    for $t \in \widehat{T}_{el}$ **do** $X \leftarrow \min(\uparrow X \cup \uparrow\min(\uparrow\min(Up_t^{-1}(\uparrow X)) \cap \uparrow Gd_t))$

     // backward exploration

  end

  // Recall that $M_{=0} = \{\mathbf{m} \in \mathbb{N}^{\widehat{P}} \mid \forall t \in T_{ab}\ \mathbf{m}(p_t) = 0\}$.

  $Y \leftarrow \mathtt{Proj}(X \cap M_{=0})$ // new $Cuttable$.

  for $t \in T_{ab}$ **do** $Closed[t] \leftarrow \min(Bg_t^{-1}(\uparrow Y))$ // updating $Closed_t$ acc. to the new $Cuttable$

until $Y \neq oldY$

return $Y, Closed$

Part III

Active Learning and Verification

Chapter 10

Property Directed Verification

10.1 Introduction

Recurrent neural networks (RNNs) are a state-of-the-art tool to represent and learn sequence-based models. They have applications in time-series prediction, sentiment analysis, and many more. In particular, they are increasingly used in safety-critical applications and act, for example, as controllers in cyber-physical systems [4]. Thus, there is a growing need for formal verification. While formal-methods based techniques such as *model checking* [16] have been successfully used in practice and reached a certain level of industrial acceptance, a transfer to machine-learning algorithms has yet to take place. We apply it on machine-learning artifacts rather than on the algorithm.

An emerging research stream aims at extracting, from RNNs, state-based formalism such as finite automata [156, 123, 115, 114, 14, 122]. Finite automata turned out to be useful for understanding and analyzing all kinds of systems using testing or model checking. In the field of formal verification, it has proven to be beneficial to run the extraction and verification process simultaneously [126]. Moreover, the state space of RNNs tends to be prohibitively large, or even infinite, and so do incremental abstractions thereof. Motivated by these facts, we propose an intertwined approach to verifying RNNs, where, in an incremental fashion, grammatical inference and model checking go hand-in-hand. Our approach is inspired by black-box checking [126], which *exploits* the property to be verified *during* the verification process. Our procedure can be used to find misclassified examples or to verify a system that the given RNN controls.

Property-directed verification. Let us give a glimpse of our method. We consider an RNN $\mathcal{R}$ as a binary classifier of finite sequences over a finite alphabet Σ. In other words, $\mathcal{R}$ represents the set of strings that are classified as positive.

We denote this set by $\mathcal{L}(\mathcal{R}) \subseteq \Sigma^*$ and call it, as usual, the *language* of $\mathcal{R}$. We would like to know whether $\mathcal{R}$ is compatible with a given specification $\mathcal{A}$, written $\mathcal{R} \models \mathcal{A}$. Here, we assume that $\mathcal{A}$ is given as a (deterministic) finite automaton. Finite automata are algorithmically feasible, albeit having a reasonable expressive power: many abstract specification languages such as temporal logics or all of the regular expressions can be compiled into finite automata [66].

But what does $\mathcal{R} \models \mathcal{A}$ actually mean? In fact, there are various options. If we know the exact language that the $\mathcal{R}$ should represent, i.e., $\mathcal{A}$ provides a complete characterization of the sequences that are to be classified as positive, then $\models$ refers to language equivalence, i.e., $\mathcal{L}(\mathcal{R}) = \mathcal{L}(\mathcal{A})$. Note that this would imply that $\mathcal{L}(\mathcal{R})$ is supposed to be a regular language, which may rarely be the case in practice. Therefore, we will focus on checking inclusion $\mathcal{L}(\mathcal{R}) \subseteq \mathcal{L}(\mathcal{A})$, which is more versatile as we explain next.

Suppose N is a finite automaton representing a negative specification, i.e., $\mathcal{R}$ must classify words in $\mathcal{L}(N)$ as negative at any cost. In other words, $\mathcal{R}$ does not produce false positives. This amounts to checking that $\mathcal{L}(\mathcal{R}) \subseteq \mathcal{L}(\overline{N})$ where $\overline{N}$ is the "complement automaton" of N. For instance, assume that $\mathcal{R}$ is supposed to recognize valid XML documents over a finite predefined set of tags. Seen as a set of strings, this is not a regular language. However, we can still check whether $\mathcal{L}(\mathcal{R})$ only contains words where every opening tag <*tag-name*> is eventually followed by a closing tag </*tag-name*> (while the number of opening and the number of closing tags may differ). As negative specification, we can then take an automaton N accepting the corresponding *regular* set of strings. For example, `<book><author></author><author></book>` $\in \mathcal{L}(N)$, since the second occurrence of `<author>` is not followed by some `</author>` anymore. On the other hand, we have `<book><author><author></author></book>` $\in \mathcal{L}(\overline{N})$, as `<book>` and `<author>` are always eventually followed by their closing counterpart.

Symmetrically, suppose P is a finite automaton representing a *positive* specification so that we can find false negative classifications: If P represents the words that $\mathcal{R}$ *must classify as positive*, we would like to know whether $\mathcal{L}(P) \subseteq \mathcal{L}(\mathcal{R})$. Our procedure can be run using the complement of P as specification and inverting the outputs of $\mathcal{R}$, i.e., we check, equivalently, $\mathcal{L}(\overline{\mathcal{R}}) \subseteq \mathcal{L}(\overline{P})$.

An important instance of this setting is *adversarial robustness certification*, which measures a neural network's resilience against adversarial examples (specialized inputs created with the purpose of confusing a neural network). Given a (regular) set of words $\mathcal{L}$ classified as positive by the given RNN, the RNN is intuitively *robust* w.r.t. $\mathcal{L}$ if slight modifications in a word from $\mathcal{L}$ do not alter the RNN's judgement. This notion actually relies on a distance function. Then, P is the set of words whose distance to a word in $\mathcal{L}$ is bounded by a predefined threshold, which is regular for several popular distances such as the *Hamming* or

Levenshtein distance. Similarly, we can also check whether the neighborhood of a regular set of words preserves a negative classification. It is important to note that the robustness discussed in Chapter 11 is a different type of robustness.

So, in all these cases, we are faced with the question of whether the language of an RNN $\mathcal{R}$ is contained in the (regular) language of a finite automaton $\mathcal{A}$. Our approach to this problem relies on black-box checking [126], which has been designed as a combination of model checking and testing in order to verify finite-state systems and is based on Angluin's L* learning algorithm [10]. L^* produces a sequel of *hypothesis* automata based on queries to $\mathcal{R}$. Every such hypothesis $\mathcal{H}$ may already share some structural properties with $\mathcal{R}$. So, instead of checking conformance of $\mathcal{H}$ with $\mathcal{R}$, it is worthwhile to first check $\mathcal{L}(\mathcal{H}) \subseteq \mathcal{L}(\mathcal{A})$ using classical model-checking algorithms. If the answer is affirmative, we apply statistical model checking to check $\mathcal{L}(\mathcal{R}) \subseteq \mathcal{L}(\mathcal{H})$ to confirm the result. Otherwise, a counterexample is exploited to refine $\mathcal{H}$, starting a new cycle in L*. Just like in black-box checking, our experimental results (Section 10.6) suggest that the process of interweaving automata learning and model checking is beneficial in the verification of RNNs and offers advantages over more obvious approaches such as (pure) statistical model checking or running automata extraction and model checking in sequence. A further key advantage of our approach is that, unlike in statistical model checking, we often find a *family* of counterexamples, in terms of loops in the hypothesis automaton, which testify conceptual problems of the given RNN.

Note that, though we only cover the case of binary classifiers, our framework is in principle applicable to multiple labels using one-vs-all classification.

Related Work. Mayr and Yovine describe an adaptation of the PAC variant of Angluin's L*(see Section 2.3). algorithm that can be applied to neural networks [115]. As L* is not guaranteed to terminate when facing non-regular languages, the authors impose a bound on the number of states of the hypotheses and on the length of the words for membership queries. In [114, 116], Mayr et al. propose *on-the-fly property checking* where one learns an automaton approximating the intersection of the RNN language and the complement of the property to be verified. Like the RNN, the property is considered as a black box, only decidability of the word problem is required. Therefore, the approach is suitable for non-regular specifications.

Weiss et al. introduce a different technique to extract finite automata from RNNs [156]. It also relies on Angluin's L* but, uses a different type of abstraction of the given RNN to perform equivalence checks between them.

The paper [4] studies formal verification of systems where an RNN-based agent interacts with an environment. The verification procedure proceeds by a reduction

to feed-forward neural networks (FFNNs). It is complete and fully automatic. This is at the expense of the expressive power of the specification language, which is restricted to properties that only depend on bounded prefixes of the system's executions. In our approach, we do not restrict the kind of regular property to verify. The work [82] also reduces the verification of RNNs to FFNN verification. To do so, the authors calculate inductive invariants, thereby avoiding a blowup in the network size. The effectiveness of their approach is demonstrated on audio signal systems. Like in [4], a time interval is imposed in which a given property is verified.

For adversarial robustness certification, Ryou et al. [141] compute a convex relaxation of the non-linear operations found in the recurrent cells for certifying the robustness of RNNs. The authors show the effectiveness of their approach in speech recognition. Besides, MARBLE [50] builds a probabilistic model to quantize the robustness of RNNs. However, these approaches are white-box based and demand the full structure and information of neural networks. Instead, our approach is based on learning with black-box checking.

Elboher et al. present a counter-example guided verification framework whose workflow shares similarities with our property-guided verification [53]. However, their approach addresses FFNNs rather than RNNs. For recent progress in the area of safety and robustness verification of deep neural networks, see [96].

Organization. In Section 10.2, we recall basic notions such as RNNs and finite automata. Section 10.3 describes two basic algorithms for the verification of RNNs, before we present property-directed verification in Section 10.4. How to handle adversarial robustness certification is discussed in Section 10.5. The experimental evaluation and a thorough discussion can be found in Section 10.6.

Based on. This chapter is base on our work in [90].

10.2 Preliminaries

In this section, we provide definitions for the probability and recurrent neural networks. For the definitions of languages, finite automata, and Angluin's L* algorithm one is directed to Section 2.3.

Notation change. Note that, in all Part III we use λ to denote the empty word instead ε.

Probability Distributions. In order to sample words over Σ, we assume a probability distribution $(p_a)_{a\in\Sigma}$ on Σ (by default, we pick the uniform distribution) and a "termination" probability $p \in (0, 1]$. Together, they determine a natural

probability distribution on Σ^* given, for $w = a_1 \dots a_n \in \Sigma^*$, by $\Pr(w) = p_{a_1} \cdot \ldots \cdot p_{a_n} \cdot (1-p)^n \cdot p$. According to the geometric distribution, the expected length of a word is $(1/p) - 1$, with a variance of $(1-p)/p^2$. Let $0 < \varepsilon < 1$ be an error parameter and $\mathcal{L}_1, \mathcal{L}_2 \subseteq \Sigma^*$ be languages. We say that $\mathcal{L}_1$ *ε-approximately correct* w.r.t. $\mathcal{L}_2$ if $\Pr(\mathcal{L}_1 \setminus \mathcal{L}_2) = \sum_{w \in \mathcal{L}_1 \setminus \mathcal{L}_2} \Pr(w) < \varepsilon$.

Recurrent Neural Networks. *Recurrent neural networks (RNNs)* are a generic term for artificial neural networks that process sequential data. They are particularly suitable for classifying sequences of varying length, which is essential in domains such as natural language processing (NLP) or time-series prediction. For the purposes of this thesis, it is sufficient to think of an RNN $\mathcal{R}$ as an effective function $\mathcal{R} : \Sigma^* \to \{0,1\}$, which determines its language as $\mathcal{L}(\mathcal{R}) = \{w \in \Sigma^* \mid \mathcal{R}(w) = 1\}$. Its complement $\overline{R}$ is defined by $\overline{\mathcal{R}}(w) = 1 - \mathcal{R}(w)$ for all $w \in \Sigma^*$. There are several ways to effectively represent $\mathcal{R}$. Among the most popular architectures are long short-term memory (LSTM) [76], and Gated recurrent units (GRUs) [33]. Their expressive power depends on the exact architecture, but generally goes beyond the power of finite automata, i.e., the class of regular languages.

We sometimes use the notations of the RNNs and DFA for their respective languages. For example, we say that $\mathcal{R}$ is ε-approximately correct w.r.t. $\mathcal{A}$ if $\mathcal{L}(\mathcal{R})$ is ε-approximately correct w.r.t. $\mathcal{L}(\mathcal{A})$.

10.3 Verification Approaches

Before we present (in Section 10.4) our method of verifying RNNs, we here describe two simple approaches. The experiments will later compare all three algorithms (SMC, AAMC, and PDV) w.r.t. their performance.

Statistical Model Checking (SMC). One obvious approach for checking whether the RNN under test $\mathcal{R}$ satisfies a given specification $\mathcal{A}$, i.e., to check whether $\mathcal{L}(\mathcal{R}) \subseteq \mathcal{L}(\mathcal{A})$, is by a form of random testing. The idea is to generate a finite test suite $T \subset \Sigma^*$ and to check, for each $w \in T \cap \mathcal{L}(\mathcal{R})$, whether $w \in \mathcal{L}(\mathcal{A})$ holds. If not, each such w is a *counterexample*. On the other hand, if none of the words turns out to be a counterexample, the property holds on $\mathcal{R}$ with a certain error probability. The algorithm is sketched as Algorithm SMC.

Note that the test suite is sampled according to a probability distribution on Σ^*. Recall that our choice depends on two parameters: a probability distribution on Σ and a "termination" probability, both are described in Section 10.2.

Theorem 10.3.1 (Correctness of SMC)**.** *If Algorithm SMC, with $\varepsilon, \gamma \in (0,1)$, terminates with "Counterexample w", then w is mistakenly classified by $\mathcal{R}$ as pos-*

Algorithm 9: SMC

Input: RNN $\mathcal{R}$, DFA $\mathcal{A}$, $\varepsilon, \gamma \in (0,1)$

```
for i = 1, ..., log(2/γ)/(2ε²) do
    w ← sampleWord()
    if w ∈ L(R) \ L(A) then
        return "Counterexample w"
end
return "Property satisfied"
```

Algorithm 10: AAMC

Input: RNN $\mathcal{R}$ and DFA $\mathcal{A}$

```
A_R ← Approximation(R)
if ∃w ∈ L(A_R) \ L(A) then
    return "Counterexample w"
else return "Property satisfied"
```

Algorithm 11: PDV

Input: RNN $\mathcal{R}$, DFA $\mathcal{A}$, $\varepsilon, \gamma \in (0,1)$

```
Initialize L*
while true do
    H ← hypothesis provided by L*
    Check L(H) ⊆ L(A)
    if L(H) ⊆ L(A) then
        Check L(R) ⊆ L(H) using Alg. SMC
        if L(R) ⊆ L(H) then
            return "Property satisfied"
        else Feed counterexample to L*
    else
        Let w ∈ L(H) \ L(A)
        if w ∈ L(R) then
            return "Counterexample w"
        else Feed counterexample w to L*
    end
end
```

itive. If it terminates with "Property satisfied", then $\mathcal{R}$ is ε-approximately correct w.r.t. $\mathcal{A}$ with probability at least $1-\gamma$.

Proof. If the algorithm terminates with "Counterexample w", we have $w \in \mathcal{L}(\mathcal{R}) \setminus \mathcal{L}(\mathcal{A})$. Thus, w is mistakenly classified. Using the sampling described in Section 10.2, denote by $\hat{p}$ the probability to pick $w \in \Sigma^*$ such that $w \in \mathcal{L}(\mathcal{R})$ and $w \notin \mathcal{L}(\mathcal{A})$. Taking $n = \log(2/\gamma)/(2\varepsilon^2)$ random samples where m of them are counter examples, by Hoeffding's inequality bound [77], we get that $\Pr(\hat{p} \notin [\frac{m}{n} - \varepsilon, \frac{m}{n} + \varepsilon]) < \gamma$. Therefore, if Algorithm 9 terminates without finding any counterexamples we get that $\mathcal{R}$ is ε-approximately correct w.r.t. $\mathcal{A}$ with probability at least $1-\gamma$. □

While the approach works in principle, it has several drawbacks for its practical application. The size of the test suite may be quite huge and it may take a while both finding a counterexample or proving correctness.

Moreover, the correctness result and the algorithm assume that the words to be tested are chosen according to a random distribution that somehow also has to take into account the RNN as well as the property automaton.

It has been reported that this method does not work well in practice [156] and our experiments (Section 10.6) support these findings.

Automaton Abstraction and Model Checking (AAMC). As model checking for finite-state systems was extensively researched, a straightforward idea would be to

(a) *approximate* the RNN $\mathcal{R}$ by a finite automaton $\mathcal{A}_\mathcal{R}$, and
(b) to check whether $\mathcal{L}(\mathcal{A}_\mathcal{R}) \subseteq \mathcal{L}(\mathcal{A})$ using model checking.

The algorithmic schema is depicted in Algorithm AAMC.

Here, we can instantiate Approximation() in Algorithm AAMC, by the DFA-extraction algorithms from [115] or [156]. In fact, for approximating an RNN by a finite-state system, several approaches have been studied in the literature, which can be, roughly, divided into two approaches: (a) *abstraction*, and (b) *automata learning*. In the first approach, the state space of the RNN is mapped to equivalence classes according to certain predicates. The second approach uses automata-learning techniques such as Angluin's L*. The approach presented in [156] is an intertwined version combining both ideas.

Therefore, there are different instances of AAMC, varying in the approximation approach. Note that, for verification as language inclusion, as considered here, it actually suffices to learn an over-approximation $\mathcal{A}_\mathcal{R}$ such that $\mathcal{L}(\mathcal{R}) \subsetneq \mathcal{L}(\mathcal{A}_\mathcal{R})$.

While the approach seems promising at first hand, its correctness has two glitches. First, the result "Property satisfied" depends on the quality of the approximation. Second, any returned counterexample w may be *spurious*: w is a counterexample with respect to $\mathcal{A}_\mathcal{R}$ satisfying $\mathcal{A}$ but may not be a counterexample for $\mathcal{R}$ satisfying $\mathcal{A}$. If $w \in \mathcal{L}(\mathcal{R})$, then it is indeed a counterexample, but if not, it is spurious—an indication that the approximation needs to be refined. If the automaton is obtained using abstraction techniques (such as predicate abstraction) that guarantee over-approximations, well-known principles like CEGAR [37] may be used to refine it. In the automata-learning setting, w may be used as a counterexample for the learning algorithm to improve the approximation. Repeating the latter idea suggests an interplay between automata learning and verification—and this is the idea that we follow. However, rather than starting from some approximation with a certain quality that is later refined according to the RNN and the property, we perform a direct, *property-directed* approach.

10.4 Property-Directed Verification of RNNs

We are now ready to present our algorithm for property-directed verification (PDV). The underlying idea is to replace the equivalence queries (EQ) in An-

gluin's L^* algorithm with a combination of classical model checking and statistical model checking, which are used as an alternative to EQs. This approach, which we call *property-directed verification of RNNs*, is outlined as Algorithm PDV and works as follows.

After initialization of L^* and the corresponding data structure, L^* automatically generates and asks membership queries (MQs) to the given RNN $\mathcal{R}$ until it comes up with a first hypothesis DFA $\mathcal{H}$ (Line 3 in Algorithm 11). In particular, the language $\mathcal{L}(\mathcal{H})$ is consistent with the MQs asked so far.

At an early stage of the algorithm, $\mathcal{H}$ is generally small. However, it already shares some characteristics with $\mathcal{R}$. So it is worth checking, using standard automata algorithms, whether there is no mismatch yet between $\mathcal{H}$ and $\mathcal{A}$, i.e., whether $\mathcal{L}(\mathcal{H}) \subseteq \mathcal{L}(\mathcal{A})$ holds (Line 4). Because otherwise (Line 10), a counterexample word $w \in \mathcal{L}(\mathcal{H}) \setminus \mathcal{L}(\mathcal{A})$ is already a candidate for being a misclassified input for $\mathcal{R}$. If indeed $w \in \mathcal{L}(\mathcal{R})$, w is mistakenly considered positive by $\mathcal{R}$ so that $\mathcal{R}$ violates the specification $\mathcal{A}$. The algorithm then outputs "Counterexample w" (Line 13). If, on the other hand, $\mathcal{R}$ happens to agree with $\mathcal{A}$ on a negative classification of w, then there is a mismatch between $\mathcal{R}$ and the hypothesis $\mathcal{H}$ (Line 14). In that case, w is fed back to L^* to refine $\mathcal{H}$.

Now, let us consider the case that $\mathcal{L}(\mathcal{H}) \subseteq \mathcal{L}(\mathcal{A})$ holds (Line 5). If, in addition, we can establish $\mathcal{L}(\mathcal{R}) \subseteq \mathcal{L}(\mathcal{H})$, we conclude that $\mathcal{L}(\mathcal{R}) \subseteq \mathcal{L}(\mathcal{A})$ and output "Property satisfied" (Line 8). This inclusion test (Line 6) relies on statistical model checking using given parameters $\varepsilon, \gamma > 0$ (cf. Algorithm SMC). If the test passes, we have some statistical guarantee of correctness of $\mathcal{R}$ (cf. Theorem 10.3.1). Otherwise, we obtain a word $w \in \mathcal{L}(\mathcal{R}) \setminus \mathcal{L}(\mathcal{H})$ witnessing a discrepancy between $\mathcal{R}$ and $\mathcal{H}$ that will be exploited to refine $\mathcal{H}$ (Line 9).

Overall, in the event that the algorithm terminates, we have the following theorem that assures the soundness of a returned counterexample and provides the statistical guarantees on the property satisfaction, depending on the result of the algorithm:

Theorem 10.4.1 (Correctness of PDV)**.** *If the Algorithm PDV terminates then it outputs "Counterexample w", then w is mistakenly classified by $\mathcal{R}$ as positive. If it outputs "Property satisfied", then $\mathcal{R}$ is ε-approximately correct w.r.t. $\mathcal{A}$ with probability at least $1 - \gamma$.*

Proof. Suppose the Algorithm PDV outputs "Counterexample w" in Line 13. Due to Lines 11 and 12, we have $w \in \mathcal{L}(\mathcal{R}) \setminus \mathcal{L}(\mathcal{A})$. Thus, w is a counterexample.

Suppose the algorithm outputs "Property satisfied" in Line 8. By Lines 6 and 7, $\mathcal{R}$ is ε-approximately correct w.r.t. $\mathcal{H}$ with probability at least $1 - \gamma$. That is, $\Pr(\mathcal{L}(\mathcal{R}) \setminus \mathcal{L}(\mathcal{H})) < \varepsilon$ with the probability $1 - \gamma$. Moreover, by Line 4, $\mathcal{L}(\mathcal{H}) \subseteq \mathcal{L}(\mathcal{A})$. This implies that $\mathcal{L}(\mathcal{R}) \setminus \mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{R}) \setminus \mathcal{L}(\mathcal{H})$ and, therefore,

$\Pr(\mathcal{L}(\mathcal{R}) \setminus \mathcal{L}(\mathcal{A})) \leq \Pr(\mathcal{L}(\mathcal{R}) \setminus \mathcal{L}(\mathcal{H}))$. We deduce that $\mathcal{R}$ is ε-approximately correct w.r.t. $\mathcal{A}$ with probability at least $1 - \gamma$. □

Although we cannot hope that Algorithm PDV will always terminate, we demonstrate empirically that it terminates on many examples. It terminates on all of our experiments (a couple of hundreds) we preformed for Section 10.6.

