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

List of Listings

Acronyms

ART Android Runtime

AST abstract syntax tree

BB basic blocks

CFG control flow graph

CPN Colored Petri nets

Creol Concurrent Reflective Object-oriented Language

DBMS Database management systems

FSM Finite-state machine

HIR High-Level Intermediate Representation

IR intermediate representation

ISA instruction set architecture

JIT just-in-time

LoLA Low Level Petri Net Analyzer

LTO link time optimization

MIR Mid-level Intermediate Representation

NT-PN Nondeterministic Transitioning Petri nets

OS operating system

PIPE2 Platform Independent Petri net Editor 2

PN Petri nets

PNML Petri Net Markup Language

RAG Resource Allocation Graph

RFCs Requests for Comments

THIR Typed High-Level Intermediate Representation

TWF transaction-wait-for

UB Undefined Behavior

WASM WebAssembly

Abstract

Detección de Deadlocks en Rust en tiempo de compilación mediante Redes de Petri

En la presente tesis de grado se presenta una herramienta de análisis estático para detección de *deadlocks* y señales perdidas en el lenguaje de programación Rust. Se realiza una traducción en tiempo de compilación del código fuente a una red de Petri. Se obtiene entonces la red de Petri como salida en uno o más de los siguientes formatos: DOT, Petri Net Markup Language o LoLA. Posteriormente se utiliza el verificador de modelos LoLA para probar de forma exhaustiva la ausencia de *deadlocks* y de señales perdidas. La herramienta está publicada como *plugin* para el gestor de paquetes *cargo* y la totalidad del código fuente se encuentra disponible en GitHub[1][2]. La herramienta demuestra de forma práctica la posibilidad de extender el compilador de Rust con un pase adicional para detectar más clases de errores en tiempo de compilación.

Compile-time Deadlock Detection in Rust using Petri Nets

This undergraduate thesis presents a static analysis tool for the detection of deadlocks and missed signals in the Rust programming language. A compile-time translation of the source code into a Petri net is performed. The Petri net is then obtained as output in one or more of the following formats: DOT, Petri Net Markup Language, or LoLA. Subsequently, the LoLA model checker is used to exhaustively prove the absence of deadlocks and missed signals. The tool is published as a plugin for the package manager *cargo* and the entirety of the source code is available on GitHub[1][2]. The tool demonstrates in a practical way the possibility to extend the Rust compiler with an additional pass to detect more error classes at compile time.

[1] https://github.com/hlisdero/granite2
[2] https://github.com/hlisdero/netcrab

Chapter 1

Introduction

In order to fully understand the scope and context of this work, it is important to provide some background topics that lay the foundation for the research. These background topics serve as the theoretical building blocks upon which the translation is built.

First, the theory of Petri nets is presented both graphically and in mathematical terms. To illustrate the modeling power and versatility of Petri nets, several different models are provided to the reader as examples. These models showcase the ability of Petri nets to capture various aspects of concurrent systems and represent them in a visual and intuitive manner. Later on, some important properties are introduced and the reachability analysis performed by the model checker is explained.

Second, the Rust programming language and its main characteristics are briefly discussed. A handful of examples of noteworthy applications of Rust in the industry are included. Compelling evidence for the use of memory-safe languages was gathered to argue the case that Rust provides an excellent base for extending the detection of classes of errors at compile time.

Third, a background on the problem of deadlocks and missed signals when using condition variables is provided, as well as a description of the common strategies used to address these issues.

Lastly, an overview of compiler architecture and the concept of model checking is provided. We will note the still untapped potential that formal verification offers to increase the safety and reliability of software systems.

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: 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.

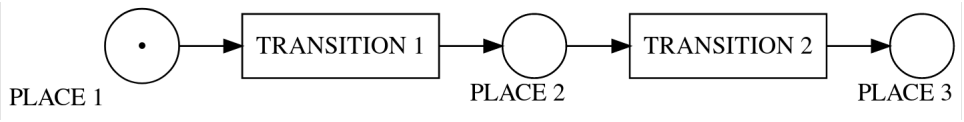

Figure 1.1: Example of a Petri net. PLACE 1 contains a token.

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.

