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

List of Figures

Listings

Acronyms

ART Android Runtime. 22

AST abstract syntax tree. 32

CPN Colored Petri nets. 43

DBMS Database management systems. 28

FSM Finite-state machine. 10

JIT just-in-time. 32

LTO link time optimization. 33

MIR Mid-level Intermediate Representation. 35

NT-Petri nets Nondeterministic Transitioning Petri nets. 43

OS operating system. 27

PN Petri nets. 5

RAG Resource Allocation Graph. 28

RFCs Requests for Comments. 21

TWF transaction-wait-for. 28

UB Undefined Behavior. 24, 35

Chapter 1

Introduction

1.1 Petri nets

1.1.1 Overview

Petri nets (PN) 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.

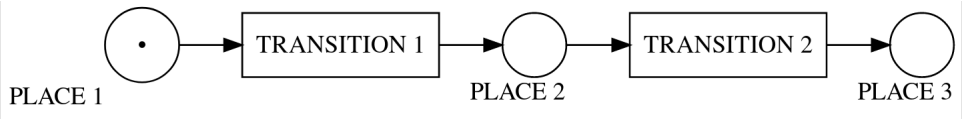

Figure 1.1: Example of a Petri net. PLACE 1 contains a token.

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.

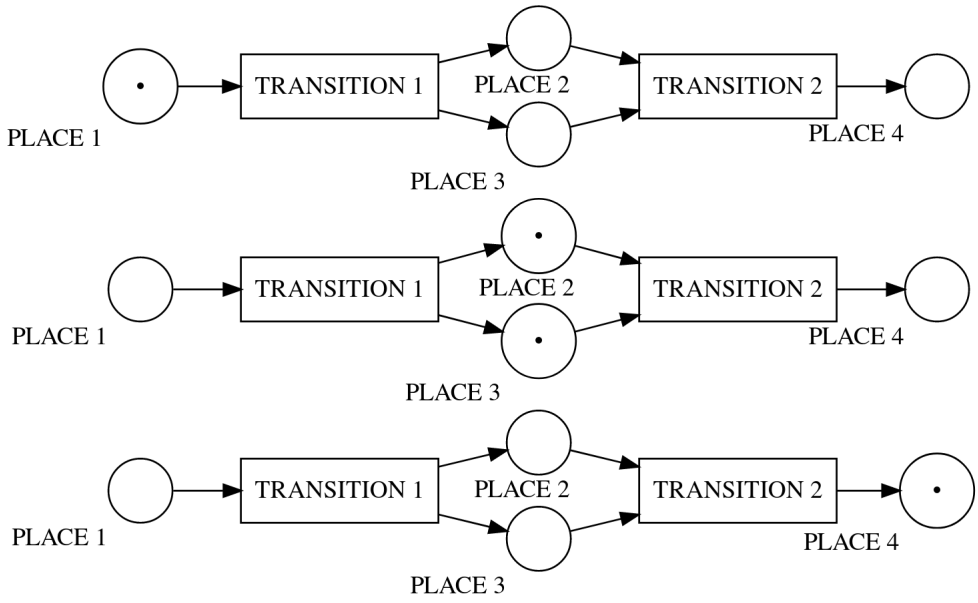

Figure 1.2: Example of transition firing: Transition 1 fires first, then transition 2 fires.

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 machine (FSM) 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.

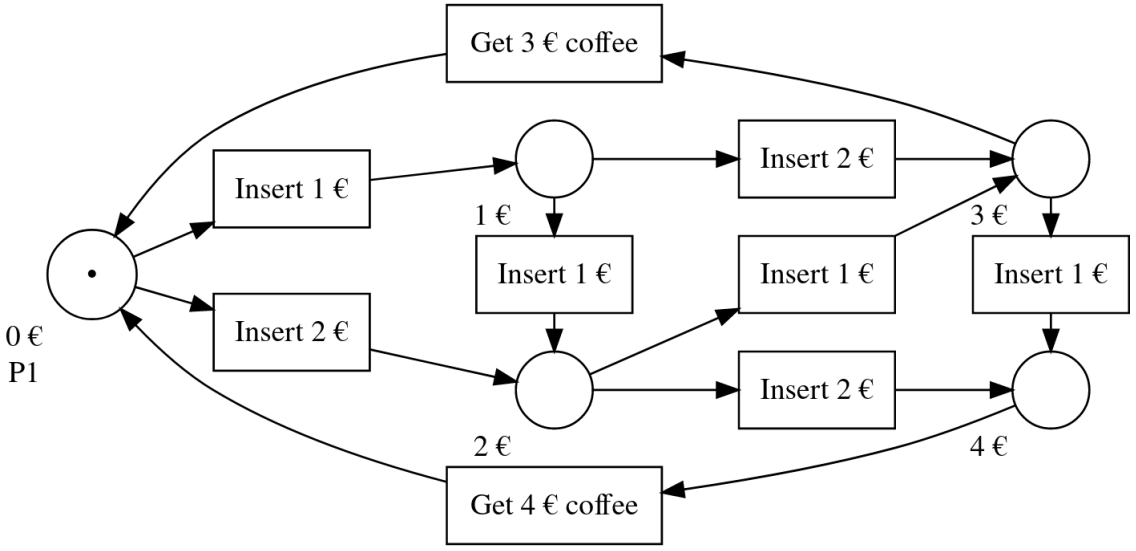

Figure 1.4: The Petri net for a coffee vending machine. It is equivalent to a state diagram.

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 FSM (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.

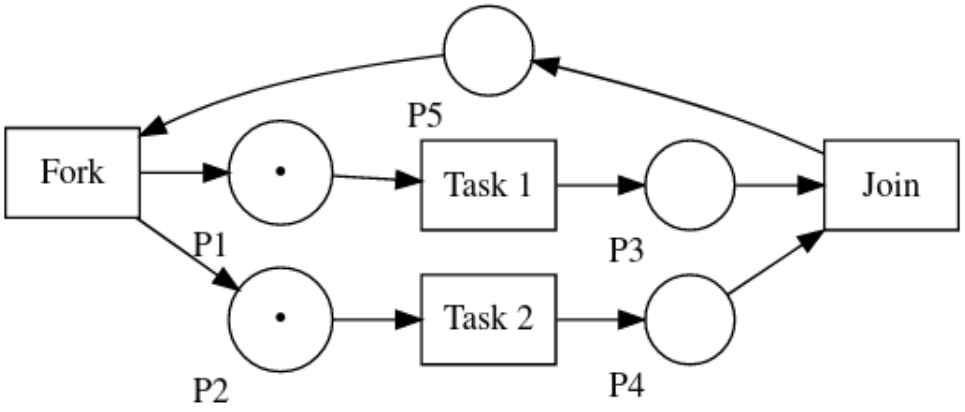

Figure 1.5: The Petri net depicting two parallel activities in a fork-join fashion.

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.

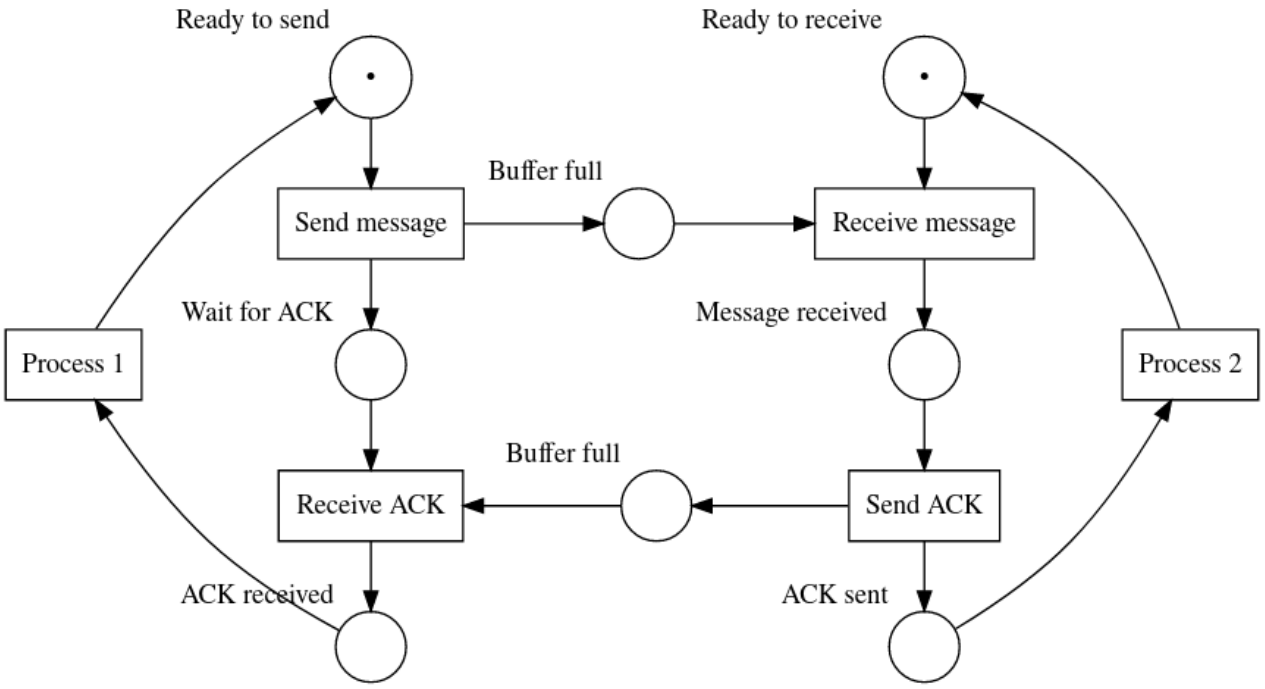

Figure 1.6: A simplified Petri net model of a communication protocol.