10.5 Adversarial Robustness Certification

Adversarial robustness certification aims to prove that an RNN is robust to small perturbations of their inputs by an adversary to change the classification of the word. Our method can especially be used for adversarial robustness certification, which is parameterized by a distance function $dist : \Sigma^* \times \Sigma^* \to [0, \infty]$. Recall that a distance function is a function that satisfies:, for all words $w_1, w_2, w_3 \in \Sigma^*$: (i) $dist(w_1, w_2) = 0$ if and only if $w_1 = w_2$, (ii) $dist(w_1, w_2) = dist(w_2, w_1)$, and (iii) $dist(w_1, w_3) \leq dist(w_1, w_2) + dist(w_2, w_3)$. Popular distance functions are *Hamming distance* and *Levenshtein distance.* The Hamming distance between two words $w_1, w_2 \in \Sigma^*$ is the number of positions in which w_1 differs from w_2, provided $|w_1| = |w_2|$ (otherwise, the distance is ∞). The Levenshtein distance between w_1 and w_2 is the minimal number of operations among substitution, insertion, and deletion that are required to transform w_1 into w_2. For $\mathcal{L} \subseteq \Sigma^*$ and $r \in \mathbb{N}$, we let $\mathcal{N}_r(\mathcal{L}) = \{w' \in \Sigma^* \mid dist(w, w') \leq r \text{ for some } w \in \mathcal{L}\}$ be the *r-neighborhood* of $\mathcal{L}$. If $\mathcal{L}$ is regular and *dist* is the Hamming or Levenshtein distance, then $\mathcal{N}_r(\mathcal{L})$ is regular (for efficient constructions of *Levenshtein automata* when $\mathcal{L}$ is a singleton, see [144]). Let $\mathcal{R}$ be an RNN, $\mathcal{L} \subseteq \Sigma^*$ be a regular language such that $\mathcal{L} \subseteq \mathcal{L}(\mathcal{R})$, $r \in \mathbb{N}$, and $0 < \varepsilon < 1$. We call $\mathcal{R}$ *ε-adversarially robust* (w.r.t. $\mathcal{L}$ and r) if $\Pr(\mathcal{N}_r(\mathcal{L}) \setminus \mathcal{L}(\mathcal{R})) < \varepsilon$. Accordingly, every word from $\mathcal{N}_r(\mathcal{L}) \setminus \mathcal{L}(\mathcal{R})$ is an *adversarial example.* Thus, checking adversarial robustness amounts to checking the inclusion $\mathcal{L}(\overline{\mathcal{R}}) \subseteq \overline{\mathcal{N}_r(\mathcal{L})}$ through one of the above-mentioned algorithms.

Note that, even when $\mathcal{L}$ is a finite set, $\mathcal{N}_r(\mathcal{L})$ can be too large for exhaustive exploration so that PDV, in combination with SMC, is particularly promising, as we demonstrate in our experimental evaluation in Section 10.6.

From the definitions and Theorem 10.4.1, we get:

Lemma 10.5.1. *If Algorithm PDV, for input complement language of the RNN $\overline{\mathcal{R}}$ and a DFA $\mathcal{A}$ recognizing $\overline{\mathcal{N}_r(\mathcal{L})}$, terminates, then if it outputs "Counterexample w", then w is an adversarial example. Otherwise, $\mathcal{R}$ is ε-adversarially robust (w.r.t. $\mathcal{L}$ and r) with probability at least $1 - \gamma$.*

Similarly, we can also check whether the neighborhood of a regular set of words preserves a negative classification. Meaning, we can handle the case where given

$\mathcal{L} \cap \mathcal{L}(\mathcal{R}) = \emptyset$, then $\mathcal{R}$ is ε-*adversarially robust* w.r.t. $\mathcal{L}$ if $\Pr(\mathcal{L}(\mathcal{R}) \cap \mathcal{N}_r(\mathcal{L})) < \varepsilon$, and every word in $\mathcal{L}(\mathcal{R}) \cap \mathcal{N}_r(\mathcal{L})$ is an *adversarial example*. Overall, this case amounts to checking $\mathcal{L}(\mathcal{R}) \subseteq \overline{\mathcal{N}_r(\mathcal{L})}$.

10.6 Experimental Evaluation

We now present an experimental evaluation of the three algorithms SMC, AAMC, and PDV, and provide a comparison of their performance on LSTM networks [76] (a variant of RNNs using LSTM units). The algorithms have been implemented[1] in Python 3.6 using PyTorch 19.09 and NumPy library. The experiments of adversarial robustness certification were run on MacBook Pro 13 with the macOS. The other experiments were run on NVIDIA DGX-2 with an Ubuntu OS.

The general schema for the three types of experiments below is as follows:

1. Train an RNN $\mathcal{R}$ on a language $\mathcal{L}$;
2. Generate specification in the form of a DFA $\mathcal{A}$;
3. Use the SMC, AAMC, and PDV to check whether $\mathcal{R}$ fulfills the specifications.

Optimization For Equivalence Queries. In [115], the authors implement AAMC but with an optimization that was originally shown in [10]. This optimization concerns the number of samples required for checking the equivalence between the hypothesis and the taught language. This number depends on ε, γ and the number of previous equivalence queries n and is calculated by $\frac{1}{\varepsilon}\left(\log\frac{1}{\gamma} + \log(2)(n+1)\right)$. We adopt this optimization in AAMC and PDV as well (Algorithm AAMC in Line 1 and Algorithm PDV in Line 6).

10.6.1 Evaluation on Randomly Generated DFA

Synthetic Benchmarks. To compare the algorithms, we implemented the following procedure, which generates a random DFA $\mathcal{A}_{\mathsf{rand}}$, an RNN $\mathcal{R}$ that learned $\mathcal{L}(\mathcal{A}_{\mathsf{rand}})$, and a finite set of specification DFA:

(1) choose a random DFA $\mathcal{A}_{\mathsf{rand}} = (Q, \delta, q_0, F)$, with $|Q| \leq 30$, over an alphabet Σ with $|\Sigma| = 5$;
(2) randomly sample words from Σ^* as described in Section 10.2 in order to create a training set and a test set;
(3) train an RNN $\mathcal{R}$ with hidden dimension $20|Q|$ and $1 + |Q|/10$ layers—if the accuracy of $\mathcal{R}$ on the training set is larger than 95%, continue, otherwise restart the procedure;

[1] available at `https://github.com/LeaRNNify/Property-directed-verification`

Table 10.1: Experimental results

Type	*Avg time* (s)	*Avg len*	*#Mistakes*	*Avg MQs*
SMC	92	111	**122**	286063
AAMC	444	**7**	30	3701916
PDV	**21**	11	109	**28318**

(4) choose randomly up to five sets $F_i \subseteq Q \setminus F$ to define specification DFA $\mathcal{A}_i = (Q, \delta, q_0, F \cup F_i)$.

Using this procedure, we created 30 DFAs/RNNs and 138 specifications.

Experimental Results. Given an RNN R and a specification DFA $\mathcal{A}$, we checked whether R satisfies $\mathcal{A}$ using Algorithms 1–3, i.e., SMC, AAMC, and PDV, with $\varepsilon, \gamma = 5 \cdot 10^{-4}$.

Table 10.1 summarizes the executions of the three algorithms on our 138 random instances. The columns of the table are as follows:

(i) *Avg time* was counted in seconds and all the algorithms were timed out after 10 minutes;
(ii) *Avg len* is the average length of the found counterexamples (if one was found);
(iii) *#Mistakes* is the number of random instances for which a mistake was found;
(iv) *Avg MQs* is the average number of membership queries asked of the RNN.

Note that not only is PDV faster and finds more errors than AAMC, the average number of states of the final DFA is also much smaller: **26** states with PDV and **319** with AAMC. Furthermore, it asked more than 10 times less MQs to the RNN. Comparing PDV to SMC, it is 4.5 times faster and the average length of counterexamples it found is 10 times smaller, although it has fewer mistakes discovered.

Faulty Flows. One of the advantages of extracting DFA in order to detect mistakes in a given RNN is the possibility to find not only one mistake but a "*faulty flow*". For example, Figure 10.1 shows a DFA which was extracted with PDV, based on which we found a mistake in the corresponding RNN. The counterexample we found was *abcee*. One can see that the word *abce* is a loop in the DFA. Hence, we can suspect that this could be a "faulty flow". Checking the words $w_n = (abce)^n e$ for $n \in \{1, \ldots, 100\}$, we observed that, for any $n \in \{1, \ldots, 100\}$, the word w_n was in the RNN language but not in the specification.

To automate the reasoning above, we developed the following procedure: Given an RNN $\mathcal{R}$, a specification $\mathcal{A}$, the extracted DFA $\mathcal{H}$, and the counterexample w:

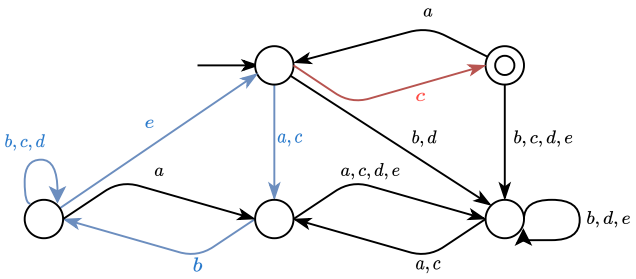


Figure 10.1: Faulty Flow in DFA extracted through PDV

(1) build the cross product DFA $\mathcal{H} \times \overline{\mathcal{A}}$;
(2) for every prefix w_1 of the counterexample $w = w_1 w_2$, denote by s_{w_1} the state to which the prefix w_1 leads in $\mathcal{H} \times \overline{\mathcal{A}}$. For any loop ℓ starting from s_{w_1}, check if $w_n = w_1 \ell^n w_2$ is a counterexample for $n \in \{1, \ldots, 100\}$;
(3) if w_n is a counterexample for more than 20 different n's, declare that a "faulty flow" was found.

Using this procedure, we managed to find faulty flows in 81/109 of the counterexamples that were found by PDV.

10.6.2 Adversarial Robustness Certification

We also examined PDV for adversarial robustness certification, following the ideas explained in Section 10.5, both on synthetic and real-world examples.

Synthetic Benchmarks. For a given DFA (representing one of the languages described below), we randomly sampled words from Σ^* by using the DFA and created a training set and a test set. For RNN training, we proceeded like in step (3) for the benchmarks in Section 10.6.1. Moreover, for certification, we randomly sampled 100 positive words and 100 negative words from the test set. For a given word w, we then let $\mathcal{L} = \{w\}$ and considered $\mathcal{N}_r(\mathcal{L})$ where $r = 1, \ldots, 5$.

Given an RNN $\mathcal{R}$, we checked whether $\mathcal{R}$ satisfies adversarial robustness using the certification methods PDV, SMC, and *neighborhood-automata generation SMC (NAG-SMC)*, with $\varepsilon, \gamma = 0.01$. In SMC, we randomly modified the input word within a certain distance to generate words in the neighborhood. In NAG-SMC, on the other hand, we first generated a neighborhood automaton of the input word, and sampled words that are accepted by the automaton. Here, we followed the algorithm by Bernardi and Giménez [19], who introduce a method for generating uniformly random words of length n in a given regular language with mean time bit-complexity $O(n)$.

Figure 10.2, which is a set of scatter plots, shows the results of the average time of executing the algorithms on the languages that we describe below. The x-axis and y-axis are both time in seconds, and each data point represents one adversarial robustness certification procedure. The length of words are from 50 to 500 and follow the normal distribution.

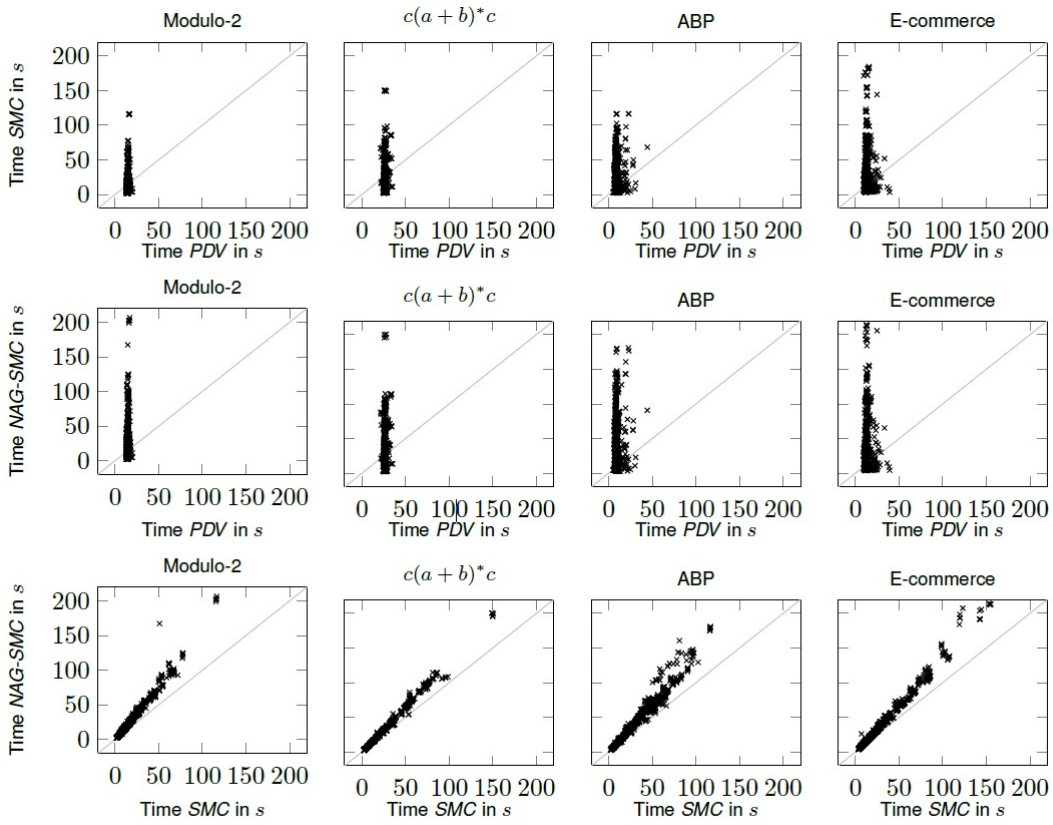


Figure 10.2: Comparing the three algorithms

Simple Regular Ranguages. As a sanity check of our approach, we considered the following two regular languages and distance functions:

$\mathcal{L}_1 = ((a+b)(a+b))^*$ (also called *modulo-2 language*) with Hamming distance;
$\mathcal{L}_2 = c(a+b)^*c$ with a distance function *dist* such that $dist(w_1, w_2)$ is the Hamming distance if $w_1, w_2 \in \mathcal{L}_2$ and $|w_1| = |w_2|$, and $dist(w_1, w_2) = \infty$ otherwise.

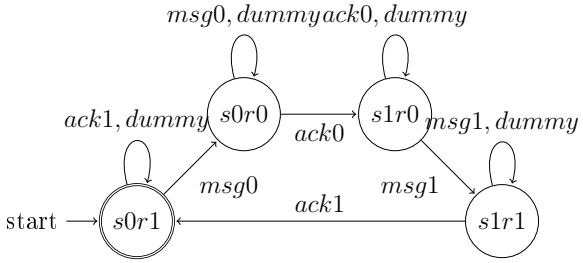


Figure 10.3: Automaton for ABP

The size of the Hamming neighborhood will exponentially grow with the distance.

The accuracies of the trained RNNs reached 100%. All three approaches successfully reported "adversarially robust" for the certified RNNs.

The first two diagrams on the first row of Figure 10.2 compare the runtimes of PDV and SMC on the two regular-language datasets, resp. whereas the first two diagrams on the second row compare the runtimes of PDV and NAG-SMC. We make two main observations. First, on average, the running time of PDV (avg. 15.70 seconds) is shorter than SMC (avg. 24.04 seconds) and NAG-SMC (avg. 32.5 seconds), which shows clearly that combining symbolically checking robustness on the extracted model and statistical approximation checking is more efficient than pure statistical approaches. Second, although SMC and NAG-SMC are able to certify short words (whose length is smaller than 30) faster, when the length of words is greater, they have to spend more time (which is more than 60 seconds) for certification. This is because, for short words, statistical approaches can easily explore the whole neighborhood, but when the neighborhood becomes larger and larger, this becomes infeasible.

The first two diagrams on the third row of Figure 10.2 compare the running time of SMC and NAG-SMC, respectively. In general, NAG-SMC is slower than SMC, this is mainly because, for sampling random words from the neighborhood, using the algorithm proposed by Bernardi et al. [19] is slower than combining the *random.choice* function in the Python library and the corresponding modification.

Real-World Dataset. We used two real-world examples considered by Mayr and Yovine [115]. The first one is the alternating-bit protocol (ABP) shown in Figure 10.3. However, we add a special letter *dummy* in the alphabet and a self-loop transition labeled with *dummy* on every state. We use the number of insertions of the letter *dummy* as the distance function. The second example is a variant of an example from the E-commerce website [119]. There are seven letters in the original automaton. Similarly, we also add *dummy* and self-loop transition

in every state (omitted in the figure for simplicity). Again, we use the number of insertions of *dummy* as the distance function.

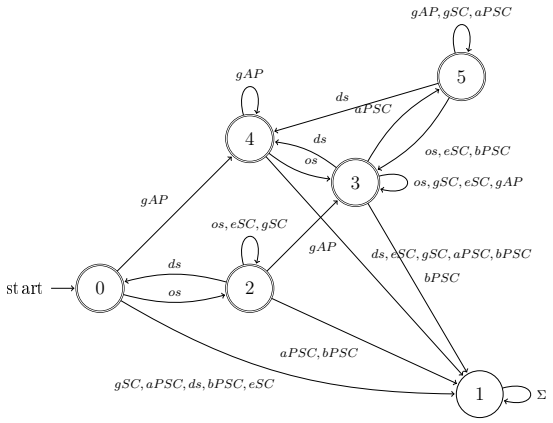


Figure 10.4: Automaton for e-commerce

The accuracies of the trained RNNs also reach 100%. For certification, the three approaches can certify the adversarial robustness for the RNNs as well.

The last two diagrams on the first (resp. second) row of Figure 10.2 compare the runtime of PDV and SMC (resp. PDV and NAG-SMC) on the ABP and the E-commerce dataset. The data points in the first and second row have a vertical shape. The reason is that the running time of PDV is usually relatively stable (10–20 seconds), while the running time of SMC and NAG-SMC increases linearly with the word length.

The last two diagrams on the third row of Figure 10.2 compare the runtimes of SMC and NAG-SMC on the two datasets. Here, the data points have a diagonal shape, but for NAG-SMC, when the word length is long (more than 300), it usually spends more time than SMC. This is mainly because it is inefficient to construct the neighborhood automaton and sample random words from the neighborhood.

10.6.3 RNNs Identifying Contact Sequences

Contact tracing [89] has proven to be increasingly effective in curbing the spread of infectious diseases. In particular, analyzing contact sequences—sequences of individuals who have been in close contact in a certain order—can be crucial in identifying individuals who might be at risk during an epidemic. We, thus, look at RNNs which can potentially aid contact tracing by identifying possible contact sequences. However, in order to deploy such RNNs in practice, one would require them to be verified adequately. One does not want to alert individuals unnecessarily even if they are safe, or overlook individuals who could be at risk.

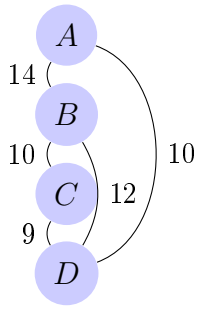


Figure 10.5: Temporal Network for contact between 4 people

In a real-world setting, one would obtain contact sequences from contact-tracing information available from, for instance, contact-tracing apps. However, such data is often difficult to procure due to privacy issues. Thus, in order to mimic a real life scenario, we use data available from `www.sociopatterns.org`, which contains information about interaction of humans in public places (hospitals, schools, etc.) presented as temporal networks.

Formally, a *temporal network* $G = (V, E)$ [78] is a graph consisting of a set of vertices V and a set of labeled edges E (i.e. $E = V \times V \times \mathbb{N}$), where the labels represent the timestamp during which the edge was active Figure 10.5 is a simple temporal network, which can be perceived as contact graph of four workers in an office where edge labels represent the time of meeting between them. A *time-respecting path* $\pi \in V^*$—a sequence of vertices such that there exists a sequence of edges with increasing time labels—depicts a contact sequence in such a network. In the example in Figure 10.5, $CDAB$ is a time-respecting path while $ABCD$ is not.

Benchmarks. For our experiment, given a temporal network G, we generated an RNN R recognizing contact sequences as follows:

1. We created training and test data for the RNN by generating (i) valid time-respecting paths (of lengths between 5 and 15) using labeled edges from G, and (ii) invalid time-respecting paths, by considering a valid path and randomly introducing breaks in the path. The number of time-respecting paths in the training set is twice the size of the number of labeled edges in G, while the test set is one-fifth the size of the training set.
2. We trained RNN $\mathcal{R}$ with hidden dimension $|V|$ (minimum 100) as well as $\lfloor 2 + |V|/100 \rfloor$ layers on the training data. We considered only those RNNs that could be trained within 5 hours with high accuracy (avg. 99%) on the test data.

3. We used a DFA that accepts all possible paths (disregarding the time labels) in the network as the specification, which would allow us to check whether the RNN learned unwanted edges between vertices.

Using this process, from the seven temporal networks taken from `www.sociopatterns.org`, we generated seven RNNs and seven specification DFA. We ran SMC, PDV, and AAMC on the generated RNNs, using the same parameters as used for the random instances.

Table 10.2: Results of model-checking algorithm on RNN identifying contact sequences

Case	*Alg.*	*Counter-example len.*	*Extracted DFA size*	*Time* (s)	*Case*	*Alg.*	*Counter-example len.*	*Extracted DFA size*	*Time* (s)
Across	SMC	3		0.3	Within	SMC	2		0.28
Kenyan	AAMC	2	328	624.76	Kenyan	AAMC	2	178	620.30
Household	PDV	2	2	0.22	Household	PDV	2	2	0.27
	SMC	2		0.23		SMC	71		1.51
Workplace	AAMC	2	111	604.99	Conference	AAMC	2	38	876.19
	PDV	2	2	0.77		PDV	2	2	0.33
	SMC	5		0.33		SMC	3		0.48
Highschool	AAMC	2	91	627.30	Workplace	AAMC	2	87	621.44
2011	PDV	2	2	0.19	2015	PDV	2	2	1.11
	SMC	7		0.24					
Hospital	AAMC	2	36	614.76					
	PDV	2	2	0.006					

Results. Table 10.2 notes the length of counterexample, the extracted DFA size (only for PDV and AAMC), and the running time of the algorithms. We make three main observations.

First, the counterexamples obtained by PDV and AAMC (avg. length 2), are much more succinct than those by SMC (avg. length 13.1). Small counterexamples help in identifying the underlying error in the RNN, while long and random counterexamples provide much less insight. For example, from the counterexamples obtained from PDV and AAMC, we learned that the RNN overlooked certain edges or identified wrong edges. This result highlights the demerit of SMC, which has also been observed in [156].

Second, the running time of SMC and PDV (avg. 0.48 seconds and 0.41 seconds) is comparable, while that of AAMC is prohibitively large (avg. 655.68 seconds), indicating that model checking on small and rough abstractions of the RNN produces superior results.

Third, the extracted DFA size, in case of AAMC (avg. size 124.14), is always larger compared to PDV (avg. size 2), indicating that RNNs are quite difficult to

be approximated by small DFA and this slows down the model-checking process as well. Again, our experiments confirm that PDV produces succinct counterexamples reasonably fast.

