Universidad de Buenos Aires

TESIS DE GRADO DE INGENIERÍA EN INFORMÁTICA

Compile-time Deadlock Detection in Rust using Petri Nets

Autor:

Horacio Lisdero Scaffino (100132)
hlisdero@fi.uba.ar

Director:

Ing. Pablo A. Deymonnaz
pdeymon@fi.uba.ar

Departamento de Computación

Facultad de Ingeniería

10 de marzo de 2023

Contents

1 Introduction **3**
1.1 Petri nets . . . 3
1.1.1 Overview . . . 3
1.1.2 Formal mathematical model . . . 5
1.1.3 Transition firing . . . 6
1.1.4 Modeling examples . . . 7
1.1.5 Important properties . . . 11
1.2 Reachability Analysis . . . 13
1.3 The Rust programming language . . . 18
1.4 Deadlocks . . . 18
1.5 Lost signals . . . 18
1.6 Compiler arquitecture . . . 18
1.7 Model checking . . . 18

2 Design of the proposed solution **19**
2.1 Rust compiler: *rustc* . . . 20
2.2 Mid-level Intermediate Representation (MIR) . . . 20
2.3 Entry point for the translation . . . 20
2.4 Function calls . . . 20
2.5 Function memory . . . 20
2.6 MIR function . . . 20
2.6.1 Basic blocks . . . 20
2.6.2 Statements . . . 20
2.6.3 Terminators . . . 20
2.7 Panic handling . . . 20
2.8 Multithreading . . . 20
2.9 Emulation of Rust synchronization primitives . . . 20
2.9.1 Mutex (`std::sync::Mutex`) . . . 20
2.9.2 Mutex lock guard (`std::sync::MutexGuard`) . . . 20
2.9.3 Condition variables (`std::sync::Condvar`) . . . 20
2.9.4 Atomic Refence Counter (`std::sync::Arc`) . . . 20

3 Testing the implementation **21**
3.1 Unit tests 21
3.2 Integration tests 21
3.3 Generating the MIR representation 21
3.4 Visualizing the result 21

4 Conclusions **22**

5 Future work **23**

6 Related work **24**

Chapter 1

Introduction

1.1 Petri nets

1.1.1 Overview

Petri nets are a graphical and mathematical modeling tool used to describe and analyze the behavior of concurrent systems. They were introduced by the German researcher Carl Adam Petri in his doctoral dissertation [Petri, 1962] and have since been applied in a variety of fields such as computer science, engineering, and biology. A concise summary of the theory of Petri nets, its properties, analysis and applications can be found in [Murata, 1989].

A Petri net is a bipartite, directed graph consisting of a set of places, transitions and arcs. There are two types of nodes, namely places and transitions. Places represent the state of the system, while transitions represent events or actions that can occur. Arcs connect places to transitions or transitions to places. There can be no arcs between places nor transitions, thus preserving the bipartite property.

Places may hold zero or more tokens. Tokens are used to represent the presence or absence of entities in the system, such as resources, data, or processes. In the most simple class of Petri nets, tokens do not carry any information and they are indistinguishable from one another. The number of tokens at a place or the simple presence of a token is what conveys meaning in the net. Tokens are consumed and produced as transitions fire, giving the impression that they move through the arcs.

In the conventional graphical representation, places are depicted using circles, while transitions are depicted as rectangles. Tokens are represented as black dots inside of the places, as seen in Fig. 1.1.

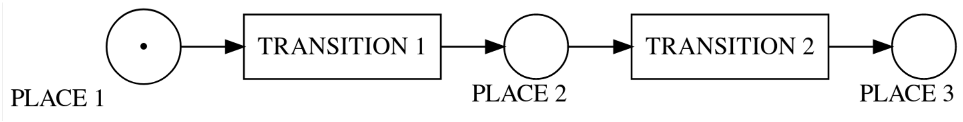

Figure 1.1: Example of a Petri net. PLACE 1 contains a token.

When a transition fires, it consumes tokens from its input places and produces tokens in its output places, reflecting a change in the state of the system. The firing of a transition is enabled when there are sufficient tokens in its input places. In Fig. 1.2, we can see how successive firings happen.

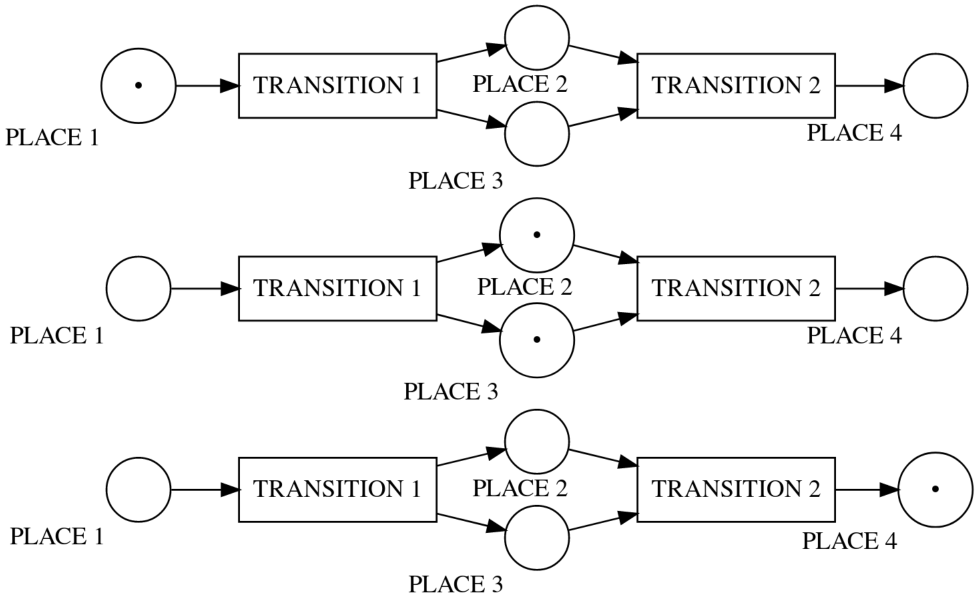

Figure 1.2: Example of transition firing: Transition 1 fires first, then transition 2 fires.