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.

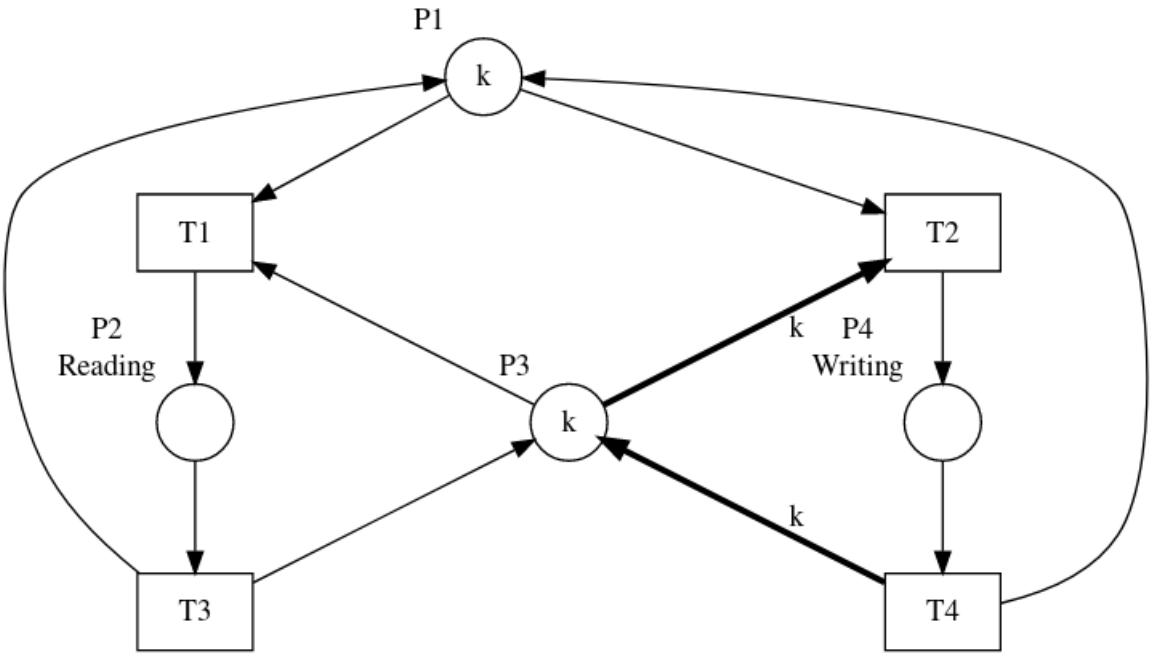

Figure 1.7: A Petri net system with k processes that either read or write.

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.

1.1.6 Reachability Analysis

Having introduced the reachability set $R(N, M_0)$ in Sec. 1.1.5, we can now present a major analysis technique for 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.

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

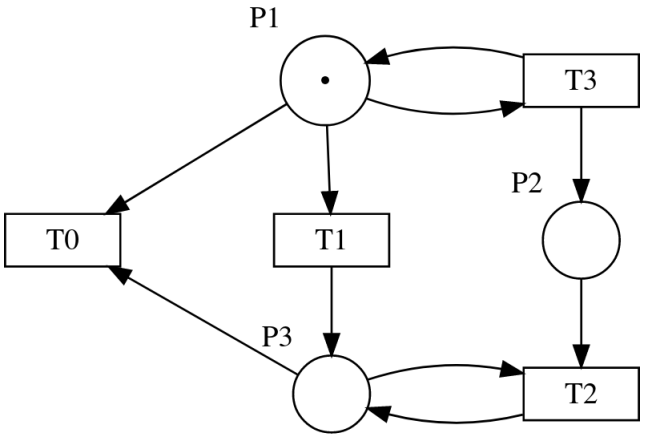

Figure 1.8: A marked Petri net for illustrating the construction of a reachability tree.

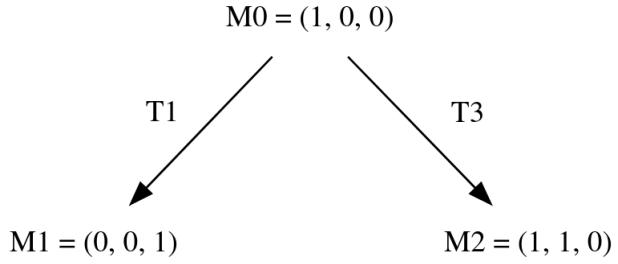

Figure 1.9: The first step building the reachability tree for the Petri net in Fig. 1.8.

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 can not 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 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

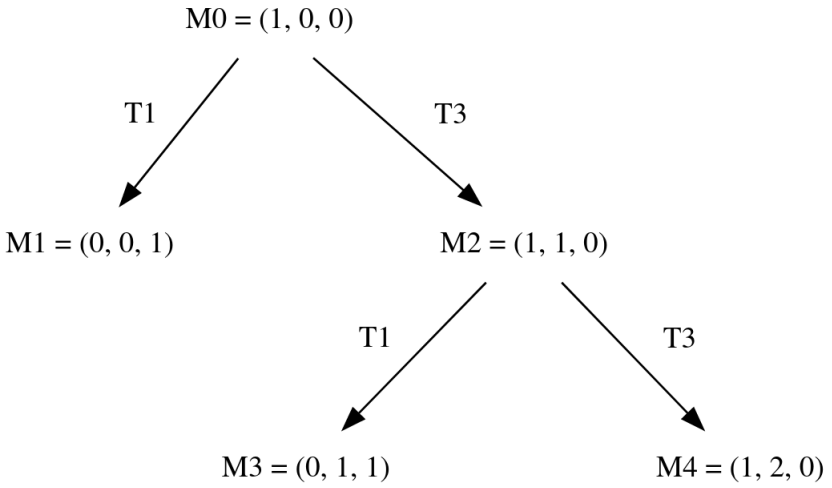

Figure 1.10: The second step building the reachability tree for the Petri net in Fig. 1.8.

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.

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

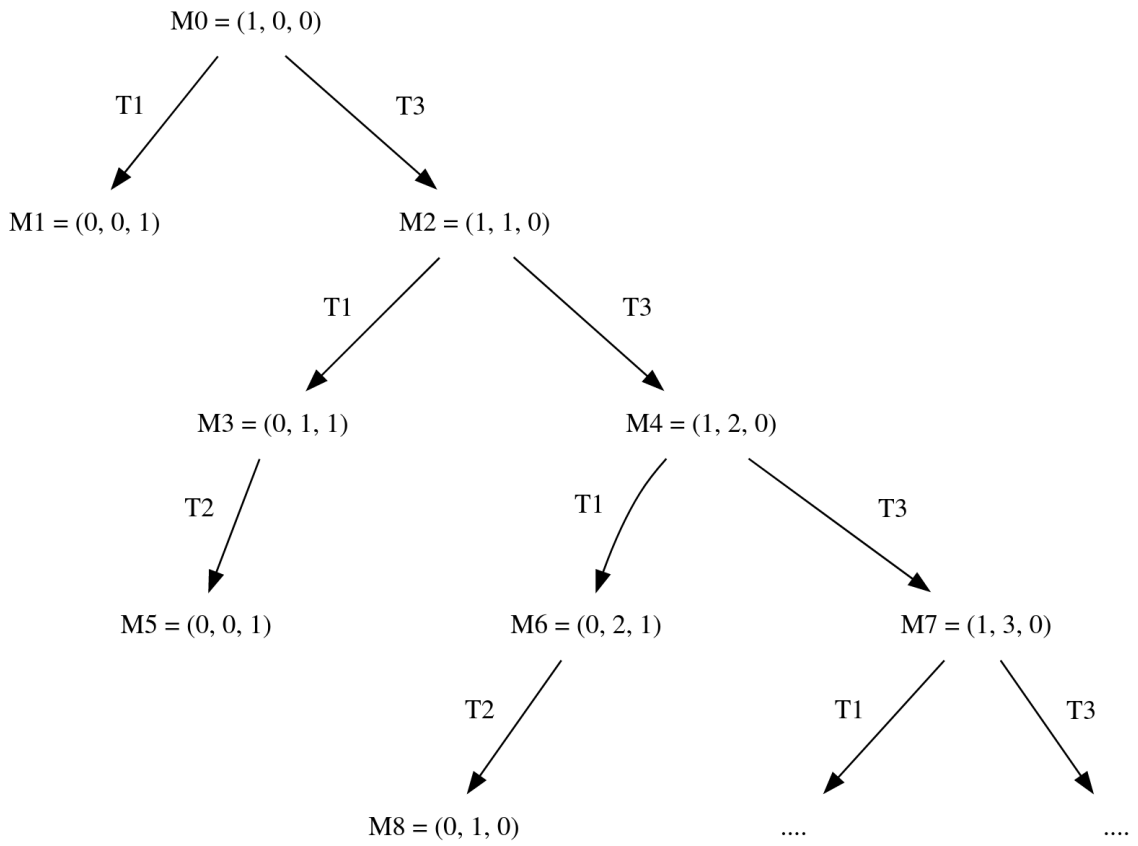

Figure 1.11: The infinite reachability tree for the Petri net in Fig. 1.8.