Chapter 11

Analyzing Robustness of Angluin's Algorithm

11.1 Introduction

Discrete-event systems and their languages. Discrete-event systems [31] is a large class of dynamic systems that given some internal state evolve from one state to another one due to the occurrence of an event. For instance, discrete-event systems can represent both a cyber-physical process whose events are triggered by the controller or the environment and a business process whose events are triggered by human activities or software executions. Often the behaviors of such systems are classified as safe (aka correct, representative, etc.) or unsafe. Since a behavior may be identified by its sequence of occurred events, this leads to the notion of language.

Analysis versus synthesis. There are numerous formalisms to specify (languages of) discrete-event systems. From a designer point of view, the simpler it is the better its analysis will be. So finite automata and their languages (regular languages) are good candidates for the specification. However, even when the system is specified by an automaton, its implementation may slightly differ due to several reasons (bugs, unplanned human activities, unpredictable environment, etc.). Thus, one generally checks whether the implementation conforms to the specification. However, in many contexts, the system has already been implemented and the original specification (if any) is lost, as for instance in the framework of process mining [155]. Thus, by observing and interacting with the system, one aims to recover a specification close to the original specification or at least that is robust with respect to the pathologic behaviors of the system.

Language learning. The problem of learning a language from its finite samples of strings by discovering the corresponding grammar is known as grammar

inference, whose significance was initially stated in [145] and an overview of its very first results can be found in [22]. As it may not always be possible to infer a grammar that exactly identifies a language, an approximate language learning was introduced in [158], where a grammar is selected from a solution space whose language approximates the target language with a specified degree of accuracy. To provide deeper insight into language learning, the problem of identifying a (minimum) deterministic finite automata (DFA) that is consistent with a given sample has attracted a lot of attention in the literature since several decades [67, 10, 149]. An understanding of regular language learning is very valuable for a generalization to other more complex classes of languages.

Angluin's L* algorithm. Recall, Angluin's L* algorithm learns the minimal DFA of a regular language using membership and equivalence queries. Thus, one could try to adapt it to the synthesis task described above. However, for most of the systems seen as black boxes, the equivalence query cannot be implemented. Thus, its probabilistic approximately correct (PAC) version substitutes an equivalence query by an enough large set of random membership queries. Then the main issue is to define and evaluate the accuracy of such a learning in this context. More precisely, here we are interested in how the PAC Angluin algorithm behaves for devices which are obtained from a DFA by introducing some noise.

Noisy learning. Most learning algorithms in the literature assume the correctness of the training data, including the example data such as attributes as well as classification results. However, sometimes the noise-free datasets are not available. Quinlan [130] carried out an experimental study of the noise effects on learning performance, whose results showed that the classification noise had more negative impact than the attribute one. Angluin and Laird [11] studied how to compensate for randomly introduced noise and discovered a theorem giving a size bound of a sample that is sufficient for PAC-identification in the presence of classification noise when the concept classes are finite. Michael Kearns formalized another related learning model from statistical queries by extending Valiant's learning model [87]. One main result shows that any class of functions learnable from this statistical query model is also learnable with classification noise in Valiant's model, as far as the noise rate is smaller than $\frac{1}{2}$.

Our contribution. In this chapter, we study against which kinds of noise the Angluin algorithm is robust, which is the very first attempt of noise analysis in the automata learning setting, to the best of our knowledge. To this end, we introduce three kinds of *noisy devices* obtained from a DFA: (1) either the noisy device is a random language obtained from a given DFA by reversing the word acceptance with a small probability, which corresponds to the classification noise in the classical learning setting, (2) either the noisy device, also with a small probability, replaces each letter of a word example by one chosen uniformly from

the alphabet, which corresponds to the attribute noise in the classical setting, (3) or the noisy DFA combines the status of a word w.r.t. the DFA and its status w.r.t. a counter automaton. Our studies are based on the distribution over words that is used for generating words associated with membership queries and defining (and statistically measuring) the *distance* between two devices as the probability that they differ on word acceptance. We have performed experiments over several hundreds of random DFA. We have pursued several goals along our experiments. The three first ones are related to methodological and efficiency issues, while the two last ones are related to the results of our experiments:

- How to choose the accuracy of the approximate equivalence query to get a good trade-off between accuracy and efficiency?
- Since in most of the cases, Angluin's algorithm may perform a huge number of refinement rounds before a possible termination, what is a "good" number of rounds to stop the algorithm avoiding underfitting and overfitting?
- What is the threshold (in terms of distance) between pertubating the DFA or producing a device that is no more "similar to" the DFA?
- What is the impact of the nature of noise on the robustness of Angluin's algorithm?
- What is the impact of the words distribution on the robustness of Angluin's algorithm?

We experimentally show that w.r.t. the random noise, both for input and output, Angluin's algorithm behaves quite well, i.e., the learned DFA is very often closer to the original one than the noisy random language while when the noise is obtained by the counter automaton, the Angluin's algorithm is not robust. Moreover, we establish that the expectation of the length of a random word should be enough large to cover a relevant part of the set of words in order for Angluin's algorithms to be robust.

In order to understand why Angluin's algorithm is robust w.r.t. random noise we have undertaken a theoretical study establishing that almost surely the languages of the output noisy device is not recursively enumerable. The same result holds for the input noisy devices under a slight condition on the original DFA. This confirms that the less the noise is structured, the more robust is Angluin's algorithm.

Organization. In Section 11.2, we introduce the technical background required for the robustness analysis. In Section 11.3, we detail the goals and the settings of our analysis. In Section 11.4, we provide and discuss the experimental results. In Section 11.5, we relate randomness with structuration.

11.2 Preliminaries

Here we provide the technical background required for the robustness analysis. Note that, the basic definition for languages, finite automata, can be found in Section 2.3. Even thought we defined Angluin's L* and its PAC version in Section 2.3, we recall its definition here and make a *small modification* to it in order to insure its termination on languages which are not regular.

Notation change. Note that, in all Part III we use λ to denote the empty word instead ε.

Words distribution and measure of a language. A distribution D over Σ^* is defined by a mapping $\mathbf{Pr}_D$ from Σ^* to $[0,1]$ such that $\sum_{w\in\Sigma^*}\mathbf{Pr}_D(w) = 1$. Let L be a language its probabilistic measure w.r.t. D, $\mathbf{Pr}_D(L)$ is defined by $\mathbf{Pr}_D(L) = \sum_{w\in L}\mathbf{Pr}_D(w)$.

Our analysis requires that we should be able to efficiently sample a word according to some D. Thus, we only consider the distributions D_μ where $\mu \in (0,1)$ defined for a word $w = a_1 \dots a_n \in \Sigma^*$ by:

$$P_{D_\mu}(w) = \mu\left(\frac{1-\mu}{|\Sigma|}\right)^n$$

In practice to sample a random word according to D_μ, we start with the empty word and iteratively we flip a biased coin with probability $1-\mu$ to add a letter (and μ to return the current word) and then uniformly select the letter in Σ.

Language distance. Given languages L_1 and L_2 their distance w.r.t. distribution D, $d_D(L_1, L_2)$ is defined by: $d_D(L_1, L_2) = \mathbf{Pr}_D(L_1 \Delta L_2)$. Computing the distance between languages is in most of the cases impossible. Fortunately whenever the membership problem for L_1 and L_2 are decidable, then using Chernoff-Hoeffding bounds [77], this distance can be statistically evaluated as follows. Let $e, d > 0$ be an error parameter and a confidence level, respectively. Let S be a set of words sampled independently according to D such that $|S| \geq \frac{\log(2/d)}{2e^2}$. Let $dist = \frac{|S\cap(L_1\Delta L_2)|}{|S|}$. Then:

$$\mathbf{Pr}_D(|d_D(L_1,L_2) - dist| > e) \;<\; d\,.$$

Since we will not simultaneously discuss several distributions, we will alleviate the notations and omit the subscript D almost everywhere.

PAC version of Angluin's L* algorithm. Given a regular language L, Angluin's L* algorithm learns the unique minimal DFA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = L$ using only membership queries 'Does w belong L?' and equivalence queries 'Does $\mathcal{L}(\mathcal{A}) = L$? and if not provide a word $w \in L\Delta\mathcal{L}(\mathcal{A})$'. An abstract version of this algorithm is depicted by Algorithm 12. The main features of this algorithm are: a data $Data$ from which `Synthesize`$(Data)$ returns an automaton $\mathcal{A}$ and such that given a

word $w \in L\Delta\mathcal{L}(\mathcal{A})$, `Update`$(Data, w)$ updates $Data$. The number of states of $\mathcal{A}$ is incremented by one after each round, and so the algorithm terminates after its number of states is equal to the (unknown) number of states of the minimal DFA recognizing L.

The Probably Approximately Correct (PAC) version of Angluin's L* algorithm takes as input an error parameter ε and a confidence level δ replaces the equivalence query by a number of membership queries '$w \in L\Delta\mathcal{L}(\mathcal{A})$?' where the words are sampled from some distribution D unknown to the algorithm. Thus, this algorithm can stop too early when all answers are negative while $L \neq \mathcal{L}(A)$. However, due to the number of such queries which depends on the current round r (i.e., $\lceil \frac{log(1/\delta)+(r+1)\log(2)}{\varepsilon} \rceil$) this algorithm ensures that:

$$\mathbf{Pr}_D(d_D(L, \mathcal{L}(\mathcal{A})) > \varepsilon) \ < \ \delta \, .$$

A key observation is that this algorithm could be used for every language L for which the membership problem is decidable. However, since L is not necessarily a regular language it could never stop and thus our adaptation includes a parameter *maxround* that ensures termination.

Algorithm 12: Angluin's L* algorithm

Input: L, a language unknown to the algorithm
Input: an integer *maxround* ensuring termination

`Angluin`()

Data: an integer r, a boolean b and a data $Data$
Output: a DFA $\mathcal{A}_E$

`Initialize`$(Data)$; $r \leftarrow 0$

`// The control of` *maxround* `is unnecessary when` L `is regular`
while $r <$ *maxround* **do**
    $\mathcal{A}_E \leftarrow$ `Synthesize`$(Data)$
    $(b, w) \leftarrow$ `IsEquivalent`$(\mathcal{A}_E)$
    if b **then return** $\mathcal{A}_E$
    `Update`$(Data, w)$; $r \leftarrow r + 1$
end
return `Synthesize`$(Data)$

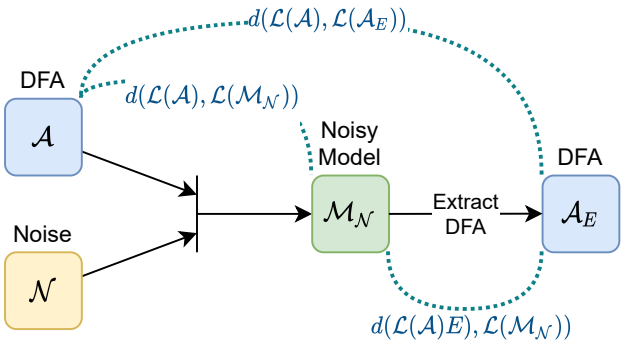


Figure 11.1: Comparison between $d(\mathcal{L}(\mathcal{A}), \mathcal{L}(\mathcal{M}_\mathcal{N}))$ and $d(\mathcal{L}(\mathcal{A}), \mathcal{L}(\mathcal{A}_E))$

11.3 Robustness Analysis

11.3.1 Principle and goals of the analysis

Principle of the analysis. Figure 11.1 illustrates the whole process of our analysis. First we set the qualitative and quantitative nature of the noise ($\mathcal{N}$). Then we generate a set of random DFA ($\mathcal{A}$). Combining $\mathcal{A}$ and $\mathcal{N}$, one gets a noisy model $\mathcal{M}_\mathcal{N}$. More precisely, depending on whether the noise is random or not, $\mathcal{M}_\mathcal{N}$ is either generated off-line (deterministic noise) or on-line (random noise) when a membership query is asked during Angluin's L* algorithm. Finally, we compare the distances (1) between $\mathcal{A}$ and $\mathcal{M}_\mathcal{N}$, and (2) between $\mathcal{A}$ and $\mathcal{A}_E$, the automaton returned by the algorithm. The aim of this comparison is to establish whether the $\mathcal{A}_E$ is closer to $\mathcal{A}$ than $\mathcal{M}_\mathcal{N}$. In order to get a quantitative measure, we define the *information gain* as:

$$\text{Information gain} = \frac{d(\mathcal{L}(\mathcal{A}), \mathcal{L}(\mathcal{M}_\mathcal{N}))}{d(\mathcal{L}(\mathcal{A}), \mathcal{L}(\mathcal{A}_E))}$$

We consider, low information gain to be in $[0, 0.9]$, medium information gain to be in $[0.9, 1.5]$, high information gain to be in $[1.5, \infty)$.

In addition, we also evaluate the distance between $\mathcal{A}_E$ and $\mathcal{M}_\mathcal{N}$ in order to study in which cases the algorithm learns in fact the noisy model instead of the original DFA.

Goals of the analysis. The two first goals are related to the tuning of the PAC Angluin's L* algorithm while the last ones are related to the robustness analysis.

- **Maximal number of rounds.** Since the running time of the algorithm quadratically depends on the number of rounds (i.e. iterations of the loop), selecting an appropriate maximal number of rounds is a critical issue. So

we let vary this maximal number of rounds and analyze how the reduction factor decreases w.r.t. this number.

- **Accurateness of the equivalence query.** As an equivalence query is replaced with a set of membership queries whose number depends on the current number of round and the pair (ε, δ), it is thus interesting to study (1) what is the effect of (ε, δ) on the ratio of executions that reach the maximal number of rounds and (2) compare the reduction factor for executions that stop before reaching this maximal number and the same execution when letting it run up to this maximal number.
- **Quantitative analysis.** The reduction factor highly depends on the 'quantity' of the noise, i.e., noise rate. So we analyze the reduction factor depending on the distance between the original DFA and the noisy device and want to identify a threshold (if any) where the reduction factor starts to significantly increase.
- **Qualitative analysis.** Another important criterion of the reduction factor is the 'nature' of the noise. So we analyze the reduction factor w.r.t. the three noisy devices that we have introduced.
- **Impact of word distribution.** Finally, the robustness of this algorithm with respect to word distribution is also analyzed.

11.3.2 Settings

In order to empirically evaluate our ideas, we have implemented some benchmarks. These benchmarks were implemented in Python, using the NumPy libraries. It can be found on GitHub[1]. All benchmarks were performed on a computer equipped by Intel i5-8250U CPU with 4 cores, 16 GB of memory and Ubuntu Linux 18.03.

We now describe the settings of the experiments we made with three different types of noises. For this we implemented the PAC version of Kearns and Vazirani L^* algorithm [88].

We choose $\mu = 10^{-2}$ for the parameter of the word distribution so that the average length of a random word is 99. All the statistic distances were computed using the Chernoff-Hoeffding bound [77] with 10^{-3} as confidence level and $5 \cdot 10^{-4}$ as width.

The benchmarks were performed on DFA randomly generated using the following procedure. Let $M_q = 50$ and $M_a = 20$ be two parameters, upper bounds on the number of states and of the alphabet, that could be tuned in future experiments. The DFA $\mathcal{A} = (Q, \sigma, q_0, F)$ on Σ is generated as follows:

- Uniformly choose $n_q \in [10, M_q]$ and $n_a \in [3, M_a]$;

[1] `https://github.com/LeaRNNify/Noisy_Learning`

- Set $Q = [0, n_q]$ and $\Sigma = [0, n_a]$;
- Uniformly choose $n_f \in [0, n_q - 1]$ and let $F = [0, n_f]$;
- Uniformly choose q_0 in Q;
- For all $(q, a) \in Q \times \Sigma$, uniformly choose the state $\sigma(q, a)$.

We now describe the three kinds of noise that we analyze.

DFA with noisy output. Given a DFA $\mathcal{A}$ on the alphabet Σ and $0 < p < 1$, the random device $\mathcal{A}^{\to p}$ switches the status of words w.r.t. $\mathcal{A}$ with probability p. More formally for all word w,

$$\mathbf{Pr}(w \in \mathcal{L}(\mathcal{A}^{\to p})) = (1-p)\mathbf{1}_{w \in \mathcal{L}(\mathcal{A})} + p\mathbf{1}_{w \notin \mathcal{L}(\mathcal{A})}$$

Observe that the expected value of $d(\mathcal{L}(\mathcal{A}), \mathcal{L}(\mathcal{A}^{\to p}))$ is p. Moreover, in our experiments when we get that $\left|\frac{d(\mathcal{L}(\mathcal{A}), \mathcal{L}(\mathcal{A}^{\to p})) - p}{p}\right| < 5 \cdot 10^{-2}$ for all the generated DFA.

DFA with noisy input. Given a DFA $\mathcal{A}$ on the alphabet Σ (with $|\Sigma| > 1$) and $0 < p < 1$, the random device $\mathcal{A}^{\leftarrow p}$ change every letter of the word with probability p uniformly to another letter and then returns the status of the new word w.r.t. $\mathcal{A}$. More formally, let $w = w_1 \dots w_n$. Then:

$$\mathbf{Pr}(w \in \mathcal{L}(\mathcal{A}^{\leftarrow p})) = \sum_{|w'|=|w| \wedge w' \in \mathcal{L}(\mathcal{A})} \prod_{i \leq n} (1-p)\mathbf{1}_{w_i = w'_i} + \frac{p}{|\Sigma| - 1}\mathbf{1}_{w_i \neq w'_i}$$

Remark. In our implantation of both noisy input and output, in order to be consistent we keep a large hash table that holds all the words seen before and their randomly assigned output. This is currently the biggest bottleneck in our benchmarks, since for measuring statistical distance requires a huge set of samples, which in turn uses most of the memory.

Counter DFA. Let $\mathcal{A}$ be a DFA of the alphabet Σ and c be a function from $\Sigma \cup \{\lambda\}$ to $\mathcal{N}$. We inductively define the function $\overline{c}$ from Σ^* to $\mathcal{N}$ by:

$$\overline{c}(\lambda) = c(\lambda) \text{ and } \overline{c}(wa) = \overline{c}(w) + c(a)$$

The counter DFA is $\mathcal{A}_c$ works as follows. For all word w:

$$w \in \mathcal{L}(\mathcal{A}_c) \text{ if and only if } w \in \mathcal{L}(A) \text{ or } \overline{c}(w) \leq 0$$

In our experiments we randomly generate the counter function as follows:

- Uniformly choose $c(\lambda)$ in $[0, |\Sigma|]$;
- For all $a \in \Sigma$, $\mathbf{Pr}(c(a) = -1) = \frac{1}{4}$ and for all $0 \leq i \leq 6$, $\mathbf{Pr}(c(a) = i) = \frac{3}{28}$.

11.3.3 Tunings

Before launching our experiments, we need to tune two key parameters for both efficiency and accurateness purposes: the maximal number of rounds of the algorithm and the accuracy of the approximate equivalence query. This tuning is based on experiments over the DFA with the noisy output since the expected distance between the DFA and the noisy model is known (p), thus simplifying the tuning.

Maximal number of rounds. In order to specify a maximal number of rounds that lead to the good performances of the Angluin's Algorithm, we took a DFA with noisy output $\mathcal{A}^{\to p}$ for $p \in \{0.005, 0.0025, 0.0015, 0.001\}$. We ran the learning algorithm, stopping every 20 rounds to estimate the distance between the current DFA $\mathcal{A}_E$ to the original DFA $\mathcal{A}$. Figure 11.2 shows the evolution graphs of $d(\mathcal{L}(\mathcal{A}), \mathcal{L}(\mathcal{A}_E))$ w.r.t. the number of rounds according to the different values of p each of them summarizing five runs on five different DFA. The vertical axis corresponds to the distance to original DFA $\mathcal{A}$, and the horizontal axis corresponds to the number of rounds. The red line is the distance with$\mathcal{A}^{\to p}$, and the blue line is the distance with $\mathcal{A}_E$. We observe that after about 250 rounds $d(\mathcal{L}(\mathcal{A}), \mathcal{L}(\mathcal{A}_E))$ is

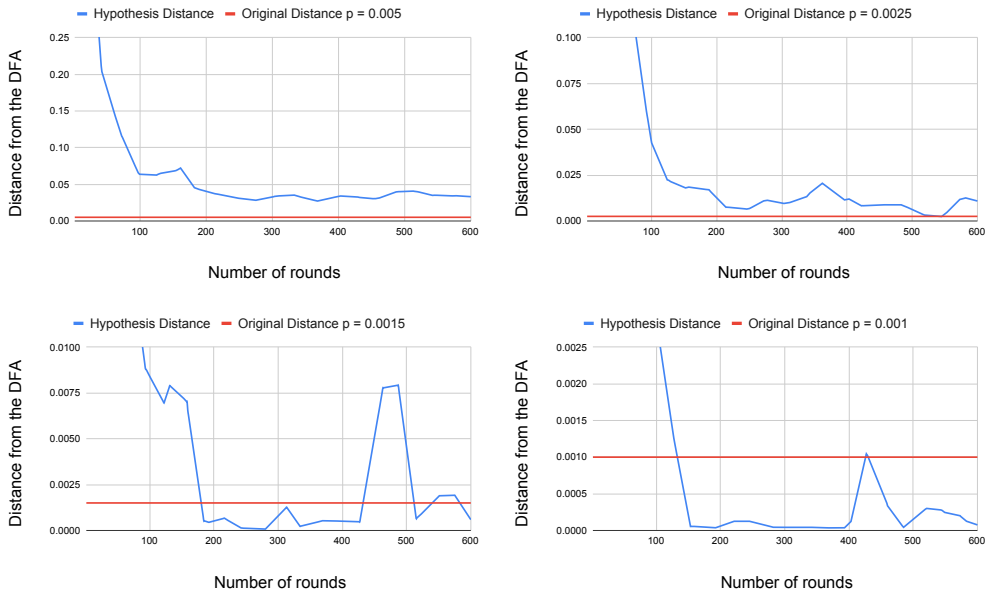


Figure 11.2: Number of rounds analysis

stabilizing. Therefore, from now on all the experiments are made with a maximum of 250 rounds. Of course this number depends on the size of $\mathcal{A}$, but for the variable size that we have chosen (between 10 and 50 states) it seems to be a good choice.

Accurateness of the approximate equivalence query. We have generated

thirty-five DFA and for each of them we generated five $\mathcal{A}^{\to p}$ with different values of p. Table 11.1 summarizes our results with different ε and δ for the approximate equivalence query. The rows correspond to the value of the noise p, the columns correspond to the values of ε and δ (where we always choose $\varepsilon = \delta$) and each cell shows the average information gain. Looking at this table, $\varepsilon = \delta = 0.01$ and $\varepsilon = \delta = 0.005$ seem to be optimal values. We decided to fix $\varepsilon = \delta = 0.005$ for all our experiments.

p \ $\varepsilon = \delta$	0.05	0.01	0.005	0.001	0.0005
0.01	0.081	0.054	0.047	0.048	0.050
0.005	0.086	0.087	0.072	0.070	0.094
0.0025	0.867	0.292	0.591	0.321	0.748
0.0015	1.401	2.933	3.082	0.980	0.710
0.001	5.334	4.524	3.594	1.811	6.440

Table 11.1: Evaluation of the impact of ε and δ.