The firing of enabled transitions is not deterministic, i.e., they fire randomly as long as they are enabled. A disabled transition is considered *dead* if there is no reachable state in the system that can lead to the transition being enabled. If all the transitions in the net are dead, then the net is considered *dead* too. This state is analogous to the deadlock of a computer program.

Petri nets can be used to model and analyze a wide range of systems, from simple systems with a few components to complex systems with many interacting components. They can be used to detect potential problems in a system, optimize system performance and design and implement systems more effectively.

They can also be used to model industrial processes [Van der Aalst, 1994], to validate software requirements expressed as use cases [Silva and Dos Santos, 2004] or to specify and analyze real-time systems [Kavi et al., 1996].

In particular, Petri nets can be used to detect deadlocks in source code by modeling the input program as a Petri net and then analyzing the structure of the resulting net. It will be shown that this approach is formally sound and practicably amenable to source code written in the Rust programming language.

1.1.2 Formal mathematical model

A Petri net is a particular kind of bipartite, weighted, directed graph, equipped with an initial state called the *initial marking*, M_0. For this work, the following general definition of a Petri net taken from [Murata, 1989] will be used.

Definition 1: Petri net

A Petri net is a 5-tuple, $PN = (P, T, F, W, M_0)$ *where:*

$P = \{p_1, p_2, \ldots, p_m\}$ *is a finite set of places,*
$T = \{t_1, t_2, \ldots, t_n\}$ *is a finite set of transitions,*
$F \subseteq (P \times T) \cup (T \times P)$ *is a set of arcs (flow relation),*
$W : F \leftarrow \{1, 2, 3, ...\}$ *is a weight function for the arcs,*
$M_0 : P \leftarrow \{0, 1, 2, 3, ...\}$ *is the initial marking,*
$P \cap T = \varnothing$ *and* $P \cup T \neq \varnothing$

In the graphical representation, arcs are labeled with their weight, which is a non-negative integer k. Usually, the weight is omitted if it is equal to 1. A k-weighted arc can be interpreted as a set of k distinct parallel arcs.

A *marking (state)* associates with each place a non-negative integer l. If a marking assigns to place p a non-negative integer l, we say that p is *marked with* l *tokens*. Pictorially, we denote his by placing l black dots (tokens) in place p. The *pth* component of M, denoted by $M(p)$, is the number of tokens in place p.

An alternative definition of Petri nets uses *bags* instead of a set to define the arcs, thus allowing multiple elements to be present. It can be found in the literature, e.g., [Peterson, 1981, Definition 2.3].

As an example, consider the Petri net $PN_1 = (P, T, F, W, M)$ where:

$$P = \{p_1, p_2\},$$
$$T = \{t_1, t_2\},$$
$$F = \{(p_1, t_1), (p_2, t_2), (t_1, p_2), (t_2, p_2)\},$$
$$W(a_i) = 1 \quad \forall a_i \in F$$
$$M(p_1) = 0, M(p_2) = 0$$

This net contains no tokens and all the arc weights are equal to 1. It is shown in Fig. 1.3.

Figure 1.3: Example of a small Petri net containing a self-loop

Fig. 1.3 contains an interesting structure that we will encounter later. This motivates the following definition.

Definition 2: Self-loop

A place node p and a transition node t define a self-loop if p is both an input place and an output place of t.

In most cases, we are interested in Petri nets containing no self-loops, which are called *pure*.

Definition 3: Pure Petri net

A Petri net is said to be pure if it has no self-loops.

Moreover, if every arc weight is equal to one, we call the Petri net *ordinary*.

Definition 4: Ordinary Petri net

A Petri net is said to be ordinary if all of its arc weights are 1's, i.e.

$$W(a) = 1 \quad \forall a \in F$$

1.1.3 Transition firing

The transition firing rule is the core concept in Petri nets. Despite being deceptively simple, its implications are far-reaching and complex.

Definition 5: Transition firing rule

Let $PN = (P, T, F, W, M_0)$ be a Petri net.

(i) A transition t is said to be enabled if each input place p of t is marked with at least $W(p,t)$ tokens, where $W(p,t)$ is the weight of the arc from p to t.

(ii) An enabled transition may or may not fire (depending on whether or not the event takes place).

(iii) A firing of an enabled transition t removes $W(t,p)$ tokens from each input place p of t, where $W(t,p)$ is the weight of the arc from t to p.

Whenever several transitions are enabled for a given marking M, any one of them can be fired. The choice is nondeterministic. Two enabled transitions are said to be in *conflict* if the firing of one of the transitions will disable the other transition. In this case, the transitions compete for the token placed in a shared input place.

If two transition t_1 and t_2 are enabled in some marking but are not in conflict, they can fire in either order, i.e. t_1 then t_2 or t_2 then t_1. Such transitions represent events that can occur concurrently or in parallel. In this sense, the Petri net model adopts an *interleaved model of parallelism*, that is, the behavior of the system is the result of an arbitrary interleaving of the parallel events.

Transitions without input places or output places receive a special name.

Definition 6: Source transition

A transition without any input place is called a source transition.

Definition 7: Sink transition

A transition without any output place is called a sink transition.

It is important to note that a source transition is unconditionally enabled and produces tokens without consuming any, while the firing of a sink transition consumes tokens without producing any.