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.

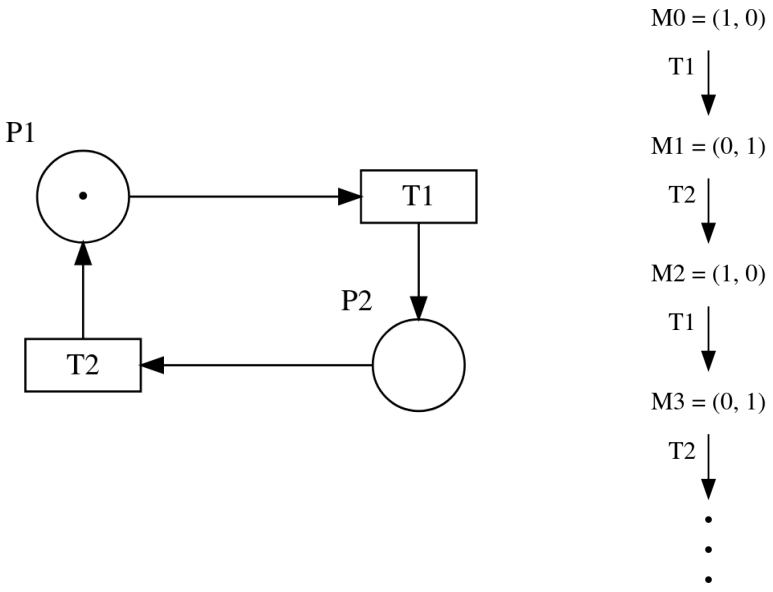

Figure 1.12: A simple Petri net with an infinite reachability tree.

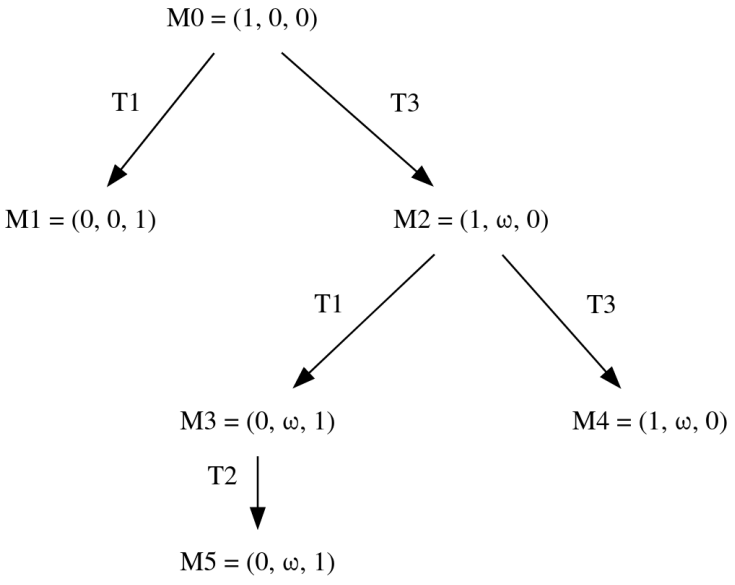

Figure 1.13: The finite reachability tree for the Petri net in Fig. 1.8.

1.2 The Rust programming language

One of the most promising modern programming languages for concurrent and memory-safe programming is Rust[1]. Rust is a multi-paradigm, general-purpose programming language that aims to provide developers with a safe, concurrent, and efficient way to write low-level code. It started as a project at Mozilla Research in 2009. The first stable release, Rust 1.0, was announced on May 15, 2015. For a brief history of Rust up to 2023, see [Thompson, 2023].

Rust's memory model based on the concept of *ownership* and its expressive type system prevent a wide variety of error classes related to memory management and concurrent programming at compile-time:

- Double free [Klabnik and Nichols, 2023, Cap. 4.1]
- Use after free [Klabnik and Nichols, 2023, Cap. 4.1]
- Dangling pointers [Klabnik and Nichols, 2023, Cap. 4.2]
- Data races [Klabnik and Nichols, 2023, Cap. 4.2]
- Passing non-thread-safe variables [Klabnik and Nichols, 2023, Cap. 16.4]

The official compiler *rustc*[2] takes care of controlling how the memory is used and allocating and deallocating objects. If a violation of its strict rules is found, the program will simply not compile.

In this section, we will justify the choice of Rust to study the detection of deadlocks and lost signals. We will show how these problems can be studied separately, knowing that other errors are already caught at compile time. In other words, we will argue that the stability and safety of the language provide a firm foundation on which to build a tool that detects additional errors during compilation.

1.2.1 Main characteristics

Some of Rust's main features are:

- Type system: Rust has a powerful type system that provides compile-time safety checks and prevents many common programming errors. It includes features such as type inference, generics, enums, and pattern matching. Every variable has a type but it is commonly inferred by the compiler.
- Performance: Rust's performance is comparable to C and C++, and it is often faster than many other popular programming languages such as Java, Go, Python or Javascript. Rust's performance is achieved through a combination of features such as zero-cost abstractions, minimal runtime, and efficient memory management.

[1] https://www.rust-lang.org/
[2] https://github.com/rust-lang/rust

- Concurrency: Rust has built-in support for concurrency. It supports several concurrency paradigms such as shared state, message passing and asynchronous programming. It does not force the developer to implement concurrency in a specific manner.
- Ownership and borrowing: Rust uses a unique ownership model to manage memory, allowing for efficient memory allocation and deallocation without the risk of memory leaks or data races. Furthermore, it does not rely on a garbage collector, thus saving resources. The *borrow checker* ensures that there is only one owner of a resource at any given time.
- Community-driven: Rust has a vibrant and growing community of developers who contribute to the language's development and ecosystem. Anyone can contribute to the language's development and suggest improvements. The documentation is also open-source and important decisions are documented in form of Requests for Comments (RFCs)[3].

The release cycle of the official Rust compiler, *rustc*, is remarkably fast. A new stable version of the compiler is released every 6 weeks [Klabnik and Nichols, 2023, Appendix G]. This is made possible by a complex automated testing system that compiles even all packages available on `crates.io`[4] using a program called *crater*[5] to verify that compiling and running the tests with the new version of the compiler does not break existing packages [Albini, 2019].

The borrow checker

Rust's borrow checker is a crucial component of its ownership model, which is designed to ensure memory safety and prevent data races in concurrent code. The borrow checker analyzes Rust code at compile-time and enforces a set of rules to ensure that a program's memory is accessed safely and efficiently.

The core idea behind the borrow checker is that each piece of memory in a Rust program has an owner. The owner may change during execution but there can only be one owner at any given time. Memory values can also be *borrowed*, that is, used without swapping the owner, similar to accessing the value through a pointer or a reference in other programming languages. When a value is borrowed, the borrower receives a reference to the value, but the original owner retains ownership. The borrow checker enforces rules to ensure that a borrowed value is not modified while it is borrowed and that the borrower releases the reference before the owner goes out of scope.

For clarity, we will now present some of the key rules enforced by the borrow checker:

- No two mutable references to the same memory location can exist simultaneously. This prevents data races, where two threads try to modify the same memory location at the same time.

[3] https://rust-lang.github.io/rfcs/
[4] https://crates.io/
[5] https://github.com/rust-lang/crater

- Mutable references can not exist at the same time as immutable references to the same memory location. This ensures that mutable and immutable references can not be used simultaneously, preventing inconsistent reads and writes.
- References can not outlive the value they reference. This ensures that references do not point to invalid memory locations, preventing null pointer dereferences and other memory errors.
- References can not be used after their owner has been moved or destroyed. This ensures that references do not point to memory that has been deallocated, preventing use-after-free errors.

It can take some effort to write Rust code that satisfies these rules. The borrow checker is usually singled out as one aspect of the language that is confusing for newcomers. However, this discipline pays off in terms of increased memory safety and performance. By ensuring that Rust programs follow these rules, the borrow checker eliminates many common programming errors that can lead to memory leaks, data races, and other bugs, while also teaching good coding practices and patterns.

1.2.2 Adoption

In this subsection, we will briefly describe the trend in the adoption of the Rust programming language. This highlights the relevance of this work as a contribution to a growing community of programmers who emphasize the importance of safe and performant systems programming for the next years in the software industry.