11.4 Experimental Evaluation

11.4.1 Qualitative and Quantitative analysis

For the three types of noise we have generated several DFA and for each DFA we have generated several noisy models depending on the 'quantity' of noise. By computing the (average) information gain for all these experiments, we have been able to get conclusions about the effect of the nature and the quantity of the noise on the performance of Angluin's algorithm.

DFA with noisy output. We have generated fifty DFA $\mathcal{A}$ and for each of them we generated DFA with noisy output $\mathcal{A}^{\to p}$ with five p between 0.01 and 0.001. Table 11.2 summarizes the results. We have identified a threshold for p between 0.0015 and 0.0025, if the noise is above 0.0025 the resulting DFA A has a bigger distance to the original one $\mathcal{A}$ than $\mathcal{A}^{\to p}$, and smaller if the noise is under 0.0015. Moreover, once we cross the threshold the robustness of the algorithm increases very quickly. We have also included a column that represents the standard deviation of the random variable $d(\mathcal{L}(\mathcal{A}), \mathcal{L}(\mathcal{A}_E))$ to assess that our conclusions are robust w.r.t. the probabilistic feature.

p	$d(\mathcal{L}(\mathcal{A}), \mathcal{L}(\mathcal{A}_E))$	$d(\mathcal{L}(\mathcal{A}^{\to p}), \mathcal{L}(\mathcal{A}_E))$	gain	standard deviation
0.01	0.12625	0.13320	0.074	0.04102
0.005	0.04420	0.04827	0.11	0.03366
0.0025	0.00333	0.00568	0.75	0.00523
0.0015	0.00027	0.00174	5.5	0.00047
0.001	0.00006	0.00103	15.7	0.00007

Table 11.2: Evaluation of the algorithm w.r.t. the noisy output.

DFA with noisy input. We have generated forty-five random DFA $\mathcal{A}$ and for each of them we have generated several DFA with noisy-input $\mathcal{A}^{\leftarrow p}$ with $p \in \{10^{-4}, 5 \cdot 10^{-4}, 10^{-3}, 5 \cdot 10^{-3}\}$. Contrary to the case of noisy output, p does not correspond to the expected value of $d(\mathcal{A}, \mathcal{A}^{\leftarrow p})$. Thus, we have evaluated this distance for every pair of the experiments and we have gathered the pairs whose distances belong to intervals that are described in the first column of Table 11.3. The second column of this table reports the number of pairs in the interval while the third one reports the average value of this distance for these pairs. Again we identify a threshold for $d(\mathcal{A}, \mathcal{A}^{\leftarrow p})$ between 0.0015 and 0.0025 and once we cross the threshold the robustness of the algorithm increases very quickly.

Range	#	$d(\mathcal{A}, \mathcal{A}^{\leftarrow p})$	$d(\mathcal{A}, \mathcal{A}_E)$	$d(\mathcal{A}^{\leftarrow p}, \mathcal{A}_E)$	gain	standard deviation
[0.025,1]	36	0.04027	0.21513	0.22658	0.18	0.05279
[0.005,0.025]	53	0.00924	0.05416	0.06077	0.17	0.04172
[0.002,0.005]	33	0.00378	0.01260	0.01611	0.30	0.01783
[0.001,0.002]	11	0.00123	0.00030	0.00154	4.1	0.00058
[0.001,0.0005]	25	0.00079	0.00002	0.00082	39.5	0.00007

Table 11.3: Evaluation of the algorithm w.r.t. the noisy input.

Counter DFA. We have generated one hundred and sixty DFA and for each of them we have generated a counter automaton (as described above). The results of our experiments are given in Table 11.4. Here whatever the quantity of noise

the Angluin's algorithm is unable to get closer to the original DFA. Moreover, the extracted DFA $\mathcal{A}_E$ is very often closer to the counter automaton A_c than the original DFA $\mathcal{A}$.

Range	#	$d(\mathcal{A}, \mathcal{A}_c)$	$d(\mathcal{A}, \mathcal{A}_E)$	$d(\mathcal{A}_c, \mathcal{A}_E)$	gain	standard deviation
[0.005,0.025]	14	0.01238	0.02586	0.02053	0.47	0.01898
[0.005,0.002]	57	0.00245	0.00396	0.00262	0.61	0.00298
[0.001,0.002]	22	0.00143	0.00209	0.00121	0.68	0.00126
[0.0005,0.001]	20	0.00079	0.00108	0.00064	0.72	0.00065
[0.0001,0.0005]	44	0.00025	0.00035	0.00021	0.71	0.00021

Table 11.4: Evaluation of the algorithm w.r.t. the 'noisy' counter.

Thus, we conjecture that when the noise is 'unstructured' and the quantity is small enough such that the word noise is still meaningful, then Angluin's algorithm is robust. On the contrary, when the noise is structured then Angluin's algorithm 'tries to learn' the noisy model whatever the quantity of noise. In section 11.5, we will strengthen this conjecture establishing that in some sense random noise implies unstructured noise.

11.4.2 Words distribution

The parameter μ determines the average length of a random word ($\frac{1}{\mu} - 1$). Table 11.5 summarizes experimental results with values of μ indicated on the first row. The other rows correspond to different values of the noise p for $\mathcal{A}^{\to p}$. The cells (at the intersection of a pair (p,μ)) contain the (average) information gain where experiments were done over twenty-two DFA always eliminating the worst and best cases to avoid that the pathological cases perturb the average values. For values of p that matter (i.e., when the gain is greater than 1), there is clear tendency for the gain to first increase w.r.t. μ, reaching a maximum about $\mu = 0.01$ the value that we have chosen and then decrease. A possible explanation would be the following: too short words (i.e., big μ) does not help to discriminate between languages while too long words (i.e., small μ) lead to overfitting and does not reduce the noise.

p \ μ	.001	.005	0.01	0.05	0.1
0.01	0.059	0.067	0.078	0.184	0.317
0.005	0.078	0.130	0.134	0.559	0.966
0.0025	0.165	0.298	0.398	1.246	0.823
0.0015	0.465	0.671	2.267	2.074	1.651
0.001	1.801	10.94	8.907	3.753	2.341

Table 11.5: Analysis of different distributions on Σ^*

11.5 Random languages versus structured languages

In this section, we want to establish that the main factor of the robustness of the Angluin's L* algorithm w.r.t. random noise is that almost surely the perturbated language is unstructured. We consider a language as structured if it can be produced by some general device. Thus, we identify the family of structured languages with the family of recursively enumerable languages.

The following lemma gives a simple mean to establish that almost surely a random language is not recursively enumerable.

Lemma 11.5.1. *Let R be a random language over Σ. Let $(w_n)_{n\in\mathbb{N}}$ be a sequence of words of Σ^*.*
Denote $W_n = \{w_i\}_{i<n}$ and $\rho_n = \max_{W\subseteq W_n} \mathbf{Pr}(R \cap W_n = W)$.
Assume that $\lim_{n\to\infty} \rho_n = 0$.
Then for all countable family of languages $\mathcal{F}$, almost surely $R \notin \mathcal{F}$ and in particular almost surely R is not a recursively enumerable language.

Proof. Let us consider an arbitrary language L.
Then for all n, $\mathbf{Pr}(R = L) \leq \mathbf{Pr}(R \cap W_n = L \cap W_n) \leq \rho_n$.
Thus $\mathbf{Pr}(R = L) = 0$ and $\mathbf{Pr}(R \in \mathcal{F}) = \sum_{L\in\mathcal{F}} \mathbf{Pr}(R = L) = 0$. □

The proofs of the two next theorems use the same notations as those given in the lemma. From this lemma, we immediately obtain that almost surely the noisy output perturbation of any language is not recursively enumerable.

Theorem 11.5.2. *Let L be a language and $0 < p < 1$.*
Then almost surely $L^{\to p}$ is not a recursively enumerable language.

Proof. Consider any enumeration $(w_n)_{n\in\mathbb{N}}$ of Σ^* and any $W \subseteq W_n$.
The probability that $L^{\to p} \cap W_n$ is equal to W is bounded by $\max(p, 1-p)^n$.
Thus $\rho_n \leq \max(p, 1-p)^n$ and $\lim_{n\to\infty} \rho_n = 0$. □

We cannot get a similar result for the noisy input perturbation. Indeed, consider the language Σ^*, whatever the kind of noise brought to the input, the obtained language is still Σ^*. With the kind of input noise that we study, consider the language that accepts words of odd length (see the automaton $\mathcal{A}'$ of Figure 11.3). Then the perturbed language is unchanged.

However, given a DFA $\mathcal{A}$, we establish a slight condition on $\mathcal{A}$ which ensures that almost surely the random language $L(\mathcal{A}^{\leftarrow p})$ is not recursively enumerable. We abbreviate bottom strongly connected component (of $\mathcal{A}$ viewed as a graph) by BSCC.

Theorem 11.5.3. *Let Σ be an alphabet with $|\Sigma| > 1$. Let $\mathcal{A} = \langle Q, F, \sigma, q_0\rangle$ be a DFA over Σ, $0 < p < 1$ and $\mathcal{C}, \mathcal{C}'$ some BSCC of $\mathcal{A}$ (possibly equal). Assume that in $\mathcal{A}$ there exist paths $q_0 \xrightarrow{w} q_1$ and $q_0 \xrightarrow{w'} q_1'$ such that $q_1 \in \mathcal{C} \cap F$, $q_1' \in \mathcal{C}' \setminus F$ and $|w| = |w'|$.*

Then almost surely $L(\mathcal{A}^{\leftarrow p})$ is not a recursively enumerable language.

Proof. Let us denote $\ell = |w|$ and m (resp. m') the periodicity of $\mathcal{C}$ (resp. $\mathcal{C}'$).

Let $a \in \Sigma$. We build a Markov chain $\mathcal{M}$ from $\mathcal{C}$ as follows: every transition $q \xrightarrow{a} q'$ has probability $1-p$ and for all $b \neq a$, every transition $q \xrightarrow{b} q'$ has probability $\frac{p}{|\Sigma|-1}$. We proceed similarly from $\mathcal{C}'$ to build $\mathcal{M}'$.

Let us denote α_n (resp. α_n') the probability in $\mathcal{M}$ (resp. $\mathcal{M}'$) that starting from q_1 (resp. q_1'), the current state at time n is q_1 (resp. q_1'). Since $\mathcal{M}$ and $\mathcal{M}'$ are irreducible, $\lim_{n\to\infty} \alpha_{mn}$ (resp. $\lim_{n\to\infty} \alpha'_{m'n}$) exists and is positive. Let us denote α (resp. α') this limit. There exists n_0 such that for all $n \geq n_0$, $\alpha_{mn} \geq \frac{\alpha}{2}$ and $\alpha'_{m'n} \geq \frac{\alpha'}{2}$.

Define for all $n \in \mathcal{N}$, $w_n = wa^{mm'(n+n_0)}$.

The probability that w_n is accepted by $L(\mathcal{A})^{\leftarrow p}$ is lower bounded by the probability that the prefix w is unchanged (thus reaching q_1) and that after $mm'(n+n_0)$ steps the current state in $\mathcal{M}$ is q_1. So a lower bound is: $\min(p, 1-p)^\ell \frac{\alpha}{2}$.

The probability that w_n is rejected by $L(\mathcal{A})^{\leftarrow p}$ is lower bounded by the probability that the prefix w is changed into w' (thus reaching q_1') and that after $mm'(n+n_0)$ steps the current state in $\mathcal{M}'$ is q_1'. So a lower bound is: $\min(p, 1-p)^\ell \frac{\alpha'}{2}$.

Let $W \subseteq W_n$. The probability that $L^{\leftarrow p} \cap W_n$ is equal to W is upper bounded by:

$$\left(1 - \min(p, 1-p)^\ell \frac{\min(\alpha, \alpha')}{2}\right)^n$$

Thus $\rho_n \leq \left(1 - \min(p, 1-p)^\ell \frac{\min(\alpha,\alpha')}{2}\right)^n$ and $\lim_{n\to\infty} \rho_n = 0$. □

The DFA $\mathcal{A}$ of Figure 11.3 that represents the formula 'a Until b' of temporal logic LTL fulfills the hypotheses of Theorem 11.5.3. The corresponding pair of

states consists of the accepting state and the leftmost one. Checking the hypotheses of this theorem can be done in quadratic time by first building a graph whose set of vertices is $Q \times Q$ and there is an edge $(q_1, q_2) \to (q_1', q_2')$ if there are some transitions $q_1 \xrightarrow{a_1} q_1'$ and $q_2 \xrightarrow{a_2} q_2'$ and then looking for a vertex (q_1, q_2) in some BSCC with $q_1 \in F$ and $q_2 \notin F$ reachable from (q_0, q_0).

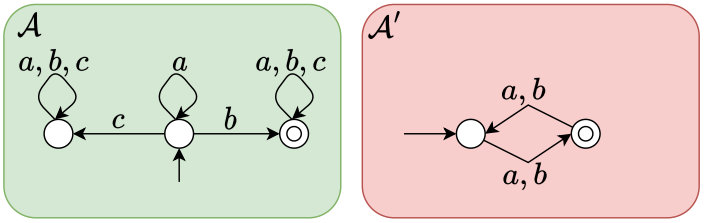


Figure 11.3: Two DFA

Range	#	$d(\mathcal{L}(\mathcal{A}), \mathcal{L}(\mathcal{A}^{\leftarrow p}))$	$d(\mathcal{L}(\mathcal{A}), \mathcal{L}(\mathcal{A}_E))$	$d(\mathcal{L}(\mathcal{A}^{\leftarrow p})), \mathcal{L}(\mathcal{A}_E))$	gain
[0.005,0.025]	85	0.01114	0.03604	0.04345	0.30902
[0.002,0.005]	81	0.00338	0.00421	0.00747	0.80443
[0.001,0.002]	25	0.00142	0.00035	0.00174	4.09784
[0.0005,0.001]	16	0.00071	0.00006	0.00077	11.08439

Table 11.6: Experiments on DFA fulfilling the hypotheses of Theorem 11.5.3

Range	#	$d(\mathcal{L}(\mathcal{A}), \mathcal{L}(\mathcal{A}^{\leftarrow p}))$	$d(\mathcal{L}(\mathcal{A}), \mathcal{L}(\mathcal{A}_E))$	$d(\mathcal{L}(\mathcal{A}^{\leftarrow p}), \mathcal{L}(\mathcal{A}_E))$	gain
[0.005,0.025]	36	0.01089	0.02598	0.03410	0.41905
[0.002,0.005]	49	0.00308	0.00387	0.00646	0.79628
[0.001,0.002]	35	0.00136	0.00057	0.00182	2.39863
[0.0005,0.001]	36	0.00075	0.00063	0.00130	1.18583

Table 11.7: Experiments on DFA not fulfilling the hypotheses of Theorem 11.5.3

In order to experimentally validate that the structural criterion is the main one for robustness of the Angluin's L* algorithm, we have refined our experiments on DFA with noisy inputs partitioning the randomly generated DFA depending on whether they fulfill the hypotheses of Theorem 11.5.3. We have chosen $|\Sigma| = 3$ since with greater size, it was difficult to generate DFA that do not satisfy the

hypotheses. Tables 11.7 and 11.6 summarize these experiments. Looking at the last rows of the tables (where the information gain is greater than one) confirms our conjecture.

Chapter 12

Visibly Pushdown Languages

12.1 Introduction

Context-free languages (CFLs), which are generated by context-free grammars (CFGs), abound in many application areas, for example when facing *formal* languages and applications such as programming languages and compilers, but especially also when processing natural language or controlled natural language. *Visibly pushdown languages* (VPLs), introduced by [7, 8], are a robust subclass of CFLs with interesting closure and decidability properties, as explained in further detail below—and are the class of languages studied in this chapter. The idea is that the underlying pushdown automata are *input-driven* [118,], i.e., every letter from the given alphabet is assigned a type among *push*, *pop*, and *internal* (we therefore deal with a *visibly pushdown alphabet*).

In this chapter, as a first contribution, we present a novel learning algorithm for VPLs, given a minimally adequate teacher.

An important application area of such a learning algorithm, as pursued in this chapter, is to derive so-called *surrogate models*, also known as approximation models, of recurrent neural networks (RNNs). RNNs play an important role in natural-language processing or time-series prediction, amongst others. While a neural network is often difficult to analyze and to understand, the surrogate model shares essential features of the underlying network but allows for simpler means for its analysis and explainability.

As a second contribution, we show that our algorithm can indeed be used for deriving a so-called visibly pushdown grammar (VPG) usable for explaining the language accepted by an underlying RNN. To this end, we perform queries to the network and infer an automaton model, which is then translated into a grammar. The latter provides structural information of the underlying network, which can hardly be obtained from the network directly.

Our Approach. As mentioned above, our learning algorithm is for the class of VPLs. Alur et al. [7] established a close relationship between VPLs and regular tree languages. We exploit this relationship and use an existing learning algorithm for regular tree languages [142, 48,] to derive a grammar-based representation of a VPL, resulting in a MAT learning algorithm.

This is similar to Sakakibara's algorithm [142,], which infers CFGs in terms of tree automata learned using *structural* queries. In our case, we also adopt tree interpretations of the words that are queried, but with not exactly the same structure.

In fact, [95] and [80] had already pointed out that it would be possible to use the algorithm of [142] for learning regular tree languages to obtain a tree representation of a VPL, albeit mentioning two potential obstacles for this. First, the final *visibly pushdown automaton* is non-deterministic, requiring thus the exponential cost in obtaining a deterministic one. Furthermore, certain structural properties cannot be guaranteed that are expected from recursive programs. Our work focusing on practical learning of VPLs shows that these critical issues can be well handled by adapting the improved version of [142,] by [48] and by leveraging the computational power of RNNs.

One advantage of our algorithm as opposed to other algorithms for classes of CFLs is that it is easier to understand, as it is based on the case for tree languages. Moreover, its correctness essentially follows from the correctness of the tree-learning algorithm so that, in principle, we can plug in any other tree-automata learning algorithm having the same interfaces. Another advantage is its extensibility to non-context-free languages, insofar as they have a representation as tree languages [111,].

Application to RNNs. Our work is inspired by [161], who infer CFGs from RNNs by extracting a sequence of deterministic finite automata (DFA) using the algorithm proposed by [156], and exploiting the notion of pattern rule sets (PRSs), from which the CFG rules are derived. Experiments show that many interesting CFLs can be learned. There are nevertheless some difficulties to overcome. For example, a sequence of extracted DFA often contains some noise, either from the RNN training or from the application of the L^* algorithm. Consequently, incorrect patterns are frequently inserted into the DFA sequence, which can deviate from the PRS. To handle this, a voting and threshold scheme has been proposed such that the languages of the given RNNs were mostly recovered in terms of CFGs, while several others were partially or incorrectly learned.

The class of VPLs is incomparable to the language class handled by [161] (cf. Example 12.2.2 in Section 12.2). It must be fairly noted that our algorithm relies on a partitioning of the input alphabet into *push*, *pop*, and *internal* symbols, which

is not required by [161]. However, it turns out that all the 15 benchmark languages considered by [161] are VPLs.

In [161], checking equivalence between the given RNN and a hypothesis grammar relies on an orthogonal learned abstraction of the RNN. In our case, the equivalence query relies on two complementary tests to look for words belonging to their symmetric difference, i.e., words in only one of the two corresponding languages.

Apart from two exceptions, the languages from [161] are very well learned with our approach, even some of the languages that are only partially generalized by applying the other approach. This demonstrates that our algorithm may be a worthwhile alternative when dealing with structured data (annotated linguistic data, programs, XML documents, etc.), i.e., in presence of a visibly pushdown alphabet.

Further Related Work. Some researchers adapted learning algorithms for regular languages to learn CFLs. For example, Clark et al. [36] presented an exact analogue of that proposed by [9] for a limited class of CFLs by combining the correspondence of non-terminals to the syntactic congruence class with weak substitutability. Then, Clark et al. [35] expanded this approach by adopting an extended MAT (minimally adequate teacher) to answer equivalence queries where the hypothesis may not be in the learnable class. Yoshinaka et al. [163] extended the syntactic congruence to tuples of strings to learn efficiently some sorts of multiple CFGs. Even though the above algorithms for learning CFLs have shown some promising results, they are limited to some constrained class. The learnability of the whole class of CFLs is widely believed to be intractable [44,].

Alur et al. showed in [6] that there is generally no unique minimal VPA for a given VPL. To this end they present a restricted VPA model they call k-module single entry visibly pushdown automata (k-SEVPA). This model has a unique minimal presentation for any VPL, where the number of states might be exponential as compared with a minimal VPA with this language. Using this representation Isberner [80] showed an algorithm learning VPL in terms of k-SEVPA.

Since decades, some approaches have been developed to extract simpler and explainable surrogate models from a neural network to facilitate comprehension and verification [148, 123,]. New algorithms for extracting (weighted or unweighted) DFA from RNNs have been proposed recently, with promising applications in verification [156, 115, 14, 131, 157, 114,]. They may also turn out to be useful for generalizations to other, more complex classes of languages. Up to now, however, there has been little research on extracting CFGs from RNNs. With the exception of [161,], existing approaches rely on an RNN augmented with external stack memory, either continuous or discrete [42, 147,]. In such a hybrid system, besides the classical input symbols, the input includes also what is read from the top of

the stack.

Organization. Section 12.2 recalls basic notions of VPLs. Trees and tree automata are presented in Section 12.3. In Section 12.4, we recall the tree-automata learning algorithm that we exploit, in Section 12.5, to learn grammars for VPLs. In Section 12.6, we apply our algorithm to inferring grammars from RNNs.

Based on. This chapter is base on our work in [17].

12.2 Visibly Pushdown Languages

Notation change. Note that, in all Part III we use λ to denote the empty word instead ε.
For the definition of context-free languages and grammar we direct the reader to Section 2.3.

The class of *visibly pushdown languages* has been introduced by [7, 8]. It was originally defined in terms of visibly pushdown automata, but can be equivalently characterized by a subclass of context-free grammars. visibly pushdown languages constitute a robust class that, unlike the class of context-free languages, is closed under complement.

The idea is to assign to every letter from an alphabet a precise role. Speaking in terms of automata, every letter is either a push, a pop, or an internal symbol. This clearly is a restriction: A pushdown automaton for the context-free language $\{a^n b a^n \mid n \in \mathcal{N}\}$ has to perform a certain number of push operations while reading the first n occurrences of a before the b, and pop operations when reading the remaining letters a. On the other hand, $\{a^n b^n \mid n \in \mathcal{N}\}$ can be recognized by a pushdown automaton where a stack symbol is pushed when reading an a and a stack symbol is popped when reading a b. Accordingly, a *visibly pushdown alphabet* is an alphabet $\Sigma = \Sigma_{\mathsf{push}} \uplus \Sigma_{\mathsf{pop}} \uplus \Sigma_{\mathsf{int}}$ that is partitioned into *push*, *pop*, and *internal letters*.

For the rest of the chapter, Σ will always denote a given visibly pushdown alphabet.

Definition 12.2.1. A *visibly pushdown grammar (VPG)* over Σ is a CFG $(V, S, \to)$ such that every rule has one of the following forms (where $A, B, C \in V$): $A \to \lambda$, or $A \to cB$ with $c \in \Sigma_{\mathsf{int}}$, or $A \to aBbC$ with $a \in \Sigma_{\mathsf{push}}$ and $b \in \Sigma_{\mathsf{pop}}$.

A language $\mathcal{L} \subseteq \Sigma^*$ is called a *visibly pushdown language (VPL)* over Σ if there is a VPG G over Σ such that $\mathcal{L}(G) = \mathcal{L}$.