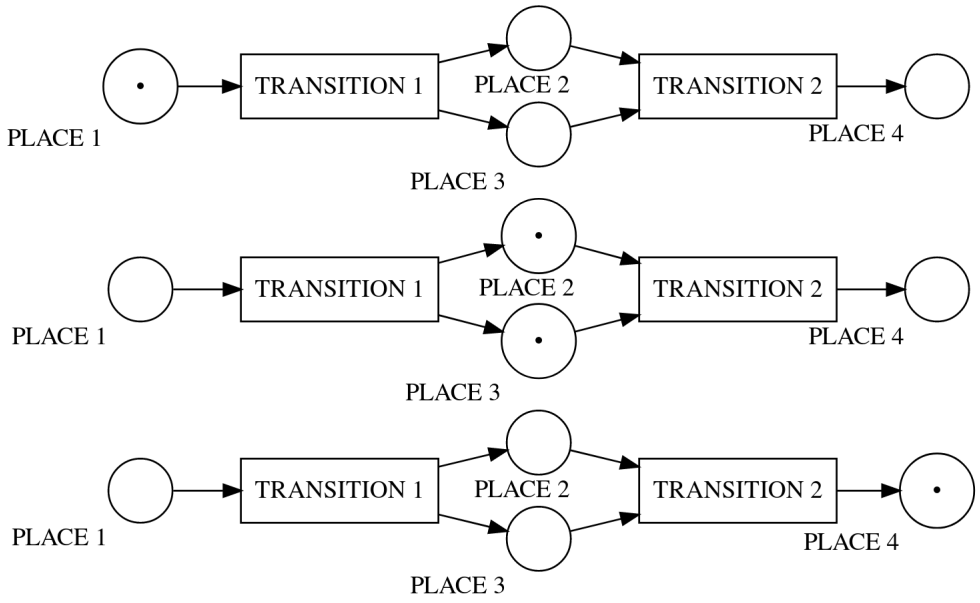

Figure 1.2: Example of transition firing: Transition 1 fires first, then transition 2 fires.

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 transitions 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, Chap. 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.

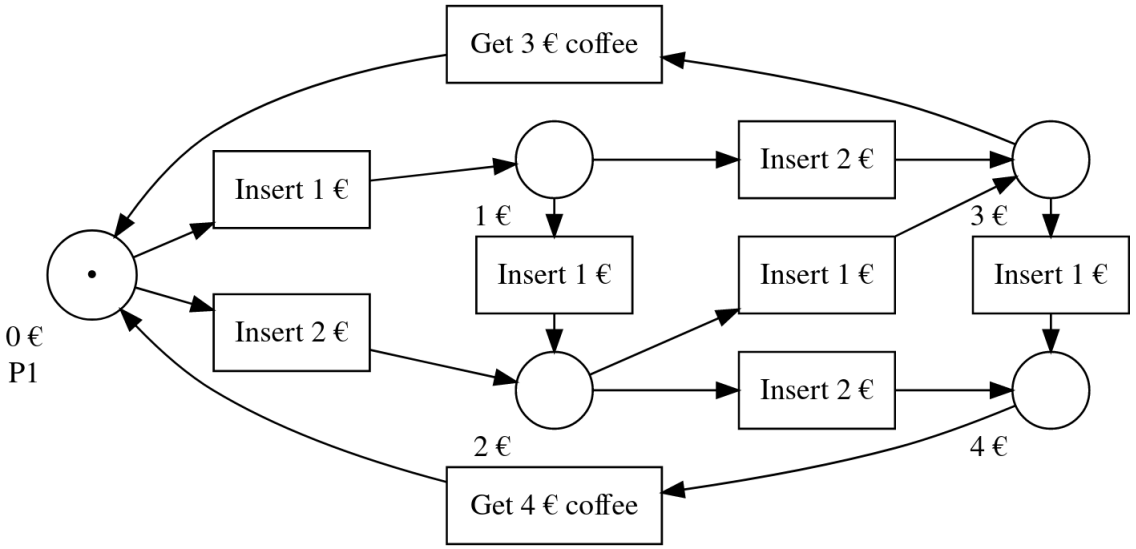

Figure 1.4: The Petri net for a coffee vending machine. It is equivalent to a state diagram.

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.