In the last few years, several major projects in the Open Source community and at private companies have decided to incorporate Rust to reduce the number of bugs related to memory management without sacrificing performance. Among them, we can name a few significant examples:

- The Android Open Source Project encourages the use of Rust for the SO components below the Android Runtime (ART) [Stoep and Hines, 2021].
- The Linux kernel, which introduces in version 6.1 (released in December 2022) official tooling support for programming components in Rust [Corbet, 2022, Simone, 2022].
- At Mozilla, the Oxidation project was created in 2015 to increase the usage of Rust in Firefox and related projects. As of March 2023, the lines of code in Rust represent more than 10% of the total in Firefox Nightly [Mozilla Wiki, 2015].
- At Meta, the use of Rust as a development language server-side is approved and encouraged since July 2022 [Garcia, 2022].
- At Cloudflare, a new HTTP proxy in Rust was built from scratch to overcome the architectural limitations of NGINX, reducing CPU usage by 70% and memory usage by 67% [Wu and Hauck, 2022].

- At Discord, reimplementing a crucial service written in Go in Rust provided great benefits in performance and solved a performance penalty due to the garbage collection in Go [Howarth, 2020].
- At npm Inc., the company behind the npm registry, Rust allowed scaling CPU-bound services to more than 1.3 billion downloads per day [The Rust Project Developers, 2019].

In other cases, Rust has proved to be a great choice in existing C/C++ projects to rewrite modules that process untrusted user input, for instance, parsers, and reduce the number of security vulnerabilities due to memory issues [Chifflier and Couprie, 2017].

Moreover, the interest of the developer community in Rust is undeniable, as it has been rated for 7 years in a row as the programming language most "loved" by programmers in the Stack Overflow Developer Survey [Stack Overflow, 2022].

1.2.3 Importance of memory safety

In this subsection, compelling evidence that the use of a memory-safe programming language is presented. The goal is to highlight the importance of advancing research in the compile-time detection of errors to prevent bugs that are subsequently difficult to correct in production systems.

Several empirical investigations have concluded that around 70% of the vulnerabilities found in large C/C++ projects occur due to memory handling errors. This high figure can be observed in projects such as:

- Android Open Source Project [Stepanov, 2020],
- the Bluetooth and media components of Android [Stoep and Zhang, 2020],
- the Chromium Projects behind the Chrome web browser [The Chromium Projects, 2015],
- the CSS component of Firefox [Hosfelt, 2019],
- iOS and macOS [Kehrer, 2019],
- Microsoft products [Miller, 2019, Fernandez, 2019],
- Ubuntu [Gaynor, 2020]

Numerous tools have set the goal to address these vulnerabilities caused by improper memory allocation in already established codebases. However, their use leads to a noticeable loss of performance and not all vulnerabilities can be prevented [Szekeres et al., 2013].

In [Jaeger and Levillain, 2014], the authors provide a detailed survey of programming language features that compromise the security of the resulting programs. They discuss the intrinsic security characteristics of programming languages and list recommendations for the education of developers or evaluators for secure software. Type safety is mentioned as one of the key elements for eliminating complete classes of bugs from the start. Another important point

is using a language where the specifications are as complete, explicit and formally defined as possible. The concept of Undefined Behavior (UB) should be included with caution and only sparingly. Examples from the C/C++ specification illustrate the confusion that follows from not following these principles. The authors conclude that memory safety achieved through garbage collection poses a threat to security and that other mechanisms should be considered instead.

We must note that Rust itself, like any other piece of software, is not exempt from security vulnerabilities. Serious bugs have been discovered in the standard library in the past [Davidoff, 2018]. Besides, code generation in Rust also includes mitigations to exploits of various kinds [Rust Project, 2023, Chap. 11]. However, this is far from the well-known issues in C and C++.

1.3 Correctness of concurrent programs

In the area of concurrent computing, one of the main challenges is to prove the correctness of a concurrent program. Unlike a sequential program where for each input the same output is always obtained, in a concurrent program the output may depend on how instructions from different processes or threads were interleaved during execution.

The correctness of a concurrent program is then defined in terms of the properties of the computation performed and not only in terms of the obtained result. In the literature [Ben-Ari, 2006, Coulouris et al., 2012, van Steen and Tanenbaum, 2017], two types of correctness properties are defined:

- **Safety properties**: The property must *always* be true.
- **Liveness properties**: The property must *eventually* become true.

Two desirable safety properties in a concurrent program are:

- **Mutual exclusion**: Two processes must not access shared resources at the same time.
- **Absence of deadlock**: A running system must be able to continue performing its task, that is, progressing and producing useful work.

Synchronization primitives such as mutexes, semaphores (as proposed by [Dijkstra, 2002]), monitors (as proposed by [Hansen, 1972, Hansen, 1973]) and condition variables (as proposed by [Hoare, 1974]) are usually used to implement coordinated access of threads or processes to shared resources. However, the correct use of these primitives is difficult to achieve in practice and can introduce errors that are difficult to detect and correct. Currently, most general-purpose languages, whether compiled or interpreted, do not allow these errors to be detected in all cases.

Given the increasing importance of concurrent programming due to the proliferation of multithreaded and multithreaded hardware systems, reducing the number of bugs linked to the

synchronization of threads or processes is extremely important for the industry. Deadlock-free operation is an unavoidable requirement for many projects, such as operating systems [Arpaci-Dusseau and Arpaci-Dusseau, 2018], autonomous vehicles [Perronnet et al., 2019] and aircraft [Carreño and Muñoz, 2005, Monzon and Fernandez-Sanchez, 2009].

In the next section, we will have a closer look at the conditions that cause a deadlock and the strategies used to cope with them.

1.4 Deadlocks

Deadlocks are a common problem that can occur in concurrent systems, which are systems where multiple threads or processes are running simultaneously and potentially sharing resources. They have been studied at least since [Dijkstra, 1964], who coined the term "deadly embrace" in Dutch, which did not catch on.

A deadlock occurs when two or more threads or processes are blocked and unable to continue executing because each is waiting for the other to release a resource that it needs. This results in a situation where none of the threads or processes can make progress and the system becomes effectively stuck. An alternative equivalent definition of deadlocks in terms of program states can be found in [Holt, 1972].

Deadlocks can be a serious problem in concurrent systems, as they can cause the system to become unresponsive or even crash. Therefore, it is important to be able to detect and prevent deadlocks. They can occur in any concurrent system where multiple threads or processes are competing for shared resources. Examples of shared resources that can lead to deadlocks include system memory, input/output devices, locks, and other types of synchronization primitives.

Deadlocks can be difficult to detect and prevent because they depend on the precise timing of events in the system. Even in cases where deadlocks can be detected, resolving them can be difficult, as it may require releasing resources that have already been acquired or rolling back completed transactions. To avoid deadlocks, it is important to carefully manage shared resources in a concurrent system. This can involve using techniques such as resource allocation algorithms, deadlock detection algorithms, and other types of synchronization primitives. By carefully managing shared resources, it is possible to prevent deadlocks from occurring and ensure the smooth operation of concurrent systems.

To understand the concept in more detail, consider a simple example where two processes, A and B, are competing for two resources, X and Y. Initially, process A has acquired resource X and is waiting to acquire resource Y, while process B has acquired resource Y and is waiting to acquire resource X. In this situation, neither process can continue executing because it is waiting for the other process to release a resource that it needs. This results in a deadlock, as neither process can make progress. Fig. 1.14 illustrates this situation. The cycle therein indicates a deadlock, as will be explained in the next section.

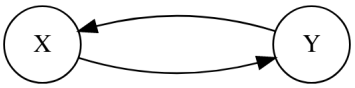

Figure 1.14: Example of a state graph with a cycle indicating a deadlock.

1.4.1 Necessary conditions

According to the classic paper on the topic [Coffman et al., 1971], the following conditions need to hold for a deadlock to arise. They are sometimes called "Coffman conditions".

1. **Mutual Exclusion**: At least one resource in the system must be held in a non-sharable mode, meaning that only one thread or process can use it at a time (e.g. a variable behind a mutex).
2. **Hold and Wait**: At least one thread or process in the system must be holding a resource and waiting to acquire additional resources that are currently being held by other threads or processes.
3. **No Preemption**: Resources cannot be preempted, which means that a thread or process holding a resource cannot be forced to release it until it has completed its task.
4. **Circular Wait**: There must be a circular chain of two or more threads or processes, where each thread or process is waiting for a resource held by the next one in the chain. This is usually visualized in a graph representing the order in which the resources are acquired.

Usually, the first three conditions are characteristics of the system under study, i.e. the protocols used for acquiring and releasing resources, while the fourth may or may not occur depending on the interleaving of instructions during the execution.

It is worth noting that the Coffman conditions are in general necessary but not sufficient for a deadlock to occur. The conditions are indeed sufficient in the case of single-instance resource systems. But they only indicate the possibility of deadlock in systems where there are multiple indistinguishable instances of the same resource.

In the general case, if any one of the conditions is not met, a deadlock cannot occur, but the presence of all four conditions does not necessarily guarantee a deadlock. Nonetheless, the Coffman conditions are an important tool for understanding and analyzing the causes of deadlocks in concurrent systems and they can help guide the development of strategies for preventing and resolving deadlocks.

1.4.2 Strategies

Several strategies for handling deadlocks exist, each of which has its strengths and weaknesses. In practice, the most effective strategy will depend on the specific requirements and constraints

of the system being developed. Designers and developers must carefully consider the trade-offs between different strategies and choose the approach that is best suited to their needs. Interested readers are referred to [Coffman et al., 1971, Singhal, 1989].