Example 12.2.2. Let's recall Dyck languages from the Preliminaries, Section 2.3. For $n \geq 1$, consider the grammar G_n given by $S \to \lambda \mid p_i S q_i S$ (for all $i \in$

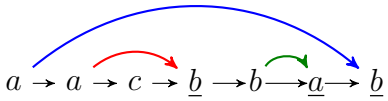


Figure 12.1: A nested word

$\{1, \ldots, n\}$) over the alphabet $\Sigma_n = \{p_1, \ldots, p_n, q_1, \ldots, q_n\}$. Hereby, as usual, $|$ separates several possible right-hand sides of rules. Then, $L(G_n)$ is the Dyck language of order n of well-bracketed words, where p_i is an opening and q_i its corresponding closing bracket. The CFG G_n is in fact also a VPG for $\Sigma_{\mathsf{push}} = \{p_1, \ldots, p_n\}$, $\Sigma_{\mathsf{pop}} = \{q_1, \ldots, q_n\}$, and $\Sigma_{\mathsf{int}} = \emptyset$ so that $\mathcal{L}(G_n)$ is a VPL. Another example of a VPL is $\{a^n x b^n \mid n \in \mathcal{N}\}$ where $\Sigma_{\mathsf{push}} = \{a\}$, $\Sigma_{\mathsf{pop}} = \{b\}$, and $\Sigma_{\mathsf{int}} = \{x\}$. This language is not captured by the PRS-formalism presented by [161] (cf. [162, Section C.3]).

We observe that, due to the form of permitted rules, a VPL $\mathcal{L}$ can only contain words $w \in \Sigma^*$ that are well-formed in a certain sense. The set $\mathcal{W}_\Sigma$ of *well-formed* words over Σ is defined as the language $\mathcal{L}(G_\Sigma)$ of the "most permissive" VPG: $G_\Sigma = (\{S\}, S, \to)$ with set of rules $\{S \to \lambda\} \cup \{S \to cS \mid c \in \Sigma_{\mathsf{int}}\} \cup \{S \to aSbS \mid a \in \Sigma_{\mathsf{push}} \text{ and } b \in \Sigma_{\mathsf{pop}}\}$.

The general framework by [7, 8] can also cope with words that have unmatched push or pop positions. For simplicity, we restrict here to well-formed words. However, the algorithms can be extended straightforwardly to the general case.

With $w = a_1 \ldots a_n \in \mathcal{W}_\Sigma$, we can associate a unique binary relation $\curvearrowright \subseteq \{1, \ldots, n\} \times \{1, \ldots, n\}$ connecting a push with a unique pop position: For $i, j \in \{1, \ldots, n\}$, we let $i \curvearrowright j$ if $i < j$, $a_i \in \Sigma_{\mathsf{push}}$, $a_j \in \Sigma_{\mathsf{pop}}$, and $a_{i+1} \ldots a_{j-1}$ is well-formed. We call the pair $(w, \curvearrowright)$ (with $w \in \mathcal{W}_\Sigma$) a *nested word*. A nested word over Σ with $\Sigma_{\mathsf{push}} = \{a, b\}$, $\Sigma_{\mathsf{pop}} = \{\underline{a}, \underline{b}\}$, and $\Sigma_{\mathsf{int}} = \{c\}$ is depicted in Figure 12.1. We do not exploit nested words in this chapter, but it is helpful to think of well-formed words as nested words when we encode them as trees.

12.2.1 Visibly Pushdown Automata

Though we are principally interested in inferring grammars, we give here the definition of visibly pushdown automata, which also constitute a characterization of the class of VPLs.

Definition 12.2.3. A *visibly pushdown automaton (VPA)* over Σ is a tuple $\mathcal{A} = (Q, \mathcal{S}, \delta, \iota, F)$ containing a finite set of control states Q, a nonempty finite set of stack symbols $\mathcal{S}$, an initial state ι, and a set of final states $F \subseteq Q$. Moreover, $\delta =$

$(\delta_{\mathsf{push}}, \delta_{\mathsf{pop}}, \delta_{\mathsf{int}})$ is a collection of transition functions $\delta_{\mathsf{push}} : Q \times \Sigma_{\mathsf{push}} \to \mathcal{P}(Q \times \mathcal{S})$, $\delta_{\mathsf{pop}} : Q \times \Sigma_{\mathsf{pop}} \times \mathcal{S} \to \mathcal{P}(Q)$, and $\delta_{\mathsf{int}} : Q \times \Sigma_{\mathsf{int}} \to \mathcal{P}(Q)$. We call $\mathcal{A}$ *deterministic* if all transition functions map all arguments to singleton sets.

A VPA recognizes a language $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^*$. Intuitively, it is the language of an infinite automaton whose states (we actually say configurations) are pairs (q, σ) where $q \in Q$ is the current control state and $\sigma \in \mathcal{S}^*$ is the current stack contents. With this, in the infinite automaton, we have a transition $(q, \sigma) \xrightarrow{a} (q', \sigma')$ if there is $A \in \mathcal{S}$ such that one of the following holds:

- $a \in \Sigma_{\mathsf{push}}$ and $(q', A) \in \delta_{\mathsf{push}}(q, a)$ and $\sigma' = \sigma \cdot A$
- $a \in \Sigma_{\mathsf{pop}}$ and $q' \in \delta_{\mathsf{pop}}(q, a, A)$ and $\sigma = \sigma' \cdot A$
- $a \in \Sigma_{\mathsf{int}}$ and $q' \in \delta_{\mathsf{int}}(q, a)$ and $\sigma' = \sigma$

We call (q, σ) a final configuration if $q \in F$ and $\sigma = \lambda$. Moreover, (ι, λ) is the only initial configuration. Finally, we define $\mathcal{L}(\mathcal{A})$ to be the language recognized by this infinite automaton in the expected way.

Fact 1 ([7, 8]). Let $\mathcal{L} \subseteq \Sigma^*$. Then, $\mathcal{L}$ is a VPL over Σ if and only if there is a VPA $\mathcal{A}$ over Σ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}$.

12.3 Trees and Regular Tree Languages

The reason why VPLs are so robust is that they are close to tree languages. In fact, nested words as introduced in the previous section can be represented as trees. Trees are defined over a *ranked alphabet*, i.e., an alphabet $\Gamma = \Gamma_0 \uplus \Gamma_1 \uplus \ldots \uplus \Gamma_{k_{max}}$ that is partitioned into letters of arity $k \in \{0, \ldots, k_{max}\}$ where $k_{max} \in \mathcal{N}$ is the maximal arity. Unless otherwise stated, we let Γ be a fixed ranked alphabet.

A tree t over Γ is a term that is generated according to the grammar $t ::= a(t_1, \ldots, t_k)$, where k ranges over $\{0, \ldots, k_{max}\}$ and a over Γ_k. Figure 12.2 depicts a syntax-tree-based representation of the tree

$$a(a(c(\Box()), \underline{b}(b(\Box(), \underline{a}(\Box())))), \underline{b}(\Box()))$$

over the ranked alphabet given by $\Gamma_0 = \{\Box\}$, $\Gamma_1 = \{\underline{a}, \underline{b}, c\}$, and $\Gamma_2 = \{a, b\}$. The size $|t|$ of t is the number of its nodes, i.e., the number of occurrences of symbols from Γ. Let $\mathsf{Trees}(\Gamma)$ denote the set of all trees over Γ.

The algorithm by [48], on which our approach is based, infers regular tree languages in terms of tree automata (later, when a tree automaton represents a VPL, we will be able to extract a corresponding VPG representation).

Definition 12.3.1. A *nondeterministic finite (bottom-up) tree automaton (NTA)* over Γ is a tuple $\mathcal{B} = (Q, \delta, F)$ where Q is the nonempty finite set of states, $F \subseteq Q$

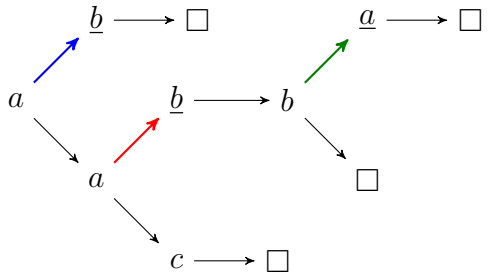


Figure 12.2: A tree

is the set of final states, and $\delta : \bigcup_{k\in\{0,\ldots,k_{max}\}}(\Gamma_k \times Q^k) \to \mathcal{P}(Q)$ is the transition function. We will write $\delta(a(q_1,\ldots,q_k))$ instead of $\delta(a,q_1,\ldots,q_k)$.

We call $\mathcal{B}$ *deterministic* (a DTA) if $|\delta(a(q_1,\ldots,q_k))| = 1$ for all arguments $a, q_1, \ldots, q_k$. Then, δ can also be seen as a total (i.e., complete) function $\delta : \bigcup_{k\in\{0,\ldots,k_{max}\}}(\Gamma_k \times Q^k) \to Q$. We let $\mathsf{DTA}(\Gamma)$ denote the set of DTAs over Γ.

From $\delta : \bigcup_{k\in\{0,\ldots,k_{max}\}}(\Gamma_k \times Q^k) \to \mathcal{P}(Q)$, we obtain a function $\hat{\delta} : \mathsf{Trees}(\Gamma) \to \mathcal{P}(Q)$ letting, for an arity $k \in \{0,\ldots,k_{max}\}$, $a \in \Gamma_k$, and $t_1,\ldots,t_k \in \mathsf{Trees}(\Gamma)$, $\hat{\delta}(a(t_1,\ldots,t_k)) = \bigcup_{q_1\in\hat{\delta}(t_1),\ldots,q_k\in\hat{\delta}(t_k)} \delta(a(q_1,\ldots,q_k))$. We can now define the tree language recognized by $\mathcal{B}$ as $T(\mathcal{B}) = \{t \in \mathsf{Trees}(\Gamma) \mid \hat{\delta}(t) \cap F \neq \emptyset\}$. We call a tree language $T \subseteq \mathsf{Trees}(\Gamma)$ *regular* if it is recognized by some NTA over Γ.

We now state some important and well-known facts about tree automata. For more details, we refer the reader to [38,].

Fact 2 (minimal DTA). For every NTA $\mathcal{B} = (Q, \delta, F)$, there is a unique (up to isomorphism) minimal DTA $\mathcal{B}' = (Q', \delta', F')$ such that $T(\mathcal{B}') = T(\mathcal{B})$. We can assume $|Q'| \leq 2^{|Q|}$.

The *index* of a regular tree language T is the number of states of the minimal DTA recognizing T.

While DTAs capture the class of regular tree languages, deterministic *top-down* finite tree automata [38,], which we do not define here, are strictly less expressive.

Fact 3 (membership and emptiness).

1. Given an NTA $\mathcal{B}$ and a tree $t \in \mathsf{Trees}(\Gamma)$, one can decide in polynomial time whether $t \in T(\mathcal{B})$. For DTAs, there is a linear-time algorithm.
2. For a given NTA $\mathcal{B}$, one can decide in polynomial time whether $T(\mathcal{B}) \neq \emptyset$.

12.4 Learning Deterministic Tree Automata

Recall, Angluin's L* algorithm, an algorithm inferring a deterministic finite automaton for a given regular word language that can only be accessed via two

types of queries: *membership queries (MQs)* and *equivalence queries (EQs)*. In [48] Angluin's algorithm is extended to tree automata, this algorithm is called TL*. TL*, can infer a DTA over a fixed ranked alphabet Γ for a given (unknown) regular tree language T. Hereby, T can be accessed through membership queries and equivalence queries, which are implemented by "oracle" mappings $\mathsf{MQ}_{\text{tree}} : \mathsf{Trees}(\Gamma) \to \{\mathsf{yes}, \mathsf{no}\}$ and $\mathsf{EQ}_{\text{tree}} : \mathsf{DTA}(\Gamma) \to \{\mathsf{yes}\} \cup \mathsf{Trees}(\Gamma)$:

- We say that $\mathsf{MQ}_{\text{tree}}$ is *sound* for T if, for all $t \in \mathsf{Trees}(\Gamma)$, $\mathsf{MQ}_{\text{tree}}(t) = \mathsf{yes}$ if and only if $t \in T$.
- We say that $\mathsf{EQ}_{\text{tree}}$ is *counterexample-sound* for T if, for all $\mathcal{B} \in \mathsf{DTA}(\Gamma)$ and $t \in \mathsf{Trees}(\Gamma)$ such that $\mathsf{EQ}_{\text{tree}}(\mathcal{B}) = t$, we have $t \in T \oplus T(\mathcal{B})$ (i.e., t is a *counterexample*).
- We call $\mathsf{EQ}_{\text{tree}}$ *equivalence-sound* for T if, for all $\mathcal{B} \in \mathsf{DTA}(\Gamma)$ such that $\mathsf{EQ}_{\text{tree}}(\mathcal{B}) = \mathsf{yes}$, we have $T = T(\mathcal{B})$.

These queries act as "oracles" and their answers are delivered instantaneously. Ideally, one assumes that $\mathsf{EQ}_{\text{tree}}$, which checks the current *hypothesis* computed by the learning algorithm, is both counterexample- and equivalence-sound. In practice, this is not always the case. In fact, in our experiments, we will make weaker assumptions on $\mathsf{EQ}_{\text{tree}}$.

The algorithm TL* by [48] takes as input a ranked alphabet Γ and two functions $\mathsf{MQ}_{\text{tree}} : \mathsf{Trees}(\Gamma) \to \{\mathsf{yes}, \mathsf{no}\}$ and $\mathsf{EQ}_{\text{tree}} : \mathsf{DTA}(\Gamma) \to \{\mathsf{yes}\} \cup \mathsf{Trees}(\Gamma)$. If TL$^*(\Gamma, \mathsf{MQ}_{\text{tree}}, \mathsf{EQ}_{\text{tree}})$ terminates, it outputs a DTA over Γ.

Fact 4 ([48]). Let $T \subseteq \mathsf{Trees}(\Gamma)$ be a regular tree language, say with index n (the minimal DTA for T has n states). Suppose $\mathsf{MQ}_{\text{tree}}$ is sound for T and that $\mathsf{EQ}_{\text{tree}}$ is both counterexample- and equivalence-sound for T. Then, TL$^*(\Gamma, \mathsf{MQ}_{\text{tree}}, \mathsf{EQ}_{\text{tree}})$ terminates and outputs the unique minimal DTA $\mathcal{B}$ with n states such that $T(\mathcal{B}) = T$. The overall running time is polynomial in $|\Gamma|$, $n^{k_{max}}$, and the maximal size of a counterexample returned by $\mathsf{EQ}_{\text{tree}}$.

Note that, in the next sections, k_{max} will be fixed. However, unlike in the case of word automata, the size of a smallest counterexample tree returned for an equivalence query may be exponential in the size of the target automaton.

12.5 Learning Visibly Pushdown Grammars

In this section, we exploit tree-automata learning for the inference of VPLs in terms of VPGs. The derived algorithm will then be exploited to extract grammars from RNNs.

12.5.1 Encoding Nested Words as Trees

The main link between words and trees is provided by an encoding of well-formed words as trees over a suitable ranked alphabet [7, 8,].

Let $\Sigma = \Sigma_{\mathsf{push}} \uplus \Sigma_{\mathsf{pop}} \uplus \Sigma_{\mathsf{int}}$ be a visibly pushdown alphabet. To encode words from $\mathcal{W}_\Sigma$ as trees, we introduce a suitable ranked alphabet $\Gamma = \Gamma_0 \uplus \Gamma_1 \uplus \Gamma_2$ letting $\Gamma_0 = \{\Box\}$, $\Gamma_1 = \Sigma_{\mathsf{pop}} \cup \Sigma_{\mathsf{int}}$, and $\Gamma_2 = \Sigma_{\mathsf{push}}$. That is, the maximal arity is 2. For the rest of this section, we fix Σ and the associated ranked alphabet Γ.

To a well-formed word $w \in \mathcal{W}_\Sigma$, we inductively assign a (parse) tree $\langle\!\langle w \rangle\!\rangle \in \mathsf{Trees}(\Gamma)$ as follows:

1. $\langle\!\langle \lambda \rangle\!\rangle = \Box()$.
2. If $w = aw_1bw_2$ such that $a \in \Sigma_{\mathsf{push}}$, $b \in \Sigma_{\mathsf{pop}}$, and w_1 and w_2 are well-formed, then $\langle\!\langle w \rangle\!\rangle = a(\langle\!\langle w_1 \rangle\!\rangle, b(\langle\!\langle w_2 \rangle\!\rangle))$.
3. If $c \in \Sigma_{\mathsf{int}}$ and w is well-formed, then $\langle\!\langle cw \rangle\!\rangle = c(\langle\!\langle w \rangle\!\rangle)$.

One can a nested word and its encoding in Figure 12.3.

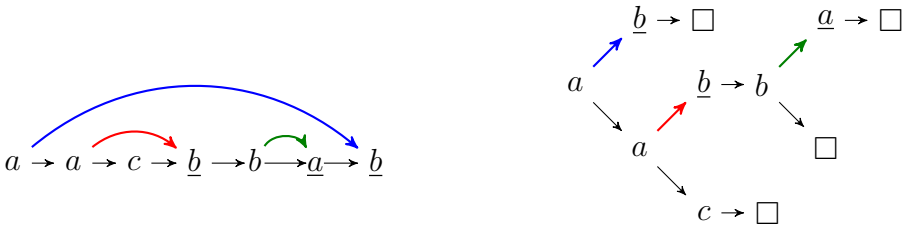


Figure 12.3: Nested word and its tree encoding

Given $\mathcal{L} \subseteq \mathcal{W}_\Sigma$, we let $\langle\!\langle \mathcal{L} \rangle\!\rangle = \{\langle\!\langle w \rangle\!\rangle \mid w \in \mathcal{L}\} \subseteq \mathsf{Trees}(\Gamma)$. Moreover, we let $\mathcal{T}_\Gamma = \langle\!\langle \mathcal{W}_\Sigma \rangle\!\rangle$ be the set of trees that encode a well-formed word. Note that $\langle\!\langle . \rangle\!\rangle : \mathcal{W}_\Sigma \to \mathcal{T}_\Gamma$ is injective and, therefore, a bijection. Indeed, its inverse mapping, which we denote by $[\![.]\!]$, is given by $[\![\Box()]\!] = \lambda$, $[\![a(t_1, b(t_2))]\!] = a[\![t_1]\!]b[\![t_2]\!]$ and $[\![c(t)]\!] = c[\![t]\!]$. For $T \subseteq \mathcal{T}_\Gamma$, let $[\![T]\!] = \{[\![t]\!] \mid t \in T\}$.

Let us state some known facts on the relation between VPGs and NTAs/DTAs due to [7, 8].

Fact 5. For every VPL $\mathcal{L}$ over Σ, there is an NTA (or DTA) $\mathcal{B}$ over Γ such that $T(\mathcal{B}) = \langle\!\langle \mathcal{L} \rangle\!\rangle$. In particular, there is a DTA $\mathcal{B}_{\mathsf{parse}}$ over Γ with a constant number of states such that $T(\mathcal{B}_{\mathsf{parse}}) = \mathcal{T}_\Gamma$.

As we will extract grammars from tree automata, the following is particularly important:

Fact 6. Let $\mathcal{B}$ be an NTA over Γ such that $T(\mathcal{B}) \subseteq \mathcal{T}_\Gamma$. One can compute, in polynomial time, a VPG nta2vpg($\mathcal{B}$) over Σ such that $\mathcal{L}(\text{nta2vpg}(\mathcal{B})) = [\![T(\mathcal{B})]\!]$.

Algorithm 13: Implementing $\mathsf{MQ}_{\text{tree}}$ in terms of MQ_{vpl}

```
1 MQ_tree(t):
2     if t ∈ T(B_parse)
3         then return MQ_vpl([[t]])
4     else return no
```

Algorithm 14: Implementing $\mathsf{EQ}_{\text{tree}}$ in terms of EQ_{vpl}

```
1 EQ_tree(B):
2     if T(B) ⊆ T(B_parse)
3         then return EQ_vpl(B)
4     else
5         pick t ∈ T(B) \ T(B_parse)
6         return t
```

We give the translation of an NTA into a VPG, as the latter will yield the representation of a VPL learned in terms of the NTA. Suppose $\mathcal{B} = (Q, \delta, F)$ is an NTA over Γ such that $T(\mathcal{B}) \subseteq \mathcal{T}_\Gamma$. We define $\text{nta2vpg}(\mathcal{B}) = (V, \mathcal{I}, \to)$ as follows. In fact, instead of just one start symbol, we assume a set of start symbols $\mathcal{I} \subseteq V$. This is no more expressive than having one single start symbol, as we can always introduce a fresh start symbol, leading to all the right-hand sides of rules associated with symbols from $\mathcal{I}$. Intuitively, the grammar derives a run of the NTA top-down, where states are successively replaced with input letters. So we let $V = Q$ and $\mathcal{I} = F$. Moreover, the set of rules contains

1. $\hat{q} \to \lambda$ for all $\hat{q} \in \delta(\Box())$;
2. $\hat{q} \to cq$ for all $c \in \Sigma_{\mathsf{int}}$, $q \in Q$, and $\hat{q} \in \delta(c(q))$;
3. $\hat{q} \to apbq$ for all $a \in \Sigma_{\mathsf{push}}$, $b \in \Sigma_{\mathsf{pop}}$, and $p, q, q', \hat{q} \in Q$ such that $q' \in \delta(b(q))$ and $\hat{q} \in \delta(a(p, q'))$.

For completeness, let us mention some connections with visibly pushdown automata (VPAs), which are effectively equivalent to VPGs w.r.t. expressive power so that we could also learn VPAs instead of VPGs (cf. [7, 8,] for the definition of VPAs). For an NTA $\mathcal{B}$ over Γ such that $T(\mathcal{B}) \subseteq \mathcal{T}_\Gamma$, one can compute, in polynomial time, a VPA $\mathcal{A}$ over Σ such that $\mathcal{L}(\mathcal{A}) = [\![T(\mathcal{B})]\!]$. Conversely, for a VPA $\mathcal{A}$ over Σ, one can compute, in polynomial time, an NTA $\mathcal{B}$ over Γ such that $T(\mathcal{B}) = \langle\!\langle L(\mathcal{A}) \rangle\!\rangle$. Hence, there is also a DTA for $\langle\!\langle \mathcal{L}(\mathcal{A}) \rangle\!\rangle$ of exponential size. In general, this exponential blow-up cannot be avoided even when we start from a deterministic VPA.

12.5.2 Learning VPLs in Terms of VPGs

Recall that Σ is a fixed visibly pushdown alphabet and Γ is the derived ranked alphabet.