1.1.4 Modeling examples

In this subsection, several simple examples are presented to introduce some basic concepts of Petri nets that are useful in modeling. This subsection has been adapted from [Murata, 1989].

For other modeling examples, such as the mutual exclusion problem, semaphores as proposed by Edsger W. Dijkstra, the producer/consumer problem and the dining philosophers problem, the reader is referred to [Peterson, 1981, Chapter 3] and [Reisig, 2013].

Finite-state machines

Finite state machines can be represented by a subclass of Petri nets.

As an example of a finite-state machine, consider a coffee vending machine. It accepts 1 € or 2 € coins and sells two types of coffee, the first costs 3 € and the second 4 €. Assume that the machine can hold up to 4 € and does not return any change. Then, the state diagram of the machine can be represented by the Petri net shown in Fig. 1.4.

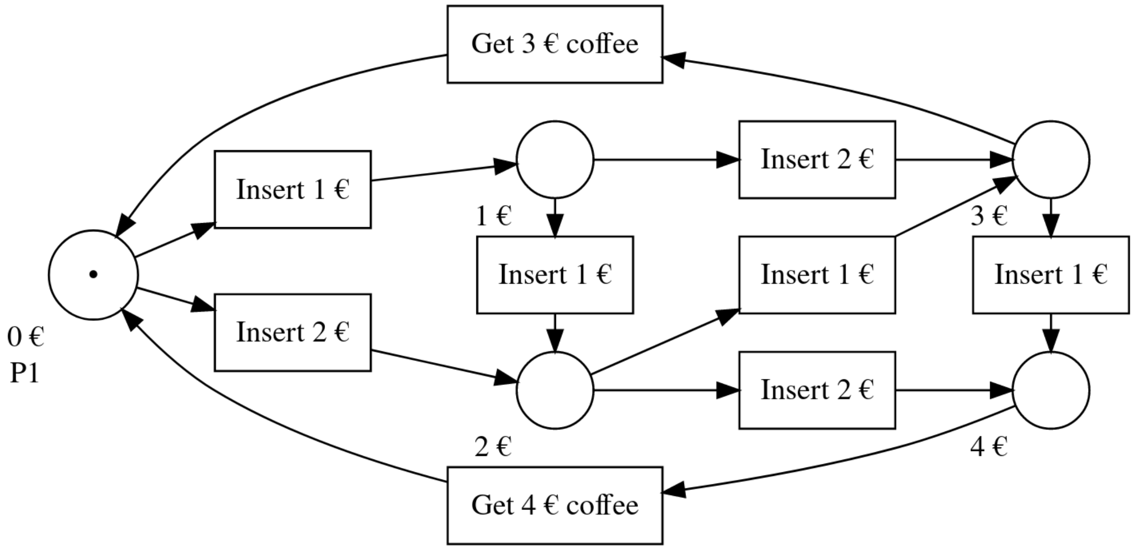

Figure 1.4: The Petri net for a coffee vending machine. It is equivalent to a state diagram.

The transitions represent the insertion of a coin of the labeled value, e.g. “Insert 1 € coin”. The places represent a possible state of the machine, i.e. the amount of money currently stored inside. The place labeled P1 is marked with a token and corresponds to the initial state of the system.

We can now present the following definition of this subclass of Petri nets.

Definition 8: State machines

A Petri net in which each transition has exactly one incoming arc and exactly one outgoing arc is known as a state machine.
Any finite-state machine (or its state diagram) can be modeled with a state machine.

The structure of a place p_1 having two (or more) output transitions t_1 and t_2 is called a *conflict*, *decision* or *choice*, depending on the application. This is seen in the initial place P1 of Fig. 1.4, where the user must select which coin to insert.

Parallel activities

Contrary to finite-state machines, Petri nets can also model parallel or concurrent activities. In Fig. 1.5 an example of this is shown, where the net represents the division of a bigger task into two subtasks that may be executed in parallel.

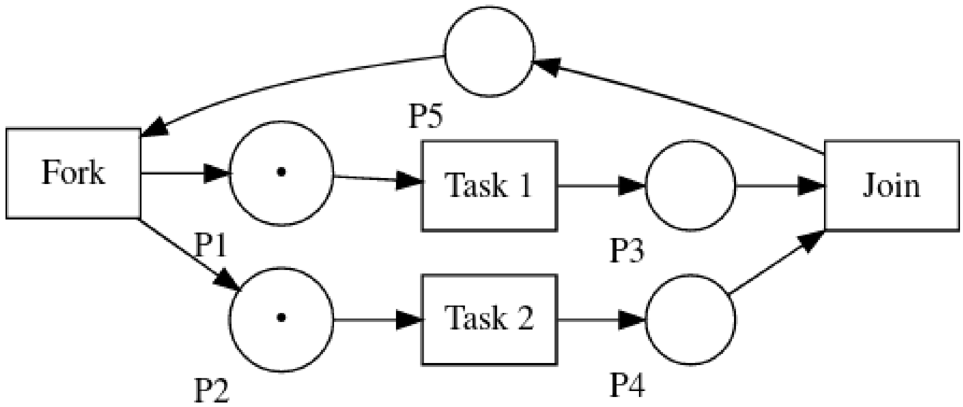

Figure 1.5: The Petri net depicting two parallel activities in a fork-join fashion.

The transition "Fork" will fire before "Task 1" and "Task 2" and that "Join" will only fire after both tasks are complete. But note that the order in which "Tast 1" and "Tast 2" execute is non-deterministic. "Task 1" could fire before, after or at the same time that "Task 2". It is precisely this property of the firing rule in Petri nets that allows the modeling of concurrent systems.