Prevention

One way to deal with deadlocks is to prevent them from occurring in the first place. The idea is for deadlocks to be excluded a priori. With this objective in mind, we must ensure that at every point in time at least one of the necessary conditions developed in Sec. 1.4.1 is not satisfied. This restricts the possible protocols in which requests for resources may be made. We will now look at each condition in isolation and elaborate on the most common approaches.

If the first condition must be false, then the program should allow shared access to all resources. Lock-free synchronization algorithms may be used for this purpose since they do not implement mutual exclusion. This is difficult to achieve in practice for all resource types, since for example a file may not be shared by more than one thread or process during an update of the file contents.

Looking at the second condition, a feasible approach would be to impose that each thread or process acquires all the required resources at once and that the thread or process can not proceed until access to all of them has been granted. This all-or-nothing policy causes a significant performance penalty, given that resources may be allocated to a specific thread or process but may remain unused for long periods. In simpler terms, it decreases concurrency.

If the no preemption condition is denied, then resources may be recovered in certain circumstances, e.g. using resource allocation algorithms that ensure that resources are never held indefinitely. After a timeout or when a condition is satisfied, the thread or process releases the resource or a supervisor process recovers the resource forcibly. Usually, this works well when the state of the resource can be easily saved and restored later. One example of this is the allocation of CPU cores in a modern operating system (OS). The scheduler allocates one processor core to one task and may switch to a different task or move the task to a new processor core at any moment just by saving the contents of the registers [Arpaci-Dusseau and Arpaci-Dusseau, 2018, Chapter 6]. However, if preserving the resource state is not possible, preemption may entail a loss of the progress done so far, which is not acceptable in many scenarios.

Lastly, if the state graph of the resources never forms a cycle, then the fourth necessary condition is false and deadlocks are prevented. To achieve this one could introduce a linear ordering of resource types. In other words, if a process or thread has been allocated resources of type r_i, it may subsequently require only those resources of types that follow r_i in the ordering. This involves using special synchronization primitives that allow resources to be shared in a controlled manner and enforcing strict rules for resource acquisition and release. Under these conditions, the state graph will be strictly speaking a forest (an acyclical graph) and no deadlocks are possible.

In practical applications, a combination of the previous strategies may prove useful when none

of them is entirely applicable.

Avoidance

Avoidance is another strategy for dealing with deadlocks, which involves dynamically detecting and avoiding potential deadlocks *before* they occur. For this, the system requires global knowledge in advance regarding which resources a thread or process will request during its lifetime. Note that, in linguistic terms, "deadlock avoidance" and "deadlock prevention" may seem similar, but in the context of deadlock handling, they are distinct concepts.

One of the classic deadlock avoidance algorithms is the Banker's algorithm [Dijkstra, 1964]. Another relevant algorithm is proposed by [Habermann, 1969].

Regrettably, these techniques are only effective in highly specific scenarios, such as in an embedded system where the complete set of tasks to be executed and their required locks are known a priori. Consequently, deadlock avoidance is not a commonly used solution applicable to a broad range of situations.

Detection and Recovery

Another strategy to handle deadlocks is to detect them *after* they occur and recover from them. For a survey of algorithms for deadlock detection in distributed systems, see [Singhal, 1989]. We will briefly present the general idea behind one of them for illustration purposes.

The Resource Allocation Graph (RAG) is a commonly used method for detecting deadlocks in concurrent systems. It represents the relationship between threads/processes and resources in the system as a directed graph. Each process and resource is represented by a node in the graph and a directed edge is drawn from a process to a resource if the process is currently holding that resource. This is analogous to the state graph shown in Fig. 1.14 but with the threads/processes represented in the diagram. The state graph may also be applied to deadlock detection [Coffman et al., 1971].

To detect deadlocks using the RAG, we need to look for cycles in the graph. If there is a cycle in the graph, it indicates that a set of processes is waiting for resources that are currently being held by other processes in the cycle. Therefore no process in the cycle can make progress.

The recovery part of the process involves terminating one of the threads or processes in the cycle. This causes the resources to be released and the other threads or processes are allowed to continue.

Database management systems (DBMS) incorporate subsystems for detecting and resolving deadlocks. A deadlock detector is executed at intervals, generating a regular allocation graph, otherwise called the transaction-wait-for (TWF) graph, and examining it for any cycles. If a cycle (deadlock) is identified, the system must be restarted. An excellent overview of deadlock detection in distributed database systems is [Knapp, 1987]. The subject of concurrency control and recovery from deadlocks in DBMS is extensively discussed in [Bernstein et al., 1987].

Acceptance or ignoring deadlocks altogether

In some cases, it may be admissible to simply accept the risk of deadlocks and manage them as they occur. This approach may be appropriate in systems where the cost of preventing or detecting deadlocks is too high, or where the frequency of deadlocks is low enough that the impact on system performance is minimal, or where the data loss incurred each time is tolerable.

UNIX is an example of an OS following this principle [Shibu, 2016, p. 477]. Other major operating systems also exhibit this behavior. On the other hand, a life-critical system cannot afford to pretend its operation will be deadlock-free for any reason.

1.5 Condition variables

Condition variables are a synchronization primitive in concurrent programming that allows threads to efficiently wait for a specific condition to be met before proceeding. They were first introduced by [Hoare, 1974] as part of a building block for the concept of monitor developed originally by [Hansen, 1973].

Following the classic definition, two main operations can be called on a condition variable:

- `wait`: Blocks the current thread or process. In some implementations, the associated mutex is released as part of the operation.
- `signal`: Wakes up one thread or process waiting on the condition variable. In some implementations, the associated mutex lock is immediately acquired by the signaled thread or process.

Condition variables are typically associated with a boolean predicate (a condition) and a mutex. The boolean predicate is the condition on which the threads or processes are waiting for. When it is set to a particular value (either true or false), the thread or process should continue executing. The mutex ensures that only one thread or process may access the condition variable at a time.

Condition variables do not contain an actual value accessible to the programmer inside of them. Instead, they are implemented using a queue data structure, where threads or processes are added to the queue when they enter the wait state. When another thread or process signals the condition, an element from the queue is selected to resume execution. The specific scheduling policy may vary depending on the implementation.

Over the years, various implementations and optimizations have been developed for condition variables to improve performance and reduce overhead. For example, some implementations allow multiple threads to be awakened at once (an operation called *broadcast*), while others use a priority queue to ensure that the most important threads are awakened first.

Condition variables are part of the POSIX standard library for threads [Nichols et al., 1996], and they are now widely used in concurrent programming languages and systems. They are

found among others in:

- UNIX[6],
- Rust[7]
- Python[8]
- Go[9]
- Java[10]

Despite their widespread use, condition variables can be tricky to use correctly, and incorrect use can lead to subtle and hard-to-debug errors such as missed signals or spurious wakeups. We will look now at these errors in detail.

1.5.1 Missed signals

A missed signal occurs when a thread or process waiting on a condition variable fails to receive a signal even though it has been emitted. This can happen due to a race condition, where the signal is emitted before the thread enters the wait state, causing the signal to be missed.

To illustrate the concept of a missed signal, we will look at an example. Suppose we have two threads, T1 and T2, and a shared integer variable called `flag`. T1 sets `flag` to `true` and signals a condition variable `cv` to wake up T2, which is waiting on `cv` to know when `flag` has been set. T2 waits on `cv` until it receives a signal from T1. Listing 1.1 shows the corresponding pseudocode.

```
// T1
lock.acquire()
flag = true
cv.signal()   // Signal T2 to wake up
lock.release()

// T2
lock.acquire()
while (flag == false)   // Wait until flag has changed
    cv.wait(lock)
lock.release()
```

Listing 1.1: Pseudocode for a missed signal example

Now, suppose that T1 sets `flag` and signals `cv`, but T2 has not yet entered the wait state on `cv` due to some scheduling delay. In this case, the signal emitted by T1 could be missed by T2, as shown in the following sequence of events:

[6] https://man7.org/linux/man-pages/man3/pthread_cond_init.3p.html
[7] https://doc.rust-lang.org/std/sync/struct.Condvar.html
[8] https://docs.python.org/3/library/threading.html
[9] https://pkg.go.dev/sync
[10] Condition Interface

1. T1 acquires the lock and sets `flag` to `true`.
2. T1 signals `cv` to wake up T2.
3. T1 releases the lock.
4. T2 acquires the lock and checks if `flag` has changed. Since `flag` is still `false`, T2 enters the wait state on `cv`.
5. Due to scheduling delays or other factors, T2 does not receive the signal emitted by T1 and remains stuck in the wait state forever.

This scenario illustrates the concept of a missed signal, where a thread waiting on a condition variable fails to receive a signal even though it has been emitted. To prevent missed signals, it is important to ensure that threads waiting on condition variables are properly synchronized with the threads emitting signals and that there are no race conditions or timing issues that could cause signals to be missed.

1.5.2 Spurious wakeups

A spurious wakeup happens when a thread waiting on a condition variable wakes up without receiving a signal or notification from another thread. Reasons for this are multiple: hardware or operating system interrupts, internal implementation details of the condition variable or other unpredictable factors.