We now present an algorithm, called VPL* in the following, that learns a VPL $\mathcal{L} \subseteq \mathcal{W}_\Sigma$ in terms of a DTA for the tree language $\langle\langle \mathcal{L} \rangle\rangle \subseteq \mathcal{T}_\Gamma$ that can then be translated into a VPG according to Fact 6. In particular, the equivalence query will take a DTA as argument, rather than a VPA. Essentially, we rely on the algorithm TL*. However, equivalence and membership queries are now answered w.r.t. the VPL $\mathcal{L}$. More precisely, we deal with a mapping $\mathsf{MQ}_{\mathrm{vpl}} : \mathcal{W}_\Sigma \to \{\mathsf{yes}, \mathsf{no}\}$ and a partial mapping $\mathsf{EQ}_{\mathrm{vpl}} : \mathsf{DTA}(\Gamma) \to \{\mathsf{yes}\} \cup \mathcal{T}_\Gamma$ whose domain is the set of DTAs $\mathcal{B} \in \mathsf{DTA}(\Gamma)$ such that $T(\mathcal{B}) \subseteq \mathcal{T}_\Gamma$:

- We call $\mathsf{MQ}_{\mathrm{vpl}}$ *sound* for $\mathcal{L}$ if, for all $w \in \mathcal{W}_\Sigma$, we have $\mathsf{MQ}_{\mathrm{vpl}}(w) = \mathsf{yes}$ if and only if $w \in \mathcal{L}$.
- We say that $\mathsf{EQ}_{\mathrm{vpl}}$ is *counterexample-sound* for $\mathcal{L}$ if, for all $\mathcal{B} \in \mathsf{DTA}(\Gamma)$ such that $T(\mathcal{B}) \subseteq \mathcal{T}_\Gamma$ and all $t \in \mathcal{T}_\Gamma$, $\mathsf{EQ}_{\mathrm{vpl}}(\mathcal{B}) = t$ implies $[\![t]\!] \in \mathcal{L} \oplus [\![T(\mathcal{B})]\!]$.
- We say that $\mathsf{EQ}_{\mathrm{vpl}}$ is *equivalence-sound* for $\mathcal{L}$ if, for all $\mathcal{B}$ over Γ such that $T(\mathcal{B}) \subseteq \mathcal{T}_\Gamma$, $\mathsf{EQ}_{\mathrm{vpl}}(\mathcal{B}) = \mathsf{yes}$ implies $\mathcal{L} = [\![T(\mathcal{B})]\!]$.

Our algorithm VPL* for learning VPLs uses TL* as a black-box. Therefore, we define a mapping $\mathsf{MQ}_{\mathrm{tree}} : \mathsf{Trees}(\Gamma) \to \{\mathsf{yes}, \mathsf{no}\}$ and a mapping $\mathsf{EQ}_{\mathrm{tree}} : \mathsf{DTA}(\Gamma) \to \{\mathsf{yes}\} \cup \mathsf{Trees}(\Gamma)$ that implement the membership and equivalence queries for tree languages, respectively (cf. Algorithms 13 and 14). The algorithm VPL* (Algorithm 15) then simply calls TL* with parameters $(\Gamma, \mathsf{MQ}_{\mathrm{tree}}, \mathsf{EQ}_{\mathrm{tree}})$ and translates the resulting DTA into a VPG.

Algorithm 13. Membership query $\mathsf{MQ}_{\mathrm{tree}}(t)$ with $t \in T(\mathcal{B}_{\mathsf{parse}}) = \mathcal{T}_\Gamma$ is answered in terms of $\mathsf{MQ}_{\mathrm{vpl}}([\![t]\!])$ (line 3). If, on the other hand, $t \notin T(\mathcal{B}_{\mathsf{parse}})$, the query returns no (line 4).

Algorithm 14. Recall that we are looking for a tree automaton for the language $T = \langle\langle \mathcal{L} \rangle\rangle$, which is included in $T(\mathcal{B}_{\mathsf{parse}})$. We will, therefore, first check whether this inclusion also applies to the current hypothesis DTA $\mathcal{B}$, i.e., whether $T(\mathcal{B}) \subseteq T(\mathcal{B}_{\mathsf{parse}})$. If not, then we can find a tree $t \in T(\mathcal{B}) \setminus T(\mathcal{B}_{\mathsf{parse}})$, which serves as a counterexample to the equivalence query (line 5). So suppose that $T(\mathcal{B}) \subseteq T(\mathcal{B}_{\mathsf{parse}})$. Let us assume that $\mathsf{EQ}_{\mathrm{vpl}}$ is both counterexample- and equivalence-sound. If it returns a tree $t = \mathsf{EQ}_{\mathrm{vpl}}(\mathcal{B})$, then $[\![t]\!] \in \mathcal{L} \oplus [\![T(\mathcal{B})]\!]$ so that t can indeed be used to refine the hypothesis $\mathcal{B}$. If, on the other hand, $\mathsf{EQ}_{\mathrm{vpl}}(\mathcal{B}) = \mathsf{yes}$, then $\mathcal{L} = [\![T(\mathcal{B})]\!]$, i.e., $T(\mathcal{B}) = \langle\langle \mathcal{L} \rangle\rangle$, so that we can return $\mathcal{B}$ as a suitable tree-language representation. Algorithm 15 then returns the VPG $G = \mathrm{nta2vpg}(\mathcal{B})$. According to Fact 6, we have $\mathcal{L}(G) = [\![T(\mathcal{B})]\!] = \mathcal{L}$.

Theorem 12.5.1. *Let $\mathcal{L}$ be a VPL and $\hat{\mathcal{B}}$ be the minimal DTA such that $T(\hat{\mathcal{B}}) = \langle\langle \mathcal{L} \rangle\rangle$. Assume $\mathsf{MQ}_{\mathrm{vpl}}$ is sound for $\mathcal{L}$ and that $\mathsf{EQ}_{\mathrm{vpl}}$ is both counterexample- and equivalence-sound for $\mathcal{L}$. Then,* VPL* *(Algorithm 15) terminates and eventually*

Algorithm 15: VPL*

1 $\mathcal{B} \leftarrow \mathrm{TL}^*(\Gamma, \mathsf{MQ}_{\mathrm{tree}}, \mathsf{EQ}_{\mathrm{tree}})$ /* $\mathsf{MQ}_{\mathrm{tree}}$ and $\mathsf{EQ}_{\mathrm{tree}}$ from Algorithms 13 and 14 */

2 **return** nta2vpg($\mathcal{B}$)

returns a VPG G of size polynomial in the size of $\hat{\mathcal{B}}$ such that $\mathcal{L}(G) = \mathcal{L}$. The overall running time is polynomial in $|\Sigma|$, the index of $\hat{\mathcal{B}}$, and the maximal size of a counterexample returned in lines 3 and 6 of Algorithm 14.

Proof. By Fact 6, we have to show that calling $\mathrm{TL}^*(\Gamma, \mathsf{MQ}_{\mathrm{tree}}, \mathsf{EQ}_{\mathrm{tree}})$ returns, in polynomial time, a DTA $\mathcal{B}$ such that $T(\mathcal{B}) = T(\hat{\mathcal{B}})$. We will show that $\mathsf{MQ}_{\mathrm{tree}}$ is sound for $T(\hat{\mathcal{B}})$ and $\mathsf{EQ}_{\mathrm{tree}}$ is counterexample- and equivalence-sound for $T(\hat{\mathcal{B}})$. By Fact 4, this implies that $\mathrm{TL}^*(\Gamma, \mathsf{MQ}_{\mathrm{tree}}, \mathsf{EQ}_{\mathrm{tree}})$ returns a DTA $\mathcal{B}$ such that $T(\mathcal{B}) = T(\hat{\mathcal{B}})$. The running time is polynomial since all additional operations in Algorithms 13 and 14 and can be performed in polynomial time (cf. Fact 3).

To show that $\mathsf{MQ}_{\mathrm{tree}}$ is sound for $T(\hat{\mathcal{B}})$, let $t \in \mathsf{Trees}(\Gamma)$. Assume $\mathsf{MQ}_{\mathrm{tree}}(t) = \mathsf{yes}$. By Algorithm 13, this implies $t \in T(\mathcal{B}_{\mathsf{parse}})$ and $\mathsf{MQ}_{\mathrm{vpl}}(\llbracket t \rrbracket) = \mathsf{yes}$. As $\mathsf{MQ}_{\mathrm{vpl}}$ is sound for $\mathcal{L}$, we have $\llbracket t \rrbracket \in \mathcal{L}$. Since $T(\hat{\mathcal{B}}) = \langle\langle \mathcal{L} \rangle\rangle$, we get $t \in T(\hat{\mathcal{B}})$. Conversely, assume $\mathsf{MQ}_{\mathrm{tree}}(t) = \mathsf{no}$. If $t \notin T(\mathcal{B}_{\mathsf{parse}})$, then $t \notin T(\hat{\mathcal{B}})$. So suppose $t \in T(\mathcal{B}_{\mathsf{parse}})$ and $\mathsf{MQ}_{\mathrm{vpl}}(\llbracket t \rrbracket) = \mathsf{no}$. As $\mathsf{MQ}_{\mathrm{vpl}}$ is sound for $\mathcal{L}$, we have $\llbracket t \rrbracket \notin \mathcal{L}$, which implies $t \notin T(\hat{\mathcal{B}})$.

Let us show that $\mathsf{EQ}_{\mathrm{tree}}$ is counterexample-sound for $T(\hat{\mathcal{B}})$. Suppose $\mathcal{B} \in \mathsf{DTA}(\Gamma)$ and $t \in \mathsf{Trees}(\Gamma)$ such that $\mathsf{EQ}_{\mathrm{tree}}(\mathcal{B}) = t$. There are two cases. First, suppose $t \in T(\mathcal{B}) \setminus T(\mathcal{B}_{\mathsf{parse}})$. As $T(\hat{\mathcal{B}}) \subseteq T(\mathcal{B}_{\mathsf{parse}})$, we have $t \in T(\mathcal{B}) \setminus T(\hat{\mathcal{B}})$ and, hence, $t \in T(\mathcal{B}) \oplus T(\hat{\mathcal{B}})$. Second, assume $T(\mathcal{B}) \subseteq T(\mathcal{B}_{\mathsf{parse}})$. As $\mathsf{EQ}_{\mathrm{vpl}}$ is counterexample-sound for $\mathcal{L}$, this implies $\llbracket t \rrbracket \in \mathcal{L} \oplus \llbracket T(\mathcal{B}) \rrbracket$. Due to $T(\hat{\mathcal{B}}) = \langle\langle \mathcal{L} \rangle\rangle$, we get $t \in T(\hat{\mathcal{B}}) \oplus T(\mathcal{B})$.

Finally, we show that $\mathsf{EQ}_{\mathrm{tree}}$ is equivalence-sound for $T(\hat{\mathcal{B}})$. Suppose $\mathcal{B} \in \mathsf{DTA}(\Gamma)$ such that $\mathsf{EQ}_{\mathrm{tree}}(\mathcal{B}) = \mathsf{yes}$. Then, $T(\mathcal{B}) \subseteq T(\mathcal{B}_{\mathsf{parse}})$ and $\mathsf{EQ}_{\mathrm{vpl}}(\mathcal{B}) = \mathsf{yes}$. As $\mathsf{EQ}_{\mathrm{vpl}}$ is equivalence-sound for $\mathcal{L}$, we get $\mathcal{L} = \llbracket T(\mathcal{B}) \rrbracket$, which implies $T(\hat{\mathcal{B}}) = T(\mathcal{B})$. □

Note that, like in TL*, a smallest counterexample tree returned for an equivalence query may be of exponential size. Also note that the size of the returned VPG G is at most exponential in the size of a minimal (nondeterministic) VPA recognizing $\mathcal{L}$.

12.6 Experiments

We applied Algorithm 15 to recurrent neural networks (RNNs) in order to extract VPGs. We implemented it in Python 3.6, using the NumPy library[1]. Since we are comparing our extractions to those done in [161,][2], we use the 15 RNNs they trained, and a modified version of the interface they wrote to communicate with these RNNs. All benchmarks were performed on a computer equipped by Intel i5-8250U CPU with 4 cores, 16 GB of memory, and Ubuntu Linux 18.03.

The experiments done for this chapter are **very preliminary**, yet they show promise and motivate a more in-depth investigation. We describe some ideas for some more in-depth experimentation in the end of Chapter 13

Recurrent Neural Networks. RNNs can be seen as language acceptors. For the purpose of this chapter, it is enough to think of an RNN $\mathcal{R}$ as an infinite automaton with infinite state space Q (e.g., $Q = \mathbb{R}^{dim}$ for some dimension $dim \geq 1$), initial state $q_0 \in Q$, transition function $\delta : Q \times \Sigma \to Q$, and a mapping $score : Q \to \mathbb{R}$ (e.g., indicating a probability of acceptance or of an "end of sequence" token). As usual, δ is extended to $\hat{\delta} : Q \times \Sigma^* \to Q$ over words by applying δ letter by letter. Then, $\mathcal{R}$ computes a (score) function $\mathcal{R} : \Sigma^* \to \mathbb{R}$ by $\mathcal{R}(w) = score(\hat{\delta}(q_0, w))$. Moreover, given a threshold $\tau \in \mathbb{R}$, one can associate with R a language letting $\mathcal{L}(\mathcal{R}) = \{w \in \Sigma^* \mid \mathcal{R}(w) \geq \tau\}$ or using different threshold criteria.

In fact, [161] use *language-model RNNs* where, in addition, every letter gets a dedicated score in a given state. They define the semantics $\mathcal{L}(R)$ as the *locally τ-truncated support* where "acceptance" is subject to the condition that the score of every letter (including "end of sequence") has a score greater than τ (cf. [74,]).

Several well-known architectures are available to effectively represent RNNs, such as LSTM [76,], and GRUs [33,]. Generally, depending on the architecture, the expressive power of RNNs goes beyond the regular languages. So it is worthwhile to study extraction methods for classes of CFLs.

Methodology and Results. The 15 CFLs considered by [161] are given in Table 12.2, together with the CFGs from Table 12.1. For conciseness, they are defined in terms of general CFGs. However, it turns out that all of them are VPLs. In most cases, there is arguably a canonical partition of the alphabet into a visibly pushdown alphabet. For all these VPLs, we considered the RNNs provided by [161], which were trained on sample sets generated by a probabilistic version of a corresponding CFG.

In our experiments, we used a Kearns Vazirani variation of TL* (cf. [49]). A query $\mathsf{MQ}_{\text{vpl}}(w)$ for a well-formed word w was answered according to the given

[1] The code is available here: `https://github.com/LeaRNNify/VPA_learning`

[2] The RNN can be found here: `https://github.com/tech-srl/RNN_to_PRS_CFG`

Table 12.1: Definition of some CFLs (X and Y are finite sets of words)

$\mathcal{L}(X,Y)$:	*RE-Dyck*(X,Y):	$Dyck_1 \triangleleft X$:
$S \to \lambda \mid xSy$	$S \to xAy$	$S \to p_1 A q_1$
(for all $x \in X$ and $y \in Y$)	$A \to xAy \mid AA \mid \lambda$	$A \to p_1 A q_1 \mid AA \mid \lambda \mid x$
	(for all $x \in X$ and $y \in Y$)	(for all $x \in X$)
$Dyck_n$:	*Alternating*:	$Dyck_2 \triangleleft X$:
$S \to p_i A q_i$	$S \to A \mid B$	$S \to p_i A q_i$
$A \to p_i A q_i \mid AA \mid \lambda$	$A \to (B) \mid \lambda$	$A \to p_i A q_i \mid AA \mid \lambda \mid x$
(for all $i \in \{1, \ldots, n\}$)	$B \to [A] \mid \lambda$	(for all $i \in \{1,2\}$ and $x \in X$)

RNN $\mathcal{R}$, i.e., $\mathsf{MQ}_{\text{vpl}}(w) = \mathsf{yes}$ if and only if $w \in \mathcal{L}(\mathcal{R})$. To answer a query $\mathsf{EQ}_{\text{vpl}}(\mathcal{B})$, we used two independent subroutines that look for counterexample words (of length under 30):

(i) We chose 1500 random words.

(ii) We noticed that the 15 languages represented by the RNNs from [161,] are very sparse (similarly to a lot of other examples of RNN languages from the literature). Therefore, taking only random samples from the RNN would most likely produce an empty language. In order to avoid this, we implemented a type of A* exploration (cf. [140,]) in the rooted directed tree of all words Σ^*, where each vertex is a word $w \in \Sigma^*$ and its children are wa for $a \in \Sigma$. This word-exploration technique relies on an evaluation function $f : \Sigma^* \to \mathbb{R}$ where the higher the score of a word the higher its priority to be explored first. The function we chose for this depended on two things: 1) The average score given to the word by the RNN and its neighborhood (where the assumption is that the higher the score the closer we are to a word in the RNN language), and 2) The length of the word, where we preferred shorter words. The function we chose is

$$f(w) = \frac{1}{|w|^2} \sum_{w' \in \Sigma^* \text{ s.t. } |w'| \leq d} R(ww')$$

where $R(\cdot)$ is the score given to the word by the RNN, and the size of the neighborhood was chosen to be $d = 4$. Using this type of exploration, we generated a set P (performed once in the beginning of the run) of positive

examples from the RNN language (only *well-formed words*; timeout of 60 seconds).

Note that EQ_{vpl} is counterexample-sound for $\mathcal{L}(\mathcal{R})$ but not necessarily equivalence-sound. It was sufficiently precise on our small set of examples, which leads us to believe that there is a reasonable chance that a further investigation might reveal that it is well performing in a more general environment. Though the given trained RNNs have imperfections, the intended languages are learned in most cases. Table 12.2 indicates the time needed to learn a VPG, averaging across five runs, the number of rules extracted, and the final column showing whether [161] were able to extract the correct language. In most runs, the extracted VPGs are equivalent to the respective CFGs the RNNs were trained on. Exceptions are $\mathcal{L}_{14}$ and $\mathcal{L}_{15}$ for which we obtain grammars approximating the respective languages. This happens due to structural errors in the given RNNs w.r.t. the target languages. Note that we extracted 13/15 languages compared to 10/15 extracted by [161].

Table 12.2: Results for learning RNNs

		Visibly Pushdown Alphabet					
	Language	Push	Pop	Int	#Rules	Time	Extracted by [161]
$\mathcal{L}_1$	$\mathcal{L}(\{a\},\{b\})$	$\{a\}$	$\{b\}$		3	1s	Yes
$\mathcal{L}_2$	$\mathcal{L}(\{a,b\},\{c,d\})$	$\{a,b\}$	$\{c,d\}$		9	23s	Yes
$\mathcal{L}_3$	$\mathcal{L}(\{ab,cd\},\{ef,gh\})$	$\{a,b,c,d\}$	$\{e,f,g,h\}$		13	74s	No
$\mathcal{L}_4$	$\mathcal{L}(\{ab\},\{cd\})$	$\{a,b\}$	$\{c,d\}$		4	1s	Yes
$\mathcal{L}_5$	$\mathcal{L}(\{abc\},\{def\})$	$\{a,b,c\}$	$\{d,e,f\}$		5	1s	Yes
$\mathcal{L}_6$	$\mathcal{L}(\{ab,c\},\{de,f\})$	$\{a,c\}$	$\{d,f\}$	$\{b,e\}$	10	49s	No
$\mathcal{L}_7$	$Dyck_2$	$\{p_1,p_2\}$	$\{q_1,q_2\}$		19	69s	Yes
$\mathcal{L}_8$	$Dyck_3$	$\{p_1,p_2,p_3\}$	$\{q_1,q_2,q_3\}$		28	74s	No
$\mathcal{L}_9$	$Dyck_4$	$\{p_1,\ldots,p_4\}$	$\{q_1,\ldots,q_4\}$		37	79s	Yes
$\mathcal{L}_{10}$	$RE\text{-}Dyck(\{(abcd\},\{wxyz)\})$	$\{(,a,b,c,d\}$	$\{w,x,y,z,)\}$		10	7s	Yes
$\mathcal{L}_{11}$	$RE\text{-}Dyck(\{ab,c\},\{de,f\})$	$\{a,c\}$	$\{d,f\}$	$\{b,e\}$	27	59s	No
$\mathcal{L}_{12}$	*Alternating*	$\{(,[\}$	$\{),]\}$		5	2s	Yes
$\mathcal{L}_{13}$	$Dyck_1 \triangleleft \{a,b,c\}$	$\{p_1\}$	$\{q_1\}$	$\{a,b,c\}$	19	66s	Yes
$\mathcal{L}_{14}$	$Dyck_2 \triangleleft \{a,b,c\}$	$\{p_1,p_2\}$	$\{q_1,q_2\}$	$\{a,b,c\}$	–	65s	Yes
$\mathcal{L}_{15}$	$Dyck_1 \triangleleft \{abc,d\}$	$\{p_1\}$	$\{q_1\}$	$\{a,b,c,d\}$	–	51s	No

To give a (successful) example, Table 12.3 depicts the grammar that was output for $\mathcal{L}_{10}$.

Table 12.3: Learned VPG for $\mathcal{L}_{10}$ with start symbol $\mathtt{A}_1$

$\mathtt{A}_1 \to (\, \mathtt{A}_2 \,)\, \mathtt{A}_0$	$\mathtt{A}_2 \to a\, \mathtt{A}_3\, z\, \mathtt{A}_0$	$\mathtt{A}_4 \to c\, \mathtt{A}_5\, x\, \mathtt{A}_0$	$\mathtt{A}_5 \to d\, \mathtt{A}_1\, w\, \mathtt{A}_0$	$\mathtt{A}_6 \to (\, \mathtt{A}_2 \,)\, \mathtt{A}_1$
$\mathtt{A}_0 \to \lambda$	$\mathtt{A}_3 \to b\, \mathtt{A}_4\, y\, \mathtt{A}_0$	$\mathtt{A}_5 \to d\, \mathtt{A}_0\, w\, \mathtt{A}_0$	$\mathtt{A}_5 \to d\, \mathtt{A}_6\, w\, \mathtt{A}_0$	$\mathtt{A}_6 \to (\, \mathtt{A}_2 \,)\, \mathtt{A}_6$

Fixing Mistakes Variation. The previous result can easily be ruined by a wrong sample of words. For example, we could pick a word that is in the RNN language but not in the original language. To mitigate this problem, one can do the following: Denote by P the set of positive examples generated from the RNN, let H be the current hypothesis grammar, and let $pos(H) = \frac{|P \cap \mathcal{L}(H)|}{|P|}$. Assume that H comes with a counterexample w_c and a new hypothesis H'. If $pos(H') < pos(H)$, then we keep refining both of them, but making sure that w_c cannot be used as a counterexample for H. In the end, we return the hypothesis which is "closest" to the RNN language, i.e., the one with largest $pos(H)$. For example, by increasing the sampling length from the RNN ($30 \to 40$) and the size of the sample set ($1500 \to 2000$), we ruined (most of the RNN have some errors in them, it is just a matter of time to find them) the extraction of language $\mathcal{L}_8$, but using the procedure above, we manage to fix this issue.

Agnostic Learning. Some criticism might be given to the fact that we assume the visibly pushdown alphabet to be known. To solve this issue, we generated a set P of positive words from the RNN like before. Using P, we examined all the possible visibly pushdown alphabets (there may be several), picked the best suited alphabet (with the least number of internal symbols), and continued learning. We succeeded in 8 of the 13 languages that were successful in the non-agnostic case.

Chapter 13

Conclusion

Cover of Petri nets

The study of the Karp and Miller algorithm led us to several results. First, in Chapter 3, we have commodified the concept of accelerations from the Karp and Miller algorithm, using new notions of abstraction, acceleration, and exploration sequence. Using these notions, we have developed a simple and elegant proof of the Karp and Miller algorithm. Then we designed an accelerated version of this algorithm which memorizes all the accelerations already computed in order to re-apply them systematically.

Furthermore, in Chapter 4, using our new notions, we designed a simple and efficient modification of the incomplete minimal coverability tree algorithm for building the Clover of a net. Compared to the alternative algorithms previously designed, we have theoretically bounded the size of the additional space (2-EXPSPACE).

In Chapter 5, we have implemented a prototype, `MinCov`, for this algorithm, and showed the optimization used in the implementation. We have compared the performance of our algorithm with other existing algorithms generating the Clover on benchmarks from the literature and on random benchmarks generated by us. On all the benchmarks `MinCov` beats the competitions in memory size, and in total time (including timeouts). Finally, we showed how one can combine `MinCov` with ideas from `qCover` in order to create a superior tool solving coverability fast.