Definition 9: Concurrency in Petri nets

Two transitions are said to be concurrent if they are causally independent, i.e. the firing of one transition does not cause and is not triggered by the firing of the other.

Note that each place in the net in Fig. 1.5 has exactly one incoming arc and one outgoing arc. This subclass of Petri nets allows the representation of concurrency but not decisions (conflicts).

Definition 10: Marked graphs

A Petri net in which each place has exactly one incoming arc and exactly one outgoing arc is known as a marked graph.

Communication protocols

Communications protocols can also be represented in Petri nets. Fig. 1.6 illustrates a simple protocol in which Process 1 sends messages to Process 2 and waits for an acknowledgment to be received before continuing. Both processes communicate through a buffered channel whose maximum capacity is one message. Therefore, only one message may be traveling between the processes at any given time. For simplicity, no timeout mechanism was included.

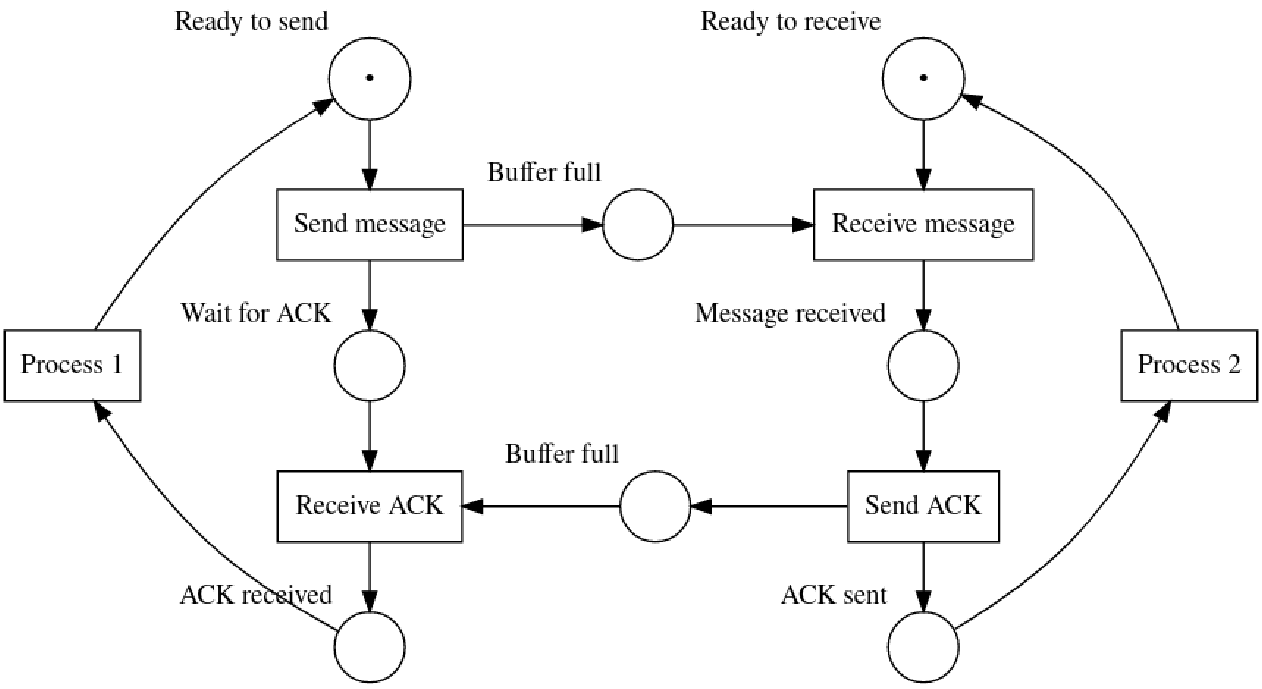

Figure 1.6: A simplified Petri net model of a communication protocol.

A timeout for the send operation could be incorporated into the model by adding a transition $t_{timeout}$ with edges from "Wait for ACK" to "Ready to send". This maps the decision between receiving the acknowledgment and the timeout.

Synchronization control

In a multithreaded system, resources and information are shared among several threads. This sharing must be controlled or synchronized to ensure the correct operation of the overall system. Petri nets have been used to model a variety of synchronization mechanisms, including the mutual exclusion, readers-writers and producers-consumers problems [Murata, 1989].

A Petri net for a readers-writers system with k processes is shown in Fig. 1.7. Each token represents a process and the choice of T1 or T2 represents whether the process performs a read or a write operation.

It makes use of weighted edges to remove atomically $k-1$ tokens from P3 before performing a write (transition T2), thus ensuring that no readers are present in the right loop of the net.

At most k processes may be reading at the same same, but when one process is reading, no process is allowed to read, that is P2 will be empty. It can be easily verified that the mutual exclusion property is satisfied for the system.