Reusing the situation described in the previous section and the pseudocode shown in Listing 1.1, suppose now that T1 sets `flag` to `true` and signals `cv`, but T2 wakes up without receiving the signal emitted by T1.

This is precisely the spurious wakeup. The following sequence of events leads to this unfortunate outcome:

1. T1 acquires the lock and sets `flag` to `true`.
2. T1 signals `cv` to wake up T2.
3. T1 releases the lock.
4. T2 acquires the lock and checks if `flag` is `true`. Since `flag` is still `false`, T2 enters the wait state on `cv`.
5. Due to some internal implementation detail of the condition variable or other unpredictable factors, T2 wakes up without receiving the signal emitted by T1 and continues executing the next statement in its code.

This example demonstrates the idea of a spurious wakeup, in which a thread waiting on a condition variable wakes up without receiving a signal or notification from another thread. To prevent spurious wakeups, it is important to use a loop to recheck the condition after waking up from a wait state, as shown in the pseudocode for T2 (line 9). This ensures that the thread

does not proceed until the condition it is waiting for has indeed occurred. If the while loop were not there, a spurious wakeup would cause T2 to continue executing after the call to `wait`, regardless of whether a signal was emitted by T1 or not.

1.6 Compiler arquitecture

Compilers are programs that transform source code written in one language into another language, usually machine code. A compiler takes in a program in one language, the *source* language, and translates it into an equivalent program in another language, the *target* language.

To achieve this, compilers typically have a series of phases or passes that are executed in sequence. The goal of these passes is to translate the high-level code into low-level code that the machine can execute. In each pass, the code is brought closer and closer to the final representation. These phases are nowadays well-defined and different compilers implement some form of them [Aho et al., 2014, Chapter 1.2].

The first pass of a typical compiler is the **lexical analysis** phase. In this phase, the source code is broken down into a stream of tokens, each of which represents a single piece of the code. The *lexer* identifies keywords, identifiers, literals and other tokens that form the building blocks of the source code.

The next pass is the **syntax analysis** phase, also known as the parser phase. In this phase, the tokens produced by the lexer are analyzed according to the rules of the programming language's grammar. The *parser* constructs a parse tree or an abstract syntax tree (AST) that represents the structure of the code.

The third pass is the **semantic analysis** phase, in which the compiler checks the code for semantic correctness, such as checking for type errors, undefined variables, and invalid operations. The *semantic analyzer* builds a symbol table that contains information about the variables, functions, and other entities defined in the code.

The fourth pass is the **code generation** phase. The compiler takes the AST and symbol table produced by the previous phases and generates low-level code that can be executed by the machine. The code generator typically generates code in assembly language or machine code. In other cases, it generates bytecode, as in Java or when using the Python just-in-time (JIT) compiler.

Finally, there may be zero or more **code optimization** phases. These are from a theoretical point of view optional, but they are usually included by default in modern compilers. In this phase, the compiler analyzes the generated code and attempts to improve its efficiency by applying various optimization techniques. Some examples of optimizations include:

- constant folding [Aho et al., 2014, Chapter 8.5.4],
- loop unrolling [Aho et al., 2014, Chapter 10.5],

- register allocation [Aho et al., 2014, Chapter 8.1.4],
- constant propagation [Aho et al., 2014, Chapter 9],
- liveness analysis [Aho et al., 2014, Chapter 9],
- and many more...

Local code optimizations concern improvements within a basic block, whereas *global* code optimization is when improvements take into account what happens across basic blocks. In Rust, one example of global optimization is link time optimization (LTO) [Huss, 2020].

Fig. 1.15 taken from [Aho et al., 2014] summarizes the compiler phases described in this section.

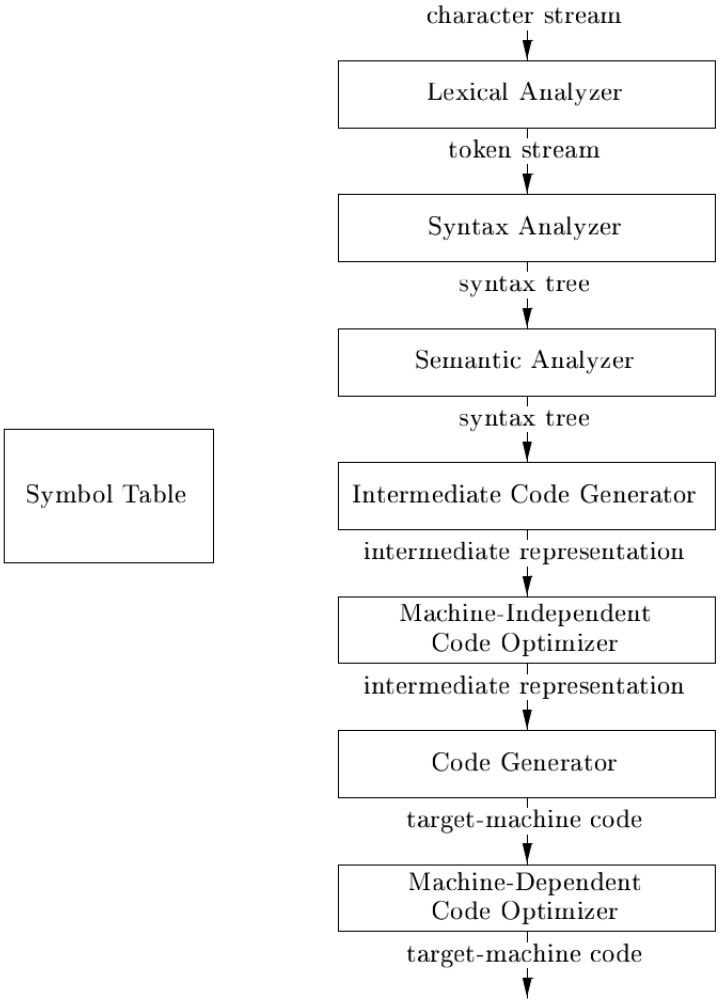

Figure 1.15: Phases of a compiler

In practice, phases might have unclear boundaries. They can overlap and some may be skipped entirely. In later sections, we will study the architecture of the Rust compiler *rustc* and explain

its general architecture.

1.7 Model checking

Model checking is a technique used in software development to formally verify the correctness of a system's behavior with respect to its specifications or requirements. It involves constructing a mathematical model of the system and analyzing it to ensure that it meets certain properties, such as mutual exclusion when accessing shared resources, absence of data races and deadlock-freedom.

The process of model checking begins by constructing a finite-state model of the system, typically using a formal language, in the case of this work the language of Petri nets. The model captures the system's behavior and the properties that are to be verified. The next step is to perform an exhaustive search of the state space of the model to ensure that all possible behaviors have been considered. This search can be performed automatically using specialized software tools.

During the search, the model checker looks for counterexamples, which are sequences of events that violate the system's specifications. If a counterexample is found, the model checker provides information on the state of the system at the time of the violation, helping developers to identify and fix the problem.

Model checking has become an important technique in the development of critical software systems, such as aerospace [Carreño and Muñoz, 2005, Monzon and Fernandez-Sanchez, 2009] and automotive control systems [Perronnet et al., 2019], medical devices, and financial systems. By verifying the correctness of the software before it is deployed, developers can ensure that the system meets its requirements and is safe to use.

One of the main advantages of model checking is that it provides a formal and rigorous approach to verifying software correctness. Unlike traditional testing methods, which can only demonstrate the presence of errors, model checking can prove the absence of errors. This is particularly important for safety-critical systems like the ones mentioned before, where a single error can have catastrophic consequences for human lives. Model checking can also be automated, allowing developers to quickly and efficiently verify the correctness of complex software systems. This reduces the time and cost of software development and increases confidence in the correctness of the system.

It is known that formal software verification tools are currently applied in a few very specific fields where formal proof of the correctness of the system is required. [Reid et al., 2020] discusses the importance of bringing verification tools closer to developers through an approach that seeks to maximize the cost-benefit ratio of its use. Improvements in the usability of existing tools and approaches to incorporate their use into the developer's routine are presented. The paper starts from the premise that from the developer's point of view, verification can be seen as a different type of unit or integration test. Therefore, it is of utmost importance that

running the verification is as easy as possible and feedback is provided to the developer quickly during the development process to increase adoption.

In this work, we develop a formal verification tool for the Rust programming language that is able to detect some classes of deadlocks and missed signals in the source code at compile time. This tool can help developers quickly verify that their use of synchronization primitives is correct and will not cause hard-to-detect bugs later on. Our goal is to make the tool user-friendly and easy to get started with, so that its adoption benefits the larger community of Rust developers.

In the next two sections, we will look at the existing tools, their scope and their goals compared to the tool we developed.

1.7.1 Formal verification of Rust code

There are numerous automatic verification tools available for Rust code. A recommended first approximation to the topic is the survey produced by Alastair Reid, a researcher at Intel. It explicitly lists that most formal verification tools do not support concurrency [Reid, 2021].