In the future, plan to study the possibility of effectively pre-calculating the set of minimal accelerations or some relevant subset. Finally, it would be interesting to introduce and apply the concept of acceleration for the study of other well-structured transition systems.

Recursive Petri nets

In chapter 8, we begin by defining a quasi-order on states of RPN compatible with the firing rule and establish that it is not a well quasi-order. Using this quasi-order, we have shown that the family of RPN coverability languages strictly includes both the family of Petri net coverability languages and context-free languages. Moreover, we showed that this family is not far from the recursive enumerable languages. Finally, from an algorithmic point of view we showed that complexity of coverability, termination, boundedness and finiteness problems are EXPSPACE-complete.

In Chapter 9, we have introduced DRPN that extends RPNs in several directions. We have showed that this model extends not only RPN but also many other Petri net extensions such as transfer-nets, rest-nets, affine-nets... We have shown that the family of DRPN coverability languages strictly includes the family of RPN coverability languages, and we have established that the coverability problem is still decidable.

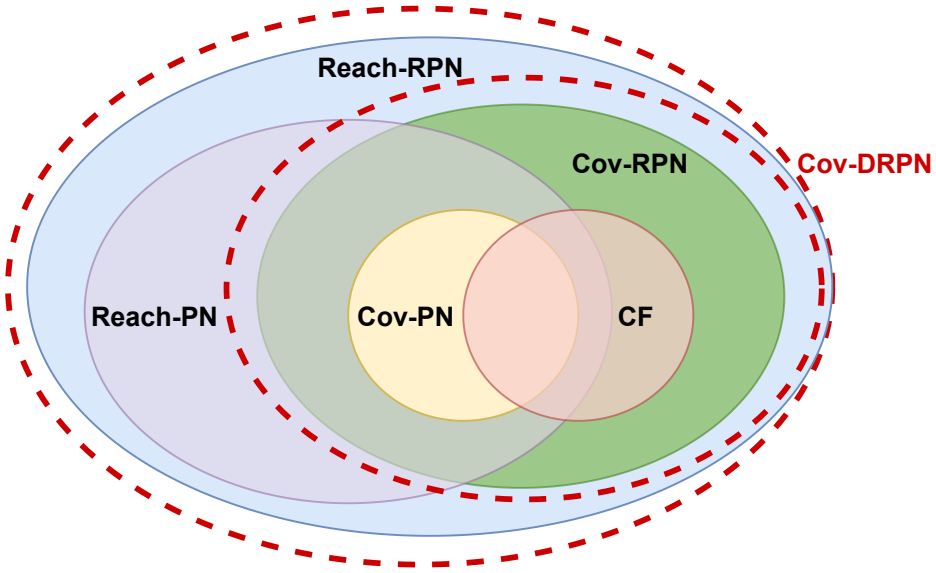


Figure 13.1: Language families defined by RPN and DRPN, and their relations to other languages.

In the future we would like to study the following problems:

- Where is the family of DRPN coverability languages compared to the reachability languages? See Figure 13.1 for the two possible position of the DRPN coverability languages,

- How to decide whether a word belongs to a coverability or reachability language of an RPN/DRPN?
- Since the quasi-order possesses an infinite antichain, but there exist short witnesses for coverability, does there exist an effective finite representation of the downward closure of the reachability set?
- Does there exist a restriction of DRPN for which it still strictly extends RPN, but for which reachability would be still decidable?
- Can we adapt our ideas from `MinCov` to have an efficient tool solving coverability for RPN/DRPN?

Active Learning and Verification

In chapter 10, we proposed property-directed verification as a new verification method for formally verifying RNNs with respect to regular specifications, with adversarial robustness certification as one important application. We have experimentally compared PDV to two other methods, on randomly generated DFA, instances of adversarial robustness certification and on contact sequences. We saw that PDV is the best approach time wise, which also gives very short counter examples if it finds a mistake. Finally, we showed that if PDV finds a mistake, then the DFA generated by the PDV can be used to find "faulty flows" helping the user to generate more counter examples, and even more importantly may help them to understand the mistake.

In the future we would like to extend our ideas to the setting of Moore/Mealy machines supporting the setting of richer classes of RNN classifiers. Another future work is to investigate the applicability of our approach for RNNs representing more expressive languages, such as context-free ones. Finally, we plan to extend the PDV algorithm for the formal verification of RNN-based agent environment systems, and to compare it with the existing results.

In Chapter 11 we have studied how the PAC Angluin's algorithm behaves for devices which are obtained from a DFA by introducing some noise. More precisely, we studied whether Angluin's algorithm reduces the noise producing a DFA closer to the original one than the noisy device. We have considered two kinds of noises: random noise and structured noise. We have shown that on average Angluin's algorithm behaves well for random noise and not for structured noise. We have completed our study by establishing that almost surely the random noisy devices produce a non recursively enumerable language, confirming the relevance of the structural criterion for robustness of Angluin's algorithm.

In the future we want to first develop an more sophisticated stopping mechanism. Currently, we are using a static one, using a maximal number. We believe it is possible to have one which depends dynamically on trackable information during its

run, e.g. its distance from the noise language. Note that Angluin's algorithm uses no extra information about the original DFA. It would be interesting to introduce a priori knowledge and design more efficient algorithms accordingly. For instance, the algorithm could take as input the maximal size of the original DFA or a regular language that is a superset of the original language.

In Chapter 12, we presented an algorithm to learn VPLs in the MAT framework. As an application, we focused on the extraction of grammars from RNNs, where we introduce a A^* type of technique of extracting positive and negative examples from an RNN. Our experiments suggest that the algorithm is a suitable alternative to current approaches when we deal with structured data.

In the future there are several directions we would like to investigate. The first one is as mentioned in the begin of Section 12.6 there is a strong need for further experimentation and evaluation of this algorithm. Here is a list of ideas towards these goals:

• Larger pool of RNNs representing different visibly pushdown languages.
• The number of queries taken while checking equality was chosen to be very small. This was done in order to reproduce the exact CFLs, since L* type of algorithms are inherently sensitive to errors. Even one error can produce a completely different VPG. If the goal had been to produce a VPG whose language is statistically close to the original one, then one could have taken a more probabilistic approach using, for example, the Chernoff-Hoeffding bound as done in chapters 10 and 11.
• Testing this technique on RNNs that represent a language which is not necessarily a CFL (e.g., the Amazon sentiment analysis, which was previously unlearnable with this type of method due to the sparseness of the language).

Moreover, currently in order to learn VPA we learn DTA which we translate into VPA, but we think that there is a better more direct way of doing that. One possible strategy is defining a minimal VPA which agrees with some type of Nerode congruence. Finally, we can use this algorithm in PDV, enlarging the verification settings we developed in Chapter 10.

Bibliography

[1] P. A. Abdulla, K. Cerans, B. Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *IEEE SLCS'96*, page 313, 1996. doi:10.1109/LICS.1996.561359.

[2] Tilak Agerwala and Mike Flynn. Comments on capabilities, limitations and "correctness" of Petri nets. *SIGARCH Comput. Archit. News*, 2(4):81–86, 1973. doi:10.1145/800123.803973.

[3] Gul Agha, Fiorella de Cindio, and Grzegorz Rozenberg, editors. *Concurrent Object-Oriented Programming and Petri Nets, Advances in Petri Nets*, volume 2001 of *LNCS*, 2001. doi:10.1007/3-540-45397-0.

[4] Michael E. Akintunde, Andreea Kevorchian, Alessio Lomuscio, and Edoardo Pirovano. Verification of RNN-based neural agent-environment systems. In *AAAI'13*, pages 6006–6013, 2019. doi:10.1609/aaai.v33i01.33016006.

[5] Artiom Alhazov, Sergiu Ivanov, Elisabeth Pelz, and Sergey Verlan. Small universal deterministic Petri nets with inhibitor arcs. *J. Autom. Lang. Comb.*, 21(1-2):7–26, 2016. doi:10.1007/978-3-319-09704-6_17.

[6] Rajeev Alur, Viraj Kumar, P. Madhusudan, and Mahesh Viswanathan. Congruences for visibly pushdown languages. In *Automata, Languages and Programming*, pages 1102–1114, 2005. doi:10.1007/11523468_89.

[7] Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In *STOC'04*, pages 202–211, 2004. doi:10.1145/1007352.1007390.

[8] Rajeev Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3):16:1–16:43, 2009. doi:10.1145/1516512.1516518.

[9] Dana Angluin. Inference of reversible languages. *J. ACM*, 29(3):741–765, 1982. doi:10.1145/322326.322334.

[10] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987. doi:10.1016/0890-5401(87)90052-6.

[11] Dana Angluin and Philip D. Laird. Learning from noisy examples. *Mach. Learn.*, 2(4):343–370, 1987. doi:10.1007/bf00116829.

[12] Toshiro Araki and Tadao Kasami. Some decision problems related to the reachability problem for Petri nets. *Theoretical Computer Science*, 3(1):85 – 104, 1976. doi:10.1016/0304-3975(76)90067-0.

[13] Mohamed Faouzi Atig and Pierre Ganty. Approximating Petri net reachability along context-free traces. In *FSTTCS'13*, volume 13 of *LIPIcs*, pages 152–163, 2011. doi:10.4230/LIPIcs.FSTTCS.2011.152.

[14] Stéphane Ayache, Rémi Eyraud, and Noé Goudian. Explaining black boxes on sequential data using weighted automata. In *ICGI'18*, volume 93, pages 81–103, 2018. URL: https://proceedings.mlr.press/v93/ayache19a.html.

[15] Eric Badouel, Luca Bernardinello, and Philippe Darondeau. *Petri Net Synthesis*. Texts in Theoretical Computer Science. An EATCS Series. 2015. doi:10.1007/978-3-662-47967-4.

[16] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[17] Benoît Barbot, Benedikt Bollig, Alain Finkel, Serge Haddad, Igor Khmelnitsky, Martin Leucker, Daniel Neider, Rajarshi Roy, and Lina Ye. Extracting context-free grammars from recurrent neural networks using tree-automata learning and A* search. In *ICGI'21*, volume 153 of *Proceedings of Machine Learning Research*, pages 113–129, 2021. URL: https://proceedings.mlr.press/v153/barbot21a.html.

[18] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and Software Verification: Model-Checking Techniques and Tools*. 2010.

[19] Olivier Bernardi and Omer Giménez. A linear algorithm for the random sampling from regular languages. *Algorithmica*, 62(1-2):130–145, 2012. doi:10.1007/s00453-010-9446-5.

[20] Bernard Berthomieu and Michel Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Trans. Software Eng.*, 17(3):259–273, 1991. doi:10.1109/32.75415.

[21] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, C-21(6):592–597, 1972. doi:10.1109/tc.1972.5009015.

[22] Alan W. Biermann and Jerome A. Feldman. A survey of results in grammatical inference. In *Frontiers of Pattern Recognition*, pages 31–54. 1972. doi:10.1016/b978-0-12-737140-5.50007-5.

[23] J. Billington. *Extensions to colored Petri nets and their applications*. PhD thesis, University of Cambridge, 1991.

[24] Michel Blockelet and Sylvain Schmitz. Model checking coverability graphs of vector addition systems. In *MFCS'11*, pages 108–119, 2011. doi:10.1007/978-3-642-22993-0_13.

[25] M. Blondin, A. Finkel, C. Haase, and S. Haddad. The logical view on continuous Petri nets. *ACM Trans. Comput. Log.*, 18(3):24:1–24:28, 2017. doi:10.1145/3105908.

[26] Michael Blondin, Alain Finkel, Christoph Haase, and Serge Haddad. Approaching the coverability problem continuously. In *TACAS'16*, volume 9636 of *LNCS*, pages 480–496, 2016. doi:10.1007/978-3-662-49674-9_28.

[27] Michael Blondin, Alain Finkel, and Pierre McKenzie. Well behaved transition systems. *LMCS*, 13(3):1–19, 2017. doi:10.23638/LMCS-13(3:24)2017.

[28] Rémi Bonnet. The reachability problem for vector addition system with one zero-test. In *MFCS'11*, volume 6907 of *LNCS*, pages 145–157, 2011. doi:10.1007/978-3-642-22993-0_16.

[29] Rémi Bonnet, Alain Finkel, Jérôme Leroux, and Marc Zeitoun. Model checking vector addition systems with one zero-test. *LMCS*, 8(2:11), 2012. doi:10.2168/lmcs-8(2:11)2012.

[30] Rémi Bonnet, Alain Finkel, and M. Praveen. Extending the Rackoff technique to affine nets. In *FSTTCS'12*, volume 18 of *LIPIcs*, 2012. doi:10.4230/LIPIcs.FSTTCS.2012.301.

[31] Christos G. Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. 2010. doi:10.1007/978-3-030-72274-6.

[32] Giovanni Chiola, Claude Dutheillet, Giuliana Franceschinis, and Serge Haddad. A symbolic reachability graph for coloured Petri nets. *Theoretical Computer Science*, 176(1-2):39–65, 1997. doi:10.1016/s0304-3975(96)00010-2.

[33] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning

phrase representations using RNN encoder-decoder for statistical machine translation. In *EMNLP'14*, pages 1724–1734. ACL, 2014. doi:10.3115/v1/d14-1179.

[34] Gianfranco Ciardo. Petri nets with marking-dependent arc cardinality: Properties and analysis. In *Petri Nets 1994*, pages 179–198, 1994. doi:10.1007/3-540-58152-9_11.

[35] Alexander Clark. Distributional learning of some context-free languages with a minimally adequate teacher. In *ICGI 2010*, volume 6339 of *LNCS*, pages 24–37, 2010. doi:10.1007/978-3-642-15488-1_4.

[36] Alexander Clark and Rémi Eyraud. Polynomial identification in the limit of substitutable context-free languages. *J. Mach. Learn. Res.*, 8:1725–1745, 2007. doi:10.1007/11564089_23.

[37] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV'00*, volume 1855 of *LNCS*, pages 154–169, 2000. doi:10.1007/10722167_15.

[38] H. Comon, M. Dauchet, F. Jacquemard, R. Gilleron, C. Löding, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata, 2007. release October, 12th 2007.

[39] J. M. Couvreur, S. Haddad, and J. F. Peyre. Generative families of positive invariants in coloured nets sub-classes. In *Petri Nets'93*, pages 51–70, 1993. doi:10.1007/3-540-56689-9_39.

[40] Wojciech Czerwinski, Slawomir Lasota, Ranko Lazic, Jérôme Leroux, and Filip Mazowiecki. The reachability problem for Petri nets is not elementary. In *STOC 19*, pages 24–33, 2019. doi:10.1145/3313276.3316369.

[41] Wojciech Czerwinski and Lukasz Orlikowski. Reachability in vector addition systems is ackermann-complete. *FOCS*, 2021. URL: https://arxiv.org/abs/2104.13866.

[42] Sreerupa Das, C. Lee Giles, and Guo-Zheng Sun. Using prior knowledge in a {NNPDA} to learn context-free languages. In *NIPS'92*, pages 65–72, 1992. URL: https://proceedings.neurips.cc/paper/1992/file/766ebcd59621e305170616ba3d3dac32-Paper.pdf.

[43] Jürgen Dassow and Sherzod Turaev. Petri net controlled grammars: the case of special Petri nets. *J. UCS*, 15(14):2808–2835, 2009.

[44] Colin de la Higuera. A bibliographical study of grammatical inference. *Pattern Recognit.*, 38(9):1332–1348, 2005. doi:10.1016/j.patcog.2005.01.003.

[45] Stéphane Demri. On selective unboundedness of VASS. *Journal of Computer and System Sciences*, 79(5):689–713, 2013. doi:10.1016/j.jcss.2013.01.014.

[46] Stéphane Demri, Marcin Jurdziński, Oded Lachish, and Ranko Lazić. The covering and boundedness problems for branching vector addition systems. *Journal of Computer and System Sciences*, 79(1):23–38, 2012. doi:10.1016/j.jcss.2012.04.002.

[47] Emanuele D'Osualdo, Jonathan Kochems, and C. H. Luke Ong. Automatic verification of Erlang-style concurrency. In *Static Analysis*, pages 454–476, 2013. doi:10.1007/978-3-642-38856-9_24.

[48] Frank Drewes and Johanna Högberg. Query learning of regular tree languages: How to avoid dead states. *Theory Comput. Syst.*, 40(2):163–185, 2007. doi:10.1007/s00224-005-1233-3.

[49] Frank Drewes, Johanna Högberg, and Andreas Maletti. MAT learners for tree series: an abstract data type and two realizations. *Acta Informatica*, 48(3):165–189, 2011. doi:10.1007/s00236-011-0135-x.

[50] Xiaoning Du, Yi Li, Xiaofei Xie, Lei Ma, Yang Liu, and Jianjun Zhao. Marble: Model-based robustness analysis of stateful deep learning systems. In *ASE'20*, pages 423–435, 2020. doi:10.1145/3324884.3416564.

[51] Catherine Dufourd, Alain Finkel, and Philippe Schnoebelen. Reset nets between decidability and undecidability. In *ICALP'98*, volume 1443 of *LNCS*, pages 103–115, Aalborg, Denmark, July 1998. doi:10.1007/bfb0055044.

[52] Amal El Fallah Seghrouchni and Serge Haddad. A recursive model for distributed planning. In *ICMAS'96*, pages 307–314, 1996.

[53] Yizhak Yisrael Elboher, Justin Gottschlich, and Guy Katz. An abstraction-based framework for neural network verification. In *CAV'20*, volume 12224 of *LNCS*, pages 43–65, 2020. doi:10.1007/978-3-030-53288-8_3.

[54] Javier Esparza, Ruslán Ledesma-Garza, Rupak Majumdar, Philipp Meyer, and Filip Niksic. An SMT-based approach to coverability analysis. In *CAV'14*, pages 603–619, 2014. doi:10.1007/978-3-319-08867-9_40.

[55] A. Finkel, P. McKenzie, and C. Picaronny. A well-structured framework for analysing Petri net extensions. *Information and Computation*, 195, 2004. doi:10.1016/j.ic.2004.01.005.

[56] Alain Finkel. The minimal coverability graph for Petri nets. In *Advances in Petri Nets 1993*, volume 674 of *LNCS*, pages 210–243, 1993. doi:10.1007/3-540-56689-9_45.

[57] Alain Finkel, Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. A counter-example the the minimal coverability tree algorithm. Technical Report 535, Université Libre de Bruxelles, Belgium, 2005. URL: http://www.lsv.fr/Publis/PAPERS/PDF/FGRV-ulb05.pdf.

[58] Alain Finkel and Jean Goubault-Larrecq. Forward analysis for WSTS, part II: Complete WSTS. *LMCS*, 8(4), 2012. doi:10.2168/lmcs-8(3:28)2012.

[59] Alain Finkel, Serge Haddad, and Igor Khmelnitsky. Coverability and termination in recursive petri nets. In *Petri nets'19*, volume 11522 of *LNCSs*, pages 429–448, Aachen, Germany, June 2019. doi:10.1007/978-3-030-21571-2_23.

[60] Alain Finkel, Serge Haddad, and Igor Khmelnitsky. Minimal coverability tree construction made complete and efficient. In *FSSCS'20*, volume 12077 of *LNCS*, pages 237–256, 2020. doi:10.1007/978-3-030-45231-5_13.

[61] Alain Finkel, Serge Haddad, and Igor Khmelnitsky. Commodification of accelerations for the Karp and Miller construction. *J-DEDS*, 31(2):251–270, 2021. doi:10.1007/s10626-020-00331-z.

[62] Alain Finkel, Serge Haddad, and Igor Khmelnitsky. Coverability, termination, and finiteness in recursive Petri nets. *Fundamenta Informaticae*, 183:33–66, 2021. doi:10.3233/fi-2021-2081.

[63] Alain Finkel and Philippe Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001. doi:10.1016/s0304-3975(00)00102-x.

[64] Gilles Geeraerts, Alexander Heußner, Manjunatha Praveen, and Jean-François Raskin. ω-Petri nets: algorithms and complexity. *Fundamenta Informaticae*, 137(1):29–60, 2015. doi:10.3233/FI-2015-1169.

[65] Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. On the efficient computation of the minimal coverability set for Petri nets. In *ATVA'07*, pages 98–113, 2007. doi:10.1007/978-3-540-75596-8_9.

[66] Giuseppe De Giacomo and Moshe Y. Vardi. Synthesis for LTL and LDL on finite traces. In *IJCAI'15*, pages 1558–1564, 2015.

[67] E Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302 – 320, 1978. doi:10.1016/s0019-9958(78)90562-4.

[68] Michel Hack. *Decidability questions for Petri Nets.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1976. URL: http://hdl.handle.net/1721.1/27441.

[69] Serge Haddad and Igor Khmelnitsky. Dynamic recursive Petri nets. In *Petri NETS'20*, volume 12152 of *LNCS*, pages 345–366, 2020. doi:10.1007/978-3-030-51831-8_17.

[70] Serge Haddad and Denis Poitrenaud. Decidability and undecidability results for recursive Petri nets. Technical Report 019, LIP6, Paris VI University, 1999.

[71] Serge Haddad and Denis Poitrenaud. Theoretical aspects of recursive Petri nets. In *ICATPN '99*, volume 1639 of *LNCS*, pages 228–247, 1999. doi:10.1007/3-540-48745-x_14.

[72] Serge Haddad and Denis Poitrenaud. Modelling and Analyzing Systems with Recursive Petri Nets. In *WODES '00*, volume 569, pages 449–458, 2000. doi:10.1007/978-1-4615-4493-7_48.

[73] Serge Haddad and Denis Poitrenaud. Checking linear temporal formulas on sequential recursive Petri nets. In *TIME'01*, pages 198–205, 2001. doi:10.1109/TIME.2001.930718.

[74] John Hewitt, Michael Hahn, Surya Ganguli, Percy Liang, and Christopher D. Manning. RNNs can generate bounded hierarchical languages with optimal memory. In *EMNLP'20*, pages 1978–2010, 2020. doi:10.18653/v1/2020.emnlp-main.156.

[75] G. Higman. Ordering by divisibility in abstract algebras. *Proc. London Math. Soc.*, 7(3):326–336, 1952. doi:10.1112/plms/s3-2.1.326.

[76] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, 1997. doi:10.1162/neco.1997.9.8.1735.

[77] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963. doi:10.1007/978-1-4612-0865-5_26.

[78] Petter Holme. Temporal networks. In *Encyclopedia of Social Network Analysis and Mining*, pages 2119–2129. 2014. doi:10.1007/978-1-4939-7131-2_42.

[79] M. Ilyas and H. Khalil. Modeling of communication protocols by using Petri nets. *Computers and Industrial Engineering*, 11(1):547–551, 1986. doi:10.1016/0360-8352(86)90151-8.

[80] Malte Isberner. *Foundations of active automata learning: an algorithmic perspective*. PhD thesis, Technical University Dortmund, Germany, 2015.

[81] Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In *Runtime Verification*, pages 307–322, 2014. doi:10.1007/978-3-319-11164-3_26.

[82] Yuval Jacoby, Clark W. Barrett, and Guy Katz. Verifying recurrent neural networks using invariant inference. *CoRR*, abs/2004.02462, 2020. doi:10.1007/978-3-030-59152-6_3.

[83] Kurt Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use - Volume 1*. EATCS Monographs on Theoretical Computer Science. 1992.

[84] Alexander Kaiser, Daniel Kroening, and Thomas Wahl. A widening approach to multithreaded program verification. *ACM Trans. Program. Lang. Syst.*, 36(4), 2014. doi:10.1145/2629608.