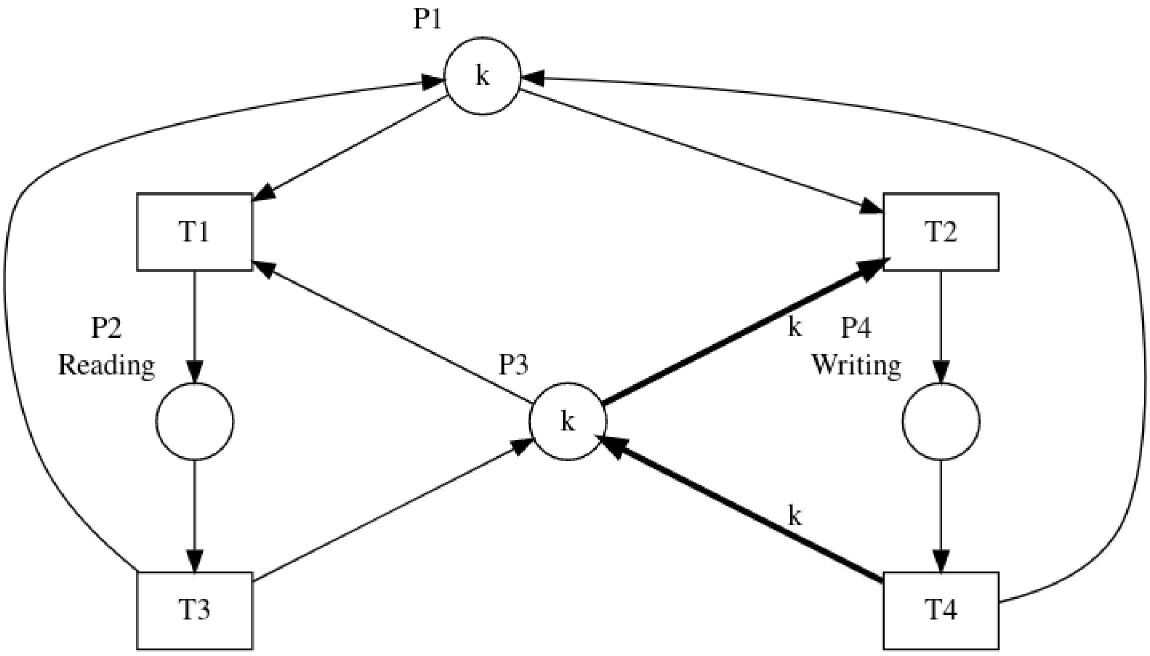

Figure 1.7: A Petri net system with k processes that either read or write.

It should be pointed out that this system is not free from starvation, since there is no guarantee that a write operation will eventually take place. The system is on the other hand free from deadlocks.

1.1.5 Important properties

In this subsection, we will look at important concepts for the analysis of Petri nets that will facilitate the understanding of the nets we will be dealing with in the rest of the work.

Reachability

Reachability is one of the most important questions when studying the dynamic properties of a system. The firing of enabled transitions causes changes in the location of the tokens. In other words, it changes the marking M. A sequence of firings creates a sequence of markings where each marking may be denoted as a vector of length n, with n being the number of places in the Petri net.

A *firing* or *occurence sequence* is denoted by $\sigma = M_0\ t_1\ M_1\ t_2\ M_2\ \cdots\ t_l\ M_l$ or simply $\sigma = t_1\ t_2\ \cdots\ t_l$, since the markings resulting from each firing are derived from the transition firing rule described in Sec. 1.1.3.

Definition 11: Reachability

We say that a marking M is reachable from M_0 if there exists a firing sequence σ such that M is contained in σ.

The set of all possible markings reachable from M_0 is denoted by $R(N, M_0)$ or more simply $R(M_0)$ when the net meant is clear. This set is called the *reachability set*.

A problem of utmost importance in the theory of Petri nets can be presented then, namely the *reachability problem*: Finding if $M_n \in R(M_0, N)$ for a given net and initial marking.

In some applications, we are just interested in the markings of a subset of places and we can ignore the remaining ones. This leads to a variation of the problem known as the *submarking reachability problem*.

It has been shown that the reachability problem is decidable [Mayr, 1981]. Nevertheless, it was also shown that it takes exponential space (formally, it is EXPSPACE-hard) [Lipton, 1976]. New methods have been proposed to make the algorithms more efficient [Küngas, 2005]. Recently, [Czerwiński et al., 2020] improved the lower bound and showed that the problem is not ELEMENTARY. These results highlight that the reachability problem is still an active area of research in theoretical computer science.

For this and other key problems, the most important theoretical results obtained up to 1998 are detailed in [Esparza and Nielsen, 1994].

Boundedness and safeness

During the execution of a Petri net, tokens may accumulate in some places. Applications need to ensure that the number of tokens in a given place does not exceed a certain tolerance. For example, if a place represents a buffer, we are interested that the buffer will never overflow.

Definition 12: Boundedness

A place in a Petri net is k-bounded or k-safe if the number of tokens in that place can not exceed a finite integer k for any marking reachable from M_0.
A Petri net is k-bounded or simply bounded if all places are bounded.

Safeness is a special case of boundedness. It occurs when the place contains either 1 or 0 tokens during execution.

Definition 13: Safeness

A place in a Petri net is safe if the number of tokens in that place never exceeds one.
A Petri net is safe if each place in that net is safe.

The nets in Fig. 1.4, 1.5 and 1.6 are all safe.

The net in Fig. 1.7 is k-bounded because all its places are k-bounded.

Liveness

The concept of liveness is analogous to the complete absence of deadlocks in computer programs.

Definition 14: Liveness

A Patri net (N, M_0) is said to be live (or equivalently M_0 is said to be a live marking for N) if, for every marking reachable from M_0, it is possible to fire any transition of the net by progressing through some firing sequence.

When a net is live, it can always continue executing, no matter the transitions that fired before. Eventually, every transition can be fired again. If a transition can be fired only once and there is no way to enable it again, then the net is not live.

This is equivalent to saying that the Petri net is *deadlock-free*. Let us now define what constitutes a deadlock and show examples of it.

Definition 15: Deadlock in Petri nets

A deadlock in a Patri net is a transition (or a set of transitions) that can not fire for any marking reachable from M_0. The transition (or a set of transitions) can not become enabled again after a certain point in the execution.

A transition is *live* if it is not deadlocked. If a transition is live, it is always possible to pick a suitable firing to get from the current marking to a marking that enables the transition.

The nets in Fig. 1.4, 1.5 and 1.6 are all live. In all these cases, after some firings, the net returns to the initial state and can restart the cycle.