The *Miri*[11] interpreter developed by the Rust project on GitHub is an experimental interpreter for the intermediate representation of the Rust language (Mid-level Intermediate Representation (MIR)) that allows executing standard cargo project binaries in a granularized way, instruction by instruction, to check for the absence of Undefined Behavior (UB) and other errors in memory handling. It detects memory leaks, unaligned memory accesses, data races, and precondition or invariant violations in code marked as `unsafe`.

[Toman et al., 2015] introduces a formal checker for Rust that does not require modifications to the source code. It was tested on past versions of modules from the Rust standard library. As a result, errors were detected in the use of memory in unsafe Rust code which in reality took months to be discovered manually by the development team. This exemplifies the importance of using automatic verification tools to complement manual code reviews.

1.7.2 Deadlock detection using Petri nets

Deadlock prevention is one of the classic strategies to address this crucial problem in concurrent programming, as discussed in Sec. 1.4.2. The main problem with the approach of detecting deadlocks before they occur is proving that the desired type of deadlock is detected in all cases and that no false negatives are produced in the process. The Petri net-based approach, being a formal method, satisfies these conditions. However, the difficulty of adoption lies mainly in the practicability of the solution due to the large number of possible states in a real software project.

In [Karatkevich and Grobelna, 2014], a method is proposed to reduce the number of explored states during the detection of deadlocks using reachability analysis. These heuristics help

[11] https://github.com/rust-lang/miri

improve the performance of the Petri net-based approach. Another optimization is presented in [Küngas, 2005]. The author proposes a very promising polynomial order method to avoid the problem of the state explosion that underlies the naïve deadlock detection algorithm. Through an algorithm that abstracts a given Petri net to a simpler representation, a hierarchy of networks of increasing size is obtained for which the verification of the absence of deadlocks is substantially faster. It is, crudely put, a "divide and conquer" strategy that checks for the absence of deadlocks in parts of the network to later build the verification of the final whole by adding parts to the initial small network.

Despite the previously mentioned caveats, the use of Petri nets as a formal software verification method has been established since the late 1980s. Petri nets allow for intuitive modeling of synchronization primitives, such as sending a message or waiting for the reception of a message. Examples of these simple nets with correspondingly simple behavior are found in [Heiner, 1992]. These nets are construction blocks that can be combined to form a more complex system.

To put these models to use, there are two possibilities:

- One is designing the system in terms of Petri nets and then translating the Petri nets to the source code.
- The other one is to translate the existing source code to a Petri net representation and then verify that the Petri net model satisfies the desired properties.

For the purposes of this work, we are interested in the latter. This approach is not novel. It has been implemented for other programming languages like C and Rust already, as seen in the literature.

In [Kavi et al., 2002] and [Moshtaghi, 2001], a translation of some synchronization primitives available as part of the POSIX library of threads (`pthread`) in C to Petri nets is described. In particular, the translation supports:

- The creation of threads with the function `pthread_create` and the handling of the variable of type `pthread_t`.
- The thread join operation with the `pthread_join` function.
- The operation of acquiring a mutex with `pthread_mutex_lock` and its eventual manual release with `pthread_mutex_unlock`.
- The `pthread_cond_wait` and `pthread_cond_signal` functions for working with condition variables.

In his master's thesis, [Meyer, 2020] establishes the bases for a Petri net semantic for the Rust programming language. He focuses his efforts however on single-threaded code, limiting himself to the detection of deadlocks caused by executing the `lock` operation twice on the same mutex in the main thread. Unfortunately, the code available as part of the thesis is no longer valid for the new version of the *rustc* compiler, since the internals of the compiler changed significantly in the last two years.

1.8 Survey of existing libraries

As part of the development of the translation of the source code to a Petri net, it will be necessary to use a Petri net library for the Rust programming language. A quick search of the packages available on *crates.io*[12], GitHub and GitLab revealed that there is unfortunately no well-maintained library.

Some Petri net simulators were found such as:

- pns[13]: Programmed in C. It does not offer the option to export the resulting network to a standard format.
- PetriSim[14]: An old DOS/PC simulator programmed in Bordland Pascal.
- WOLFGANG[15]: A Petri net editor in Java, maintained by the Department of Computer Science at the University of Freiburg, Germany.

Regrettably, none of them meet the requirements of the job.

Since a Petri net is a graph, we considered the possibility of using a graph library and adapting it to the objectives of this work. Two graph libraries were found in Rust:

- petgraph[16]: The most widely used library for graphs in *crates.io*. It offers an option to export to the DOT format.
- gamma[17]: Unstable and unchanged since 2021. It does not offer the ability to export the graph.

None of the possibilities satisfies the requirement to export the resulting network to the PNML format. In addition, if a graph library is used, the operations of a Petri net should be implemented as a *wrapper* around a graph, which reduces the possibility of optimizations for our use case and hinders the long-term extensibility of the project.

In conclusion, it is imperative to implement a Petri net library in Rust from scratch as a separate project. This contributes one more tool to the community that could be reused in the future.

[12] https://crates.io/
[13] https://gitlab.com/porky11/pns
[14] https://staff.um.edu.mt/jskl1/petrisim/index.html
[15] https://github.com/iig-uni-freiburg/WOLFGANG
[16] https://docs.rs/petgraph/latest/petgraph/
[17] https://github.com/metamolecular/gamma

Chapter 2

Design of the proposed solution

Chapter 3

Testing the implementation

3.1 Unit tests

3.2 Integration tests

3.3 Generating the MIR representation

3.4 Visualizing the result

[Gansner et al., 2015] [Hillah and Petrucci, 2010] [Jüngel et al., 2000]

Chapter 4

Conclusions

Chapter 5

Future work

Chapter 6

Related work

In [Rawson and Rawson, 2022], the authors propose a generalized model based on colored Petri nets and implement an open-source middleware framework in Rust[1] to build, design, simulate and analyze the resulting Petri nets.

Colored Petri nets (CPN) are a type of Petri net that can represent more complex systems than traditional Petri nets. In a CPN, tokens have a specific value associated with them, which can represent various attributes or properties of the system being modeled. This allows for more detailed and accurate modeling of real-world systems, including those with complex data structures and behaviors. In the visual representation, each token has a color (analogous to a type in programming languages) and the transitions expect tokens from a particular color (type) and can generate tokens of the same color or tokens of a different color. As a short example, consider a transition with two input places and one output place representing the mixing of primary colors. If the input token colors are red and blue, then the output token color is purple. If the input token colors are yellow and blue, then the output token color is green.

The model proposed by the authors is an even more general type of Petri net, named Nondeterministic Transitioning Petri nets (NT-Petri nets), which allows transitions to fire without having all their input places marked with tokens, while also allowing each transition to define which output places should be marked depending on the input. In other words, each transition defines arbitrary rules for its firing to take place. They explain briefly how the Petri net could be analyzed to solve for the maximal number of useful threads to execute the task modeled therein. They also mention the modeling step as a tool for checking for erroneous states before deploying an electronic or computer system.

[1] https://github.com/MarshallRawson/nt-petri-net

Bibliography

[Aho et al., 2014] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2014). *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2 edition.

[Albini, 2019] Albini, P. (2019). RustFest Barcelona - Shipping a stable compiler every six weeks. https://www.youtube.com/watch?v=As1gXp5kX1M. Accessed on 2023-02-24.

[Arpaci-Dusseau and Arpaci-Dusseau, 2018] Arpaci-Dusseau, R. H. and Arpaci-Dusseau, A. C. (2018). *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 edition. https://pages.cs.wisc.edu/~remzi/OSTEP/.

[Ben-Ari, 2006] Ben-Ari, M. (2006). *Principles of Concurrent and Distributed Programming*. Pearson Education, 2nd edition.

[Bernstein et al., 1987] Bernstein, P. A., Hadzilacos, V., Goodman, N., et al. (1987). *Concurrency control and recovery in database systems*, volume 370. Addison-Wesley Reading.

[Carreño and Muñoz, 2005] Carreño, V. and Muñoz, C. (2005). Safety verification of the small aircraft transportation system concept of operations. In *AIAA 5th ATIO and16th Lighter-Than-Air Sys Tech. and Balloon Systems Conferences*, page 7423.

[Chifflier and Couprie, 2017] Chifflier, P. and Couprie, G. (2017). Writing parsers like it is 2017. In *2017 IEEE Security and Privacy Workshops (SPW)*, pages 80–92. IEEE.

[Coffman et al., 1971] Coffman, E. G., Elphick, M., and Shoshani, A. (1971). System deadlocks. *ACM Computing Surveys (CSUR)*, 3(2):67–78.

[Corbet, 2022] Corbet, J. (2022). The 6.1 kernel is out. https://lwn.net/Articles/917504/. Accessed on 2023-02-24.

[Coulouris et al., 2012] Coulouris, G., Dollimore, J., Kindberg, T., and Blair, G. (2012). *Distributed Systems, Concepts and Design*. Pearson Education, 5th edition.

[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.

[Davidoff, 2018] Davidoff, S. (2018). How Rust's standard library was vulnerable for years and nobody noticed. https://shnatsel.medium.com/

how-rusts-standard-library-was-vulnerable-for-years-and-nobody-noticed-aebf0503c3d6. Accessed on 2023-02-20.