[85] Richard M. Karp and Raymond E. Miller. Parallel program schemata. *JCSS*, 3(2):147–195, 1969. doi:10.1016/s0022-0000(69)80011-5.

[86] Krishna M. Kavi, Alireza Moshtaghi, and Deng-Jyi Chen. Modeling multithreaded applications using petri nets. *Int. J. Parallel Program.*, 30(5):353–371, October 2002. doi:10.1023/A:1019917329895.

[87] Michael J. Kearns. Efficient noise-tolerant learning from statistical queries. *J. ACM*, 45(6):983–1006, 1998. doi:10.1145/293347.293351.

[88] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994. doi:10.7551/mitpress/3897.001.0001.

[89] C.W. Keck. *Principles of Public Health Practice*. 2002.

[90] Igor Khmelnitsky, Daniel Neider, Rajarshi Roy, Xuan Xie, Benoît Barbot, Benedikt Bollig, Alain Finkel, Serge Haddad, Martin Leucker, and Lina Ye. Property-directed verification and robustness certification of recurrent neural networks. In *ATVA'21*, 2021. To appear. doi:10.1007/978-3-030-88885-5_24.

[91] Astrid Kiehn. Petri net systems and their closure properties. In *Petri nets'89*, pages 306–328, 1990. doi:10.1007/3-540-52494-0_35.

[92] Michael Kishinevsky, Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, Alexander Taubin, and Alex Yakovlev. Coupling asynchrony and interrupts: Place chart nets. In *Petri nets'97*, pages 328–347, 1997. doi:10.1007/3-540-63139-9_44.

[93] Johannes Kloos, Rupak Majumdar, Filip Niksic, and Ruzica Piskac. Incremental, inductive coverability. In *CAV'13*, pages 158–173, 2013. doi:10.1007/978-3-642-39799-8_10.

[94] S. Kosaraju. Limitations of Dijkstra's semaphore primitives and petri nets. Technical report, Johns Hopkins University, 1973. doi:10.1145/800009.808062.

[95] Viraj Kumar, P. Madhusudan, and Mahesh Viswanathan. Minimization, learning, and conformance testing of boolean programs. In *CONCUR'06*, volume 4137 of *LNCS*, pages 203–217, 2006. doi:10.1007/11817949_14.

[96] Marta Z. Kwiatkowska. Safety verification for deep neural networks with provable guarantees. In *CONCUR'19*, volume 140 of *LIPIcs*, pages 1:1–1:5, 2019.

[97] Charles Lakos and Søren Christensen. A general systematic approach to arc extensions for coloured Petri nets. In *Petri Nets'94*, pages 338–357, 1994. doi:10.1007/3-540-58152-9_19.

[98] Ranko Lazic. The reachability problem for vector addition systems with a stack is not elementary. *CoRR*, abs/1310.1767, 2013. arXiv:1310.1767.

[99] Ranko Lazić, Tom Newcomb, Joël Ouaknine, A. W. Roscoe, and James Worrell. Nets with tokens which carry data. *Fundamenta Informaticae*, 88(3):251–274, 2008.

[100] Ranko Lazic and Sylvain Schmitz. Non-elementary complexities for branching VASS, MELL, and extensions. In *CSL-LICS '14*, pages 61:1–61:10, 2014. doi:10.1145/2603088.2603129.

[101] Ranko Lazić and Sylvain Schmitz. The Complexity of Coverability in ν-Petri Nets. In *LICS'16*, pages 467–476, New York, United States, 2016. doi:10.1145/2933575.2933593.

[102] J. Leroux and P. Schnoebelen. On functions weakly computable by Petri nets and vector addition systems. In *RP'14*, volume 8762 of *LNCS*, pages 190–202, 2014. doi:10.1007/978-3-319-11439-2_15.

[103] Jérôme Leroux. Distance between mutually reachable Petri net configurations. In *FSTTCS'19*, pages 47:1–47:14, 2019. doi:10.4230/LIPIcs.FSTTCS.2019.47.

[104] Jérôme Leroux. The reachability problem for Petri nets is not primitive recursive. *FOCS22*, abs/2104.12695, 2021. URL: https://arxiv.org/abs/2104.12695, doi:10.1145/3422822.

[105] Jérôme Leroux, M. Praveen, and Grégoire Sutre. Hyper-Ackermannian bounds for pushdown vector addition systems. In *LICS'14*, 2014. doi:10.1145/2603088.2603146.

[106] Jérôme Leroux, Grégoire Sutre, and Patrick Totzke. On the coverability problem for pushdown vector addition systems in one dimension. In *Automata, Languages, and Programming*, pages 324–336, 2015. doi:10.1007/978-3-662-47666-6_26.

[107] Richard J. Lipton. The reachability problem requires exponential space. Technical Report 062, Yale University, Department of Computer Science, January 1976.

[108] I. A. Lomazova. Modelling of multi-agent dynamic systems by nested Petri nets (in russian). *Programmnye Sistemy: Teoreticheskie Osnovy i Prilozheniya*, 28:143–156, 1998.

[109] Irina Lomazova. Modeling dynamic objects in distributed systems with nested Petri nets. *Fundamenta Informaticae*, 51:121–133, 06 2002.

[110] Irina A. Lomazova and Philippe Schnoebelen. Some decidability results for nested Petri nets. In *Perspectives of System Informatics*, pages 208–220, 2000. doi:10.1007/3-540-46562-6_18.

[111] P. Madhusudan and Gennaro Parlato. The tree width of auxiliary storage. In *POPL'11*, pages 283–294, 2011. doi:10.1145/1925844.1926419.

[112] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. Wiley Series in Parallel Computing. 1995. doi:10.1145/288197.581193.

[113] Gairatzhan Mavlankulov, Mohamed Othman, Sherzod Turaev, Mohd Hasan Selamat, Laula Zhumabayeva, and Tamara Zhukabayeva. Concurrently controlled grammars. *Kybernetika*, 54(4):748–764, 2018. doi:10.14736/kyb-2018-4-0748.

[114] Franz Mayr, Ramiro Visca, and Sergio Yovine. On-the-fly black-box probably approximately correct checking of recurrent neural networks. In *Proceedings of CD-MAKE 2020*, volume 12279 of *LNCS*, pages 343–363, 2020. doi:10.1007/978-3-030-57321-8_19.

[115] Franz Mayr and Sergio Yovine. Regular inference on artificial neural networks. In *CD-MAKE'18*, volume 11015 of *LNCS*, pages 350–369, 2018. doi:10.1007/978-3-319-99740-7_25.

[116] Franz Mayr, Sergio Yovine, and Ramiro Visca. Property checking with interpretable error characterization for recurrent neural networks. *Mach. Learn. Knowl. Extr.*, 3(1):205–227, 2021. doi:10.3390/make3010010.

[117] Richard Mayr. Combining petri nets and pa-processes. In *TACS '97*, volume 1281 of *LNCS*, pages 547–561, 1997. doi:10.1007/bfb0014567.

[118] Kurt Mehlhorn. Pebbling moutain ranges and its application of dcfl-recognition. In J. W. de Bakker and Jan van Leeuwen, editors, *ICALP '80*, volume 85 of *LNCS*, pages 422–435, 1980. doi:10.1007/3-540-10003-2_89.

[119] Maik Merten. *Active automata learning for real life applications*. PhD thesis, Dortmund University of Technology, 2013. doi:10.17877/DE290R-5169.

[120] M. Montali and Andrey Rivkin. Model checking Petri nets with names using data-centric dynamic systems. *Formal Aspects of Computing*, 28:615–641, 2016. doi:10.1007/s00165-016-0370-6.

[121] William Ogden. A helpful result for proving inherent ambiguity. *Mathematical systems theory*, 2(3):191–194, 1968. doi:10.1007/bf01694004.

[122] Takamasa Okudono, Masaki Waga, Taro Sekiyama, and Ichiro Hasuo. Weighted automata extraction from recurrent neural networks via regression on state spaces. In *AAAI'20*, pages 5306–5314, 2020. doi:10.1609/aaai.v34i04.5977.

[123] Christian W. Omlin and C. Lee Giles. Extraction of rules from discrete-time recurrent neural networks. *Neural Networks*, 9(1):41–52, 1996. doi:10.1016/0893-6080(95)00086-0.

[124] David Parnas. On a solution to the cigarette smoker's problem (without conditional statements). *Communications of the ACM*, 18:181–183, 1975. doi:10.1145/360680.360709.

[125] Suhas S. Patil. Limitations and capabilities of Dijkstra's semaphore primitives for coordination among processesre. Technical report, Massachusetts Institute of Technology, 1971.

[126] Doron A. Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. *Journal of Automata, Languages and Combinatorics*, 7(2):225–246, 2002.

[127] James Peterson. *Petri net theory and the modeling of systems.* 1981.

[128] James L. Peterson. *Modeling of parallel systems.* PhD thesis, Univercity of Stanford, 1973.

[129] Artturi Piipponen and Antti Valmari. Constructing minimal coverability sets. *Fundamenta Informaticae*, 143(3–4):393–414, 2016. doi:10.3233/fi-2016-1319.

[130] J. R. Quinlan. The effect of noise on concept learning. In *Machine Learning, An Artificial Intelligence Approach Volume II*, chapter 6, pages 149–166. 1986.

[131] Guillaume Rabusseau, Tianyu Li, and Doina Precup. Connecting weighted automata and recurrent neural networks through spectral learning. In *AISTATS'19*, volume 89, pages 1630–1639, 2019. URL: http://proceedings.mlr.press/v89/rabusseau19a/rabusseau19a.pdf.

[132] Charles Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*, 6(2):223 – 231, 1978. doi:10.1016/0304-3975(78)90036-1.

[133] Klaus Reinhardt. Reachability in Petri nets with inhibitor arcs. *Electr. Notes Theor. Comput. Sci.*, 223:239–264, 2008. doi:10.1016/j.entcs.2008.12.042.

[134] Pierre-Alain Reynier and Frédéric Servais. On the computation of the minimal coverability set of Petri nets. In *RP'19*, pages 164–177, 2019. doi:10.1007/978-3-030-30806-3_13.

[135] Pierre-Alain Reynier and Frédéric Servais. Minimal coverability set for Petri nets: Karp and Miller algorithm with pruning. *Fundamenta Informaticae*, 122(1–2):1–30, 2013. doi:10.3233/fi-2013-781.

[136] Fernando Rosa-Velardo. Depth boundedness in multiset rewriting systems with name binding. In *RP'04*, pages 161–175, 08 2010. doi:10.1007/978-3-642-15349-5_11.

[137] Fernando Rosa-Velardo and David de Frutos-Escrig. Name creation vs. replication in Petri net systems. In *ICATPN'07*, pages 402–422, 2007. doi:10.1007/978-3-540-73094-1_24.

[138] Fernando Rosa-Velardo and David de Frutos-Escrig. Decidability and complexity of Petri nets with unordered data. *Theoretical Computer Science*, 412(34):4439–4451, 2011. doi:10.1016/j.tcs.2011.05.007.

[139] Valentín Valero Ruiz, David de Frutos-Escrig, and Fernando Cuartero Gómez. On non-decidability of reachability for timed-arc Petri nets. In *PNPM'99*, pages 188–196, 1999. doi:10.1109/PNPM.1999.796565.

[140] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. 2020.

[141] Wonryong Ryou, Jiayu Chen, Mislav Balunovic, Gagandeep Singh, Andrei Marian Dan, and Martin T. Vechev. Fast and effective robustness certification for recurrent neural networks. *CoRR*, abs/2005.13300, 2020.

[142] Yasubumi Sakakibara. Efficient learning of context-free grammars from positive structural examples. *Inf. Comput.*, 97(1):23–60, 1992. doi:10.1016/0890-5401(92)90003-x.

[143] Philippe Schnoebelen. Revisiting Ackermann-hardness for lossy counter machines and reset Petri nets. In *MFCS'10*, volume 6281 of *LNCS*, pages 616–628, 2010. doi:10.1007/978-3-642-15155-2_54.

[144] Klaus U. Schulz and Stoyan Mihov. Fast string correction with Levenshtein automata. *Int. J. Document Anal. Recognit.*, 5(1):67–85, 2002. doi:10.1007/s10032-002-0082-8.

[145] Ray J. Solomonoff. A formal theory of inductive inference. *Inf. Control.*, 7(1, 2):1–22, 224–254, 1964. doi:10.1016/S0019-9958(64)90223-2.

[146] M. Stadel. A remark on the time complexity of the subtree problem. *Computing*, 19(4):297–302, 1978. doi:10.1007/bf02252027.

[147] Guo-Zheng Sun, C. Lee Giles, and Hsing-Hen Chen. The neural network pushdown automaton: Architecture, dynamics and training. In *APSDS'97*, volume 1387 of *LNCS*, pages 296–345, 1997. doi:10.1007/bfb0054003.

[148] Sebastian Thrun. Extracting rules from artifical neural networks with distributed representations. In *NIPS'94*, pages 505–512, 1994. URL: https://proceedings.neurips.cc/paper/1994/file/bea5955b308361a1b07bc55042e25e54-Paper.pdf.

[149] Leslie G. Valiant. A theory of the learnable. *Commun. ACM*, 27(11):1134–1142, 1984. doi:10.1145/800057.808710.

[150] R. Valk. Self-modifying nets, a natural extension of Petri nets. In *ICALP'78*, volume 62 of *LNCS*, pages 464–476, 1978. doi:10.1007/3-540-08860-1_35.

[151] Rüdiger Valk. Petri nets as token objects. In *Petri Nets'98*, pages 1–24, 1998. doi:10.1007/3-540-69108-1_1.

[152] Antti Valmari and Henri Hansen. Old and new algorithms for minimal coverability sets. *Fundamenta Informaticae*, 131(1):1–25, 2014. doi:10.3233/fi-2014-1002.

[153] W. M. P. van der. *Workflow Patterns*. 2018. doi:10.1007/978-1-4614-8265-9_826.

[154] W. M. P. van der Aalst. *Business Process Management*, pages 370–374. 2018. doi:10.1007/978-1-4614-8265-9_1179.

[155] Wil M. P. van der Aalst. Process mining. *CACM*, 55(8):76–83, 2012. doi:10.1007/978-3-662-49851-4.

[156] Gail Weiss, Yoav Goldberg, and Eran Yahav. Extracting automata from recurrent neural networks using queries and counterexamples. In *ICML'18*, volume 80, pages 5244–5253, 2018. URL: http://proceedings.mlr.press/v80/weiss18a/weiss18a.pdf.

[157] Gail Weiss, Yoav Goldberg, and Eran Yahav. Learning deterministic weighted automata with queries and counterexamples. In *NeurIPS'19*, pages 8558–8569, 2019. URL: https://proceedings.neurips.cc/paper/2019/file/d3f93e7766e8e1b7ef66dfdd9a8be93b-Paper.pdf.

[158] R. M. Wharton. Approximate language identification. *Information and Control*, 26(3):236 – 255, 1974. doi:10.1016/s0019-9958(74)91369-2.

[159] Alex Yakovlev, Luis Gomes, and Luciano Lavagno. *Hardware Design and Petri Nets*. 2000. doi:10.1007/978-1-4757-3143-9.

[160] Mitsuharu Yamamoto, Shogo Sekine, and Saki Matsumoto. Formalization of Karp-Miller tree construction on Petri nets. In *SIGPLAN'17*, pages 66–78, 2017. doi:10.1145/3018610.3018626.

[161] Daniel M. Yellin and Gail Weiss. Synthesizing context-free grammars from recurrent neural networks. In *TACAS'21*, volume 12651 of *LNCS*, pages 351–369, 2021. doi:10.26226/morressier.604907f41a80aac83ca25cf9.

[162] Daniel M. Yellin and Gail Weiss. Synthesizing context-free grammars from recurrent neural networks (extended version). *CoRR*, abs/2101.08200, 2021. URL: https://arxiv.org/abs/2101.08200, arXiv:2101.08200.

[163] Ryo Yoshinaka and Alexander Clark. Polynomial time learning of some multiple context-free languages with a minimally adequate teacher. In *FG'10*, LNCS, pages 192–207, 2010. doi:10.1007/978-3-642-32024-8_13.

[164] G. Zavattaro. When to move to transfer nets - on the limits of Petri nets as models for process calculi. In *Programming Languages with Applications to Biology and Security*, 2015. doi:10.1007/978-3-319-25527-9_22.

[165] Georg Zetzsche. The emptiness problem for valence automata or: Another decidable extension of Petri nets. In *RP'15*, volume 9328 of *LNCS*, pages 166–178, 2015. doi:10.1007/978-3-319-24537-9_15.

université PARIS-SACLAY | ÉCOLE DOCTORALE Sciences et technologies de l'information et de la communication (STIC)

Titre: Vérification de systèmes infinis et apprentissage automatique

Mots clés: réseaux de Petri, apprentissage automatique,vérification

Résumé: Cette thèse est découpée en trois parties. La première est consacrée à la vérification des réseaux de Petri, la seconde à la vérification des réseaux de Petri récursifs qui étendent les réseaux de Petri et la dernière vise à combiner l'apprentissage actif et la vérification.

Un réseau de Petri peut être analysé en calculant et étudiant son Clover, la représentation canonique de la sur-approximation vers le bas de son ensemble d'accessibilité. A l'aide de l'algorithme de Karp-Miller on peut calculer le Clover, mais cet algorithme est très inefficace et de plus sa preuve originelle de correction n'est pas satisfaisante. Il y a de nombreuses variantes de cet algorithme mais certaines sont incomplètes et d'autres requièrent une mémoire additionnelle de taille potentiellement ackermannienne. Enfin les preuves de correction sont souvent intriquées. Notre première contribution est la conception d'un algorithme complet incluant une borne théorique sur la taille de la mémoire additionelle. L'idée clef de cet algorithme est l'introduction d'un nouveau concept, appelé accélération. Plus précisément à l'aide des accélérations, nous avons pu:

1. simplifier la preuve de correction de l'algorithme de Karp-Miller;
2. de présenter la première modification simple de l'algorithme incomplet de construction du "Minimal Coverability Tree";
3. de prouver que la mémoire supplémentaire requise par notre algorithme est élémentaire (2-EXPSPACE);
4. de développer un prototype MinCov et de montrer expérimentalement qu'il est l'outil le plus efficace parmi ceux qui calculent le Clover.

Au début des années 2000, les réseaux de Petri récursifs (RPN) ont été introduits en vue de la modélisation et de l'analyse de la planification distribuée de systèmes multi-agents pour lesquels la présence de compteurs et la recursivité sont nécessaires. Bien que les RPN étendent strictement les réseaux de Petri et les grammaires algébriques, la plupart des problèmes usuels (accessibilité, terminaison, etc.) restent décidables. Pour presque tous les modèles incluant les réseaux de Petri et les grammaires algébriques, la complexité des problèmes de la couverture et de la terminaison est inconnue ou strictement plus grande que EXPSPACE. Ici, nous établissons que les problèmes de couverture, terminaison, caractère borné et finitude des RPN sont EXPSPACE-complets comme ceux des réseaux de Petri. Bien qu'ayant un grand pouvoir d'expression, les RPN souffrent de plusieurs limitations en terme de modélisation. Aussi nous introduisons les réseaux de Petri récursifs dynamiques (DRPN) qui répondent à ces limitations et par conséquent étendent le pouvoir d'expression des RPN. Les DRPN généralisent presque toutes les extensions de réseaux de Petri pour lesquelles le problème de couverture est décidable. Nous démontrons alors que le problème de couverture reste décidable pour les DRPN.

Dans la troisième partie, notre travail se concentre sur l'algorithme L* d'Angluin. Cet algorithme apprend l'automate fini déterministe (DFA) minimal d'un langage régulier à l'aide de questions d'appartenance et d'équivalence de langages. Sa version probabilistiquement approximativement correcte (PAC) remplace une question d'équivalence par un ensemble de questions aléatoires d'appartenance. Nous avons étudié comment la PAC version se comporte pour des machines qui sont obtenues en "bruitant" un DFA et si l'algorithme réduit le bruit. Nous établissons que la réduction du bruit dépend fortement de la nature du bruit et de sa quantité. De plus, nous utilisons cet algorithme pour développer une approche de vérification des réseaux neuronaux récurrents (RNN). Un DFA est appris comme une abstraction d'un RNN puis analysé à l'aide de techniques de "model checking". Nous établissons deux avantages de cette approche : lorsque la propriété n'est pas vérifiée les contre-exemples exhibés sont de petite taille et susceptibles d'être généralisés en un patron d'erreur mettant en évidence la nature de la faute du RNN.

Title: Verification of Infinite-State Systems and Machine Learning

Keywords: Petri nets, machine learning, verification

Abstract: This thesis consists of three parts. The first one is devoted to the verification of Petri nets, the second one to the verification of recursive Petri nets which extend Petri nets, and the final one aims at combining active learning and verification.

A Petri net can be analyzed by computing and studying its Clover, that is, the canonical representation of the downward over approximation of its reachability set. Using the Karp-Miller algorithm one can compute the Clover, but this algorithm is very inefficient and moreover, its original proof of correctness is not satisfying. Many variations of the original Karp-Miller algorithm computing the clover exist, but some are incomplete, others introduced an unknown supplementary memory size (possibly Ackermannian) and proofs are often heavy. Our first contribution is the design of a complete algorithm in such a way that we can theoretically bound the additional memory requirements. The key idea of this algorithm is the introduction of a new concept, called acceleration. More precisely, using accelerations, we were able:

1. to simplify the proof of correctness of the Karp-Miller algorithm;
2. to present the first simple modification of the original but incomplete Minimal Coverability Tree algorithm;
3. to prove that the supplementary memory needed by our algorithm is elementary (2-EXPSPACE);
4. to implement a prototype MinCov, showing experimentally that it is the most efficient one compared to other tools computing the Clover.

In the early two-thousands, Recursive Petri nets (RPN) have been introduced in order to model distributed planning of multiagent systems for which counters and recursivity were necessary. Although RPN strictly extend Petri nets and context-free grammars, most of the usual problems (reachability, termination, etc.) were shown to be decidable. For almost all other models extending Petri nets and context-free grammars, the complexity of coverability and termination are unknown or strictly larger than EXPSPACE. In contrast, we establish here that for RPN, the coverability, termination, boundedness and finiteness problems are EXPSPACE-complete as for Petri net. While having a great expressive power, RPN suffer several modeling limitations. We introduce Dynamic Recursive Petri nets (DRPN) which address these issues, extending the expressiveness of RPN. This model generalizes almost all previous known models, which extend the Petri net and keep the coverability problem decidable. Thus, we establish that the coverability problem is decidable for DRPN.

For active learning and formal methods, our work focuses on Angluin's L* algorithm. Angluin's algorithm learns the minimal deterministic finite automaton (DFA) of a regular language using membership and equivalence queries. Its probabilistic approximately correct (PAC) version substitutes an equivalence query by a set of random membership queries. Thus, it can be applied to any kind of device and may be viewed as synthesizing an automaton from observations of the device. We are interested in how the PAC version behaves for devices which are obtained from a DFA by introducing some noise. More precisely, we study whether the algorithm reduces the noise, producing a DFA closer to the original one than the noisy device. We found that the reduction of the noise strongly depends on the type of noise and its amount. Moreover, we use this algorithm to develop a property-directed approach for verification of recurrent neural networks (RNNs). It learns a DFA as a surrogate model from a given RNN, which is then analyzed using model checking as a verification technique. We show that this not only allows us to discover small counterexamples fast, but also to generalize them by pumping towards faulty flows, hinting at the underlying error in the RNN.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France