The net in Fig. 1.1 is not live. After two firings it finishes executing and nothing more can happen. The net in Fig. 1.3 is also not live, because T1 will only execute once and only T2 can be enabled from that point on.

1.2 Reachability Analysis

Having introduced the reachability set $R(N, M_0)$ in Sec. 1.1.5, we can now present the major analysis technique, which has been used with Petri nets: the *reachability tree*.

We will run the algorithm for constructing the reachability tree step by step and then present its advantages and drawbacks. In general terms, the reachability tree has the following structure. Nodes represent markings generated from M_0, the root of the tree, and its successors, and each arc represents a transition firing, which transforms one marking into another.

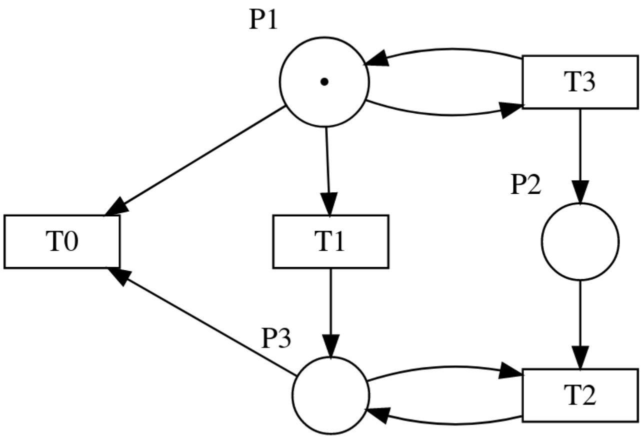

Figure 1.8: A marked Petri net for illustrating the construction of a reachability tree.

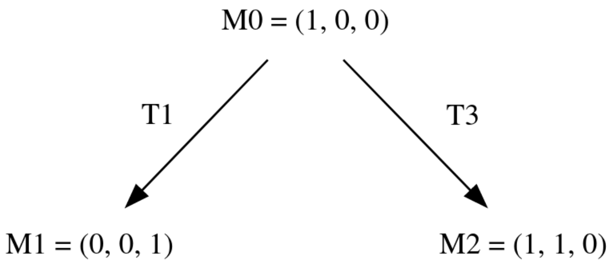

Figure 1.9: The first step building the reachability tree for the Petri net in Fig. 1.8.

Consider the Petri net shown in Fig. 1.8. The initial marking is $(1, 0, 0)$. In this initial marking, two transitions are enabled: T1 and T3. Given that we would like to obtain the entire reachability set, we define a new node in the reachability tree for each reachable marking, which results from firing each transition. An arc, labeled by the transition fired, leads from the initial marking (the root of the tree) to each of the new markings. After this first step (Fig. 1.9), the tree contains all markings that are immediately reachable from the initial marking.

Now we must consider all markings reachable from the leaves of the tree.

From marking $(0, 0, 1)$ we cannot fire any transition. This is known as a *dead marking*. In other words, it is a "dead-end" node. This class of end-states is particularly relevant for deadlock analysis.

From the marking on the right of the tree, denoted $(1, 1, 0)$, we can fire T1 or T3. If we fire T1, we obtain $(0, 1, 1)$ and if T3 fires, the resulting marking is $(1, 2, 0)$. This produces the tree of Fig. 1.10.

Note that starting with marking $(0, 1, 1)$, only the transition T2 is enabled, which will lead to a marking $(0, 0, 1)$ that was already seen before. If instead we take $(1, 2, 0)$ we have again the

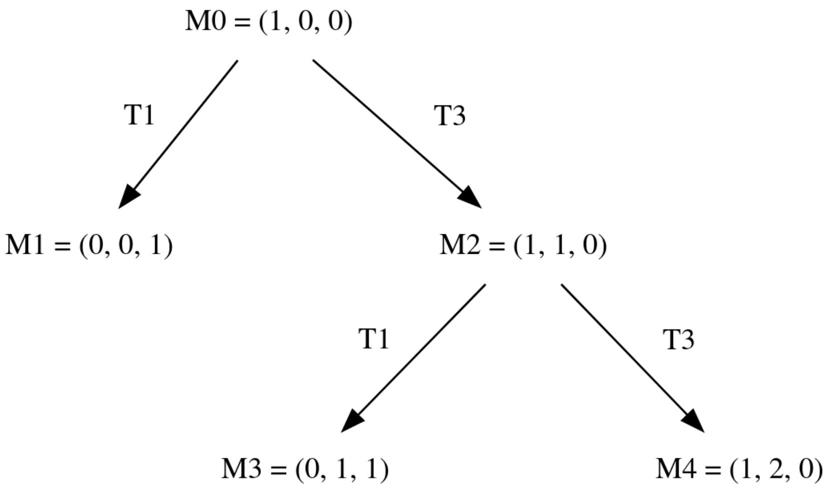

Figure 1.10: The second step building the reachability tree for the Petri net in Fig. 1.8.

same possibilities as starting from $(1, 1, 0)$. It is easy to see that the tree will continue to grow down that path. The tree is therefore infinite and this is because the net in Fig. 1.8 is not bounded. See Fig. 1.11 for the abbreviated final result.

The previously presented method enumerates the elements in the reachability set. Every marking in the reachability set will be produced, and so for any Petri net with an infinite reachability set (i.e. an infinite number of possible states), the corresponding tree would also be infinite. Nonetheless, this opposite is not true. A Petri net with a finite reachability set can have an infinite tree (see Fig. 1.12). This net is even *safe*. In conclusion, dealing with a bounded or safe net is not a guarantee that the total number of reachable states will be finite.