[Dijkstra, 1964] Dijkstra, E. W. (1964). Een algorithme ter voorkoming van de dodelijke omarming. http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD108.PDF.

[Dijkstra, 2002] Dijkstra, E. W. (2002). *Cooperating Sequential Processes*, pages 65–138. Springer New York, New York, NY.

[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.

[Fernandez, 2019] Fernandez, S. (2019). A proactive approach to more secure code. https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/. Accessed on 2023-02-24.

[Gansner et al., 2015] Gansner, E. R., Koutsofios, E., and North, S. C. (2015). *Drawing Graphs With Dot*.

[Garcia, 2022] Garcia, E. (2022). Programming languages endorsed for server-side use at Meta. https://engineering.fb.com/2022/07/27/developer-tools/programming-languages-endorsed-for-server-side-use-at-meta/. Accessed on 2023-02-24.

[Gaynor, 2020] Gaynor, A. (2020). What science can tell us about C and C++'s security. https://alexgaynor.net/2020/may/27/science-on-memory-unsafety-and-security/. Accessed on 2023-02-24.

[Habermann, 1969] Habermann, A. N. (1969). Prevention of system deadlocks. *Communications of the ACM*, 12(7):373–ff.

[Hansen, 1972] Hansen, P. B. (1972). Structured multiprogramming. *Communications of the ACM*, 15(7):574–578.

[Hansen, 1973] Hansen, P. B. (1973). *Operating system principles*. Prentice-Hall, Inc.

[Heiner, 1992] Heiner, M. (1992). Petri net based software validation. *International Computer Science Institute ICSI TR-92-022, Berkeley, California*.

[Hillah and Petrucci, 2010] Hillah, L. M. and Petrucci, L. (2010). Standardisation des réseaux de Petri : état de l'art et enjeux futurs. *Génie logiciel : le magazine de l'ingénierie du logiciel et des systèmes*, 93:5–10.

[Hoare, 1974] Hoare, C. A. R. (1974). Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557.

[Holt, 1972] Holt, R. C. (1972). Some deadlock properties of computer systems. *ACM Computing Surveys (CSUR)*, 4(3):179–196.

[Hosfelt, 2019] Hosfelt, D. (2019). Implications of Rewriting a Browser Component in Rust. https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/. Accessed on 2023-02-24.

[Howarth, 2020] Howarth, J. (2020). Why Discord is switching from Go to Rust. https://discord.com/blog/why-discord-is-switching-from-go-to-rust. Accessed on 2023-03-20.

[Huss, 2020] Huss, E. (2020). Disk space and LTO improvements. https://blog.rust-lang.org/inside-rust/2020/06/29/lto-improvements.html. Accessed on 2023-04-06.

[Jaeger and Levillain, 2014] Jaeger, E. and Levillain, O. (2014). Mind your language (s): A discussion about languages and security. In *2014 IEEE Security and Privacy Workshops*, pages 140–151. IEEE.

[Jüngel et al., 2000] Jüngel, M., Kindler, E., and Weber, M. (2000). The petri net markup language. *Petri Net Newsletter*, 59(24-29):103–104.

[Karatkevich and Grobelna, 2014] Karatkevich, A. and Grobelna, I. (2014). Deadlock detection in petri nets: one trace for one deadlock? In *2014 7th International Conference on Human System Interactions (HSI)*, pages 227–231. IEEE.

[Kavi et al., 2002] Kavi, K. M., Moshtaghi, A., and Chen, D.-J. (2002). Modeling multithreaded applications using petri nets. *International Journal of Parallel Programming*, 30:353–371.

[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.

[Kehrer, 2019] Kehrer, P. (2019). Memory Unsafety in Apple's Operating Systems. https://langui.sh/2019/07/23/apple-memory-safety/. Accessed on 2023-02-24.

[Klabnik and Nichols, 2023] Klabnik, S. and Nichols, C. (2023). *The Rust programming language*. No Starch Press. https://doc.rust-lang.org/stable/book/.

[Knapp, 1987] Knapp, E. (1987). Deadlock detection in distributed databases. *ACM Computing Surveys (CSUR)*, 19(4):303–328.

[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. PDF available from public profile on ResearchGate.

[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.

[Meyer, 2020] Meyer, T. (2020). A Petri-Net semantics for Rust. Master's thesis, Universität Rostock — Fakultät für Informatik und Elektrotechnik.

[Miller, 2019] Miller, M. (2019). Trends, Challenges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape. https://www.youtube.com/watch?v=PjbGojjnBZQ. Accessed on 2023-02-24.

[Monzon and Fernandez-Sanchez, 2009] Monzon, A. and Fernandez-Sanchez, J. L. (2009). Deadlock risk assessment in architectural models of real-time systems. In *2009 IEEE International Symposium on Industrial Embedded Systems*, pages 181–190. IEEE.

[Moshtaghi, 2001] Moshtaghi, A. (2001). Modeling Multithreaded Applications Using Petri Nets. Master's thesis, The University of Alabama in Huntsville.

[Mozilla Wiki, 2015] Mozilla Wiki (2015). Oxidation Project. https://wiki.mozilla.org/Oxidation. Accessed on 2023-03-20.

[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.

[Nichols et al., 1996] Nichols, B., Buttlar, D., and Farrell, J. (1996). *Pthreads programming: A POSIX standard for better multiprocessing*. O'Reilly Media, Inc.

[Perronnet et al., 2019] Perronnet, F., Buisson, J., Lombard, A., Abbas-Turki, A., Ahmane, M., and El Moudni, A. (2019). Deadlock prevention of self-driving vehicles in a network of intersections. *IEEE Transactions on Intelligent Transportation Systems*, 20(11):4219–4233.

[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/.

[Rawson and Rawson, 2022] Rawson, M. and Rawson, M. (2022). Petri nets for concurrent programming. *arXiv preprint arXiv:2208.02900*.

[Reid, 2021] Reid, A. (2021). Automatic Rust verification tools (2021). https://alastairreid.github.io/automatic-rust-verification-tools-2021/. Accessed on 2023-02-20.

[Reid et al., 2020] Reid, A., Church, L., Flur, S., de Haas, S., Johnson, M., and Laurie, B. (2020). Towards making formal methods normal: meeting developers where they are. Accepted at HATRA 2020.

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

[Rust Project, 2023] Rust Project (2023). The rustc book. https://doc.rust-lang.org/rustc/. Accessed on 2023-02-20.

[Shibu, 2016] Shibu, K. V. (2016). *Introduction to Embedded Systems*. McGraw Hill Education (India), 2nd 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.

[Simone, 2022] Simone, S. D. (2022). Linux 6.1 Officially Adds Support for Rust in the Kernel. https://www.infoq.com/news/2022/12/linux-6-1-rust/. Accessed on 2023-02-24.

[Singhal, 1989] Singhal, M. (1989). Deadlock detection in distributed systems. *Computer*, 22(11):37–48.

[Stack Overflow, 2022] Stack Overflow (2022). 2022 Developer Survey. https://survey.stackoverflow.co/2022/#section-most-loved-dreaded-and-wanted-programming-scripting-and-markup-languages. Accessed on 2023-02-22.

[Stepanov, 2020] Stepanov, E. (2020). Detecting Memory Corruption Bugs With HWASan. https://android-developers.googleblog.com/2020/02/detecting-memory-corruption-bugs-with-hwasan.html. Accessed on 2023-02-24.

[Stoep and Hines, 2021] Stoep, J. V. and Hines, S. (2021). Rust in the Android platform. https://security.googleblog.com/2021/04/rust-in-android-platform.html. Accessed on 2023-02-22.

[Stoep and Zhang, 2020] Stoep, J. V. and Zhang, C. (2020). Queue the Hardening Enhancements. https://android-developers.googleblog.com/2020/02/detecting-memory-corruption-bugs-with-hwasan.html. Accessed on 2023-02-24.

[Szekeres et al., 2013] Szekeres, L., Payer, M., Wei, T., and Song, D. (2013). Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE.

[The Chromium Projects, 2015] The Chromium Projects (2015). Memory safety. https://www.chromium.org/Home/chromium-security/memory-safety/. Accessed on 2023-02-24.

[The Rust Project Developers, 2019] The Rust Project Developers (2019). Rust case study: Community makes rust an easy choice for npm. https://www.rust-lang.org/static/pdfs/Rust-npm-Whitepaper.pdf.

[Thompson, 2023] Thompson, C. (2023). How Rust went from a side project to the world's most-loved programming language. https://www.technologyreview.com/2023/02/14/1067869/rust-worlds-fastest-growing-programming-language/.

[Toman et al., 2015] Toman, J., Pernsteiner, S., and Torlak, E. (2015). Crust: a bounded verifier for rust (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 75–80. IEEE.

[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.

[van Steen and Tanenbaum, 2017] van Steen, M. and Tanenbaum, A. S. (2017). *Distributed Systems*. Pearson Education, 3rd edition.

[Wu and Hauck, 2022] Wu, Y. and Hauck, A. (2022). How we built Pingora, the proxy that connects Cloudflare to the Internet. https://blog.cloudflare.com/how-we-built-pingora-the-proxy-that-connects-cloudflare-to-the-internet/. Accessed on 2023-03-20.