For the reachability tree to be a useful analysis tool, it is necessary to devise a method to limit it to a finite size. This implies in general a certain loss of information since the method will have to map an infinite number of reachable markings onto a single element. The reduction to a finite representation may be accomplished by the following means.

Notice on one hand that we may encounter duplicate nodes in our tree and we always naively treat them as new. This is illustrated most clearly in Fig. 1.12. It is thus possible to stop the exploration of the successors of a duplicated node.

Notice on the other hand that some markings are strictly different from previously seen markings but they enable the same set of transitions. We say in this case that the marking with additional tokens *covers* the one that has the minimum number of tokens needed to enable the set of transitions in question. Firing some transitions may allow us to accumulate an arbitrary number of tokens in one place. For example firing T3 in the Petri net seen in Fig. 1.8 exhibits exactly this behavior. Therefore, it would suffice to mark the accumulating place with a special label ω, which stands for infinity since we could get as many tokens as we wish in that place.

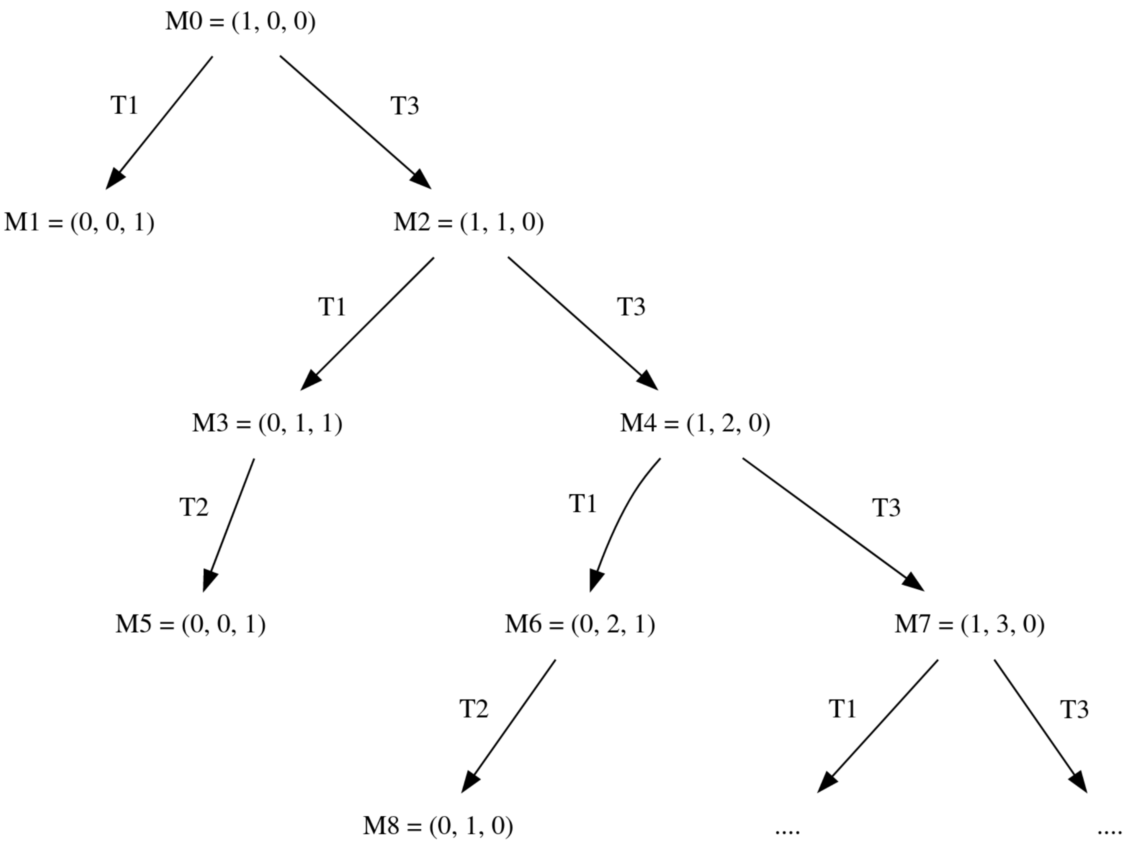

Figure 1.11: The infinite reachability tree for the Petri net in Fig. 1.8.

For instance, the result of converting the tree of Fig. 1.11 to a finite tree is shown in Fig. 1.13. For more details about

1. the technique for representing infinite reachability trees using ω,
2. a definition of the algorithm and precise steps for constructing the reachability tree,
3. mathematical proof that the reachability tree generated by it is finite,
4. and the distinction between the reachability tree and the *reachability graph*

the reader is referred to [Murata, 1989] and [Peterson, 1981]. These concepts are beyond the scope of this work and are not required in the following chapters.

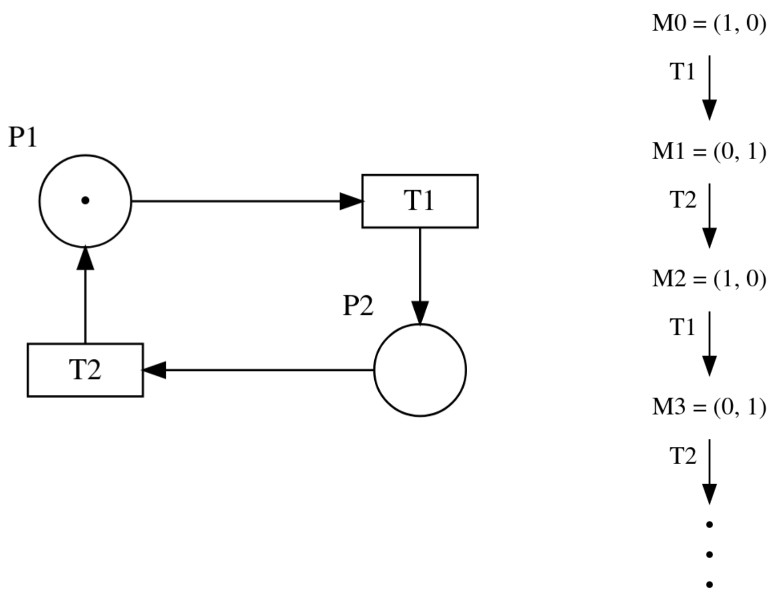

Figure 1.12: A simple Petri net with an infinite reachability tree.

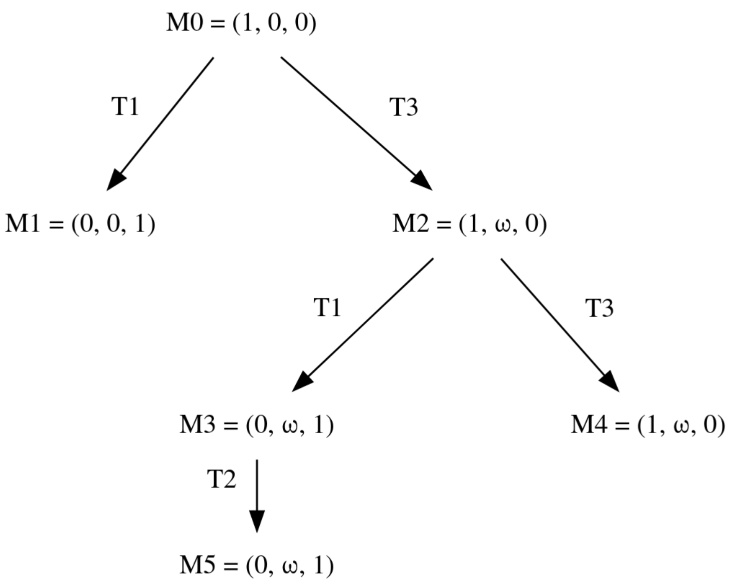

Figure 1.13: The finite reachability tree for the Petri net in Fig. 1.8.

1.3 The Rust programming language

1.4 Deadlocks

1.5 Lost signals

1.6 Compiler arquitecture

1.7 Model checking

Chapter 2

Design of the proposed solution

2.1 Rust compiler: *rustc*

2.2 Mid-level Intermediate Representation (MIR)

2.3 Entry point for the translation

2.4 Function calls

2.5 Function memory

2.6 MIR function

2.6.1 Basic blocks

2.6.2 Statements

2.6.3 Terminators

2.7 Panic handling

2.8 Multithreading

2.9 Emulation of Rust synchronization primitives

2.9.1 Mutex (`std::sync::Mutex`)

2.9.2 Mutex lock guard (`std::sync::MutexGuard`)

2.9.3 Condition variables (`std::sync::Condvar`)

2.9.4 Atomic Refence Counter (`std::sync::Arc`)

Chapter 3

Testing the implementation

3.1 Unit tests

3.2 Integration tests

3.3 Generating the MIR representation

3.4 Visualizing the result

Chapter 4

Conclusions

Chapter 5

Future work

Chapter 6

Related work

Bibliography

[Czerwiński et al., 2020] Czerwiński, W., Lasota, S., Lazić, R., Leroux, J., and Mazowiecki, F. (2020). The reachability problem for petri nets is not elementary. *Journal of the ACM (JACM)*, 68(1):1–28. https://arxiv.org/abs/1809.07115.

[Esparza and Nielsen, 1994] Esparza, J. and Nielsen, M. (1994). Decidability issues for petri nets. *BRICS Report Series*, 1(8). https://tidsskrift.dk/brics/article/download/21662/19099/49254.

[Kavi et al., 1996] Kavi, K. M., Sheldon, F. T., and Reed, S. (1996). Specification and analysis of real-time systems using csp and petri nets. *International Journal of Software Engineering and Knowledge Engineering*, 6(02):229–248.

[Küngas, 2005] Küngas, P. (2005). Petri net reachability checking is polynomial with optimal abstraction hierarchies. In *Abstraction, Reformulation and Approximation: 6th International Symposium, SARA 2005, Airth Castle, Scotland, UK, July 26-29, 2005. Proceedings 6*, pages 149–164. Springer. Available PDF online.

[Lipton, 1976] Lipton, R. J. (1976). The reachability problem requires exponential space. *Technical Report 63, Department of Computer Science, Yale University*. http://cpsc.yale.edu/sites/default/files/files/tr63.pdf.

[Mayr, 1981] Mayr, E. W. (1981). An algorithm for the general petri net reachability problem. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, page 238–246, New York, NY, USA. Association for Computing Machinery.

[Murata, 1989] Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580. http://www2.ing.unipi.it/ a009435/issw/extra/murata.pdf.

[Peterson, 1981] Peterson, J. L. (1981). *Petri Net Theory and the Modeling of Systems.* Prentice-Hall.

[Petri, 1962] Petri, C. A. (1962). Kommunikation mit Automaten. *Institut für Instrumentelle Mathematik*, 3. http://edoc.sub.uni-hamburg.de/informatik/volltexte/2011/160/.

[Reisig, 2013] Reisig, W. (2013). *Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies*. Springer-Verlag Berlin Heidelberg, 1 edition.

[Silva and Dos Santos, 2004] Silva, J. R. and Dos Santos, E. A. (2004). Applying petri nets to requirements validation. *IFAC Proceedings Volumes*, 37(4):659–666.

[Van der Aalst, 1994] Van der Aalst, W. (1994). Putting high-level petri nets to work in industry. *Computers in industry*, 25(1):45–54.