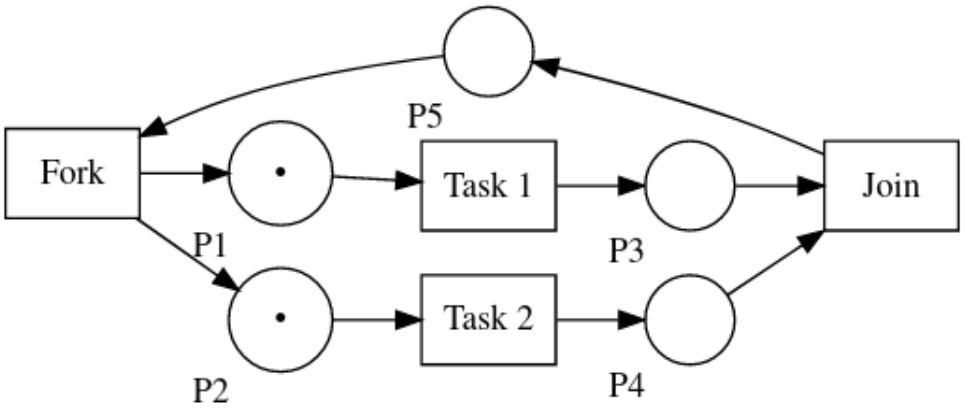

Figure 1.5: The Petri net depicting two parallel activities in a fork-join fashion.

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.

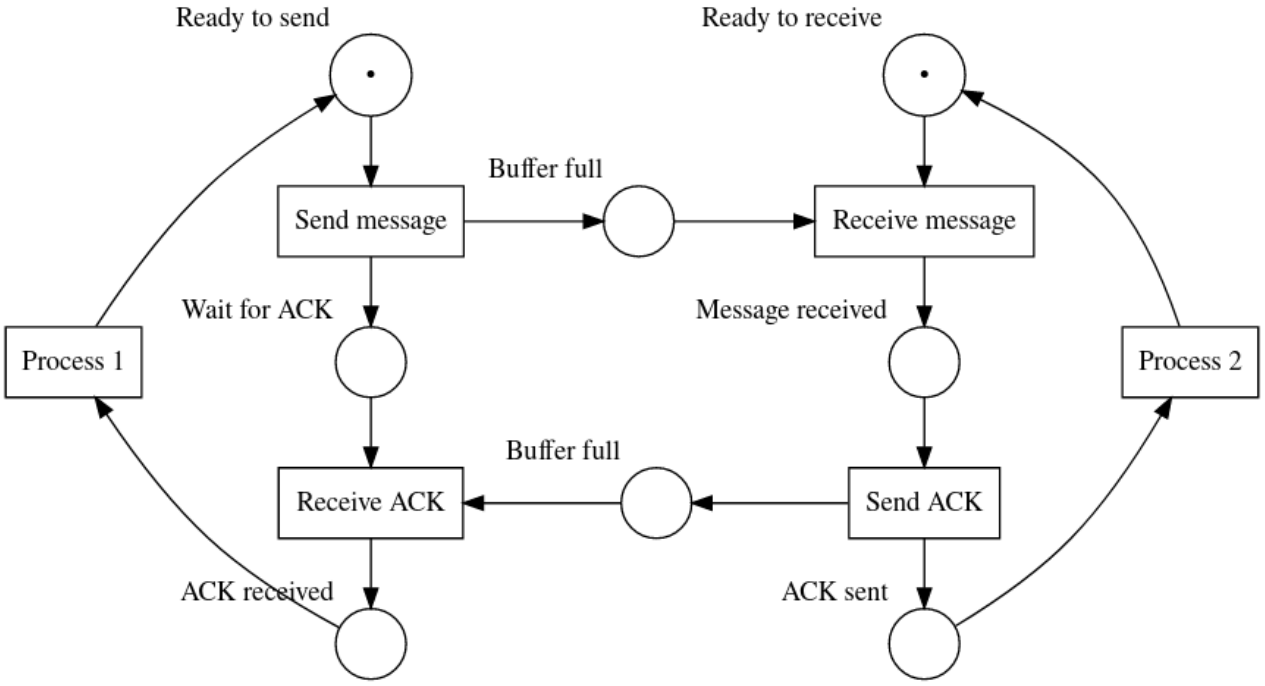

Figure 1.6: A simplified Petri net model of a communication protocol.

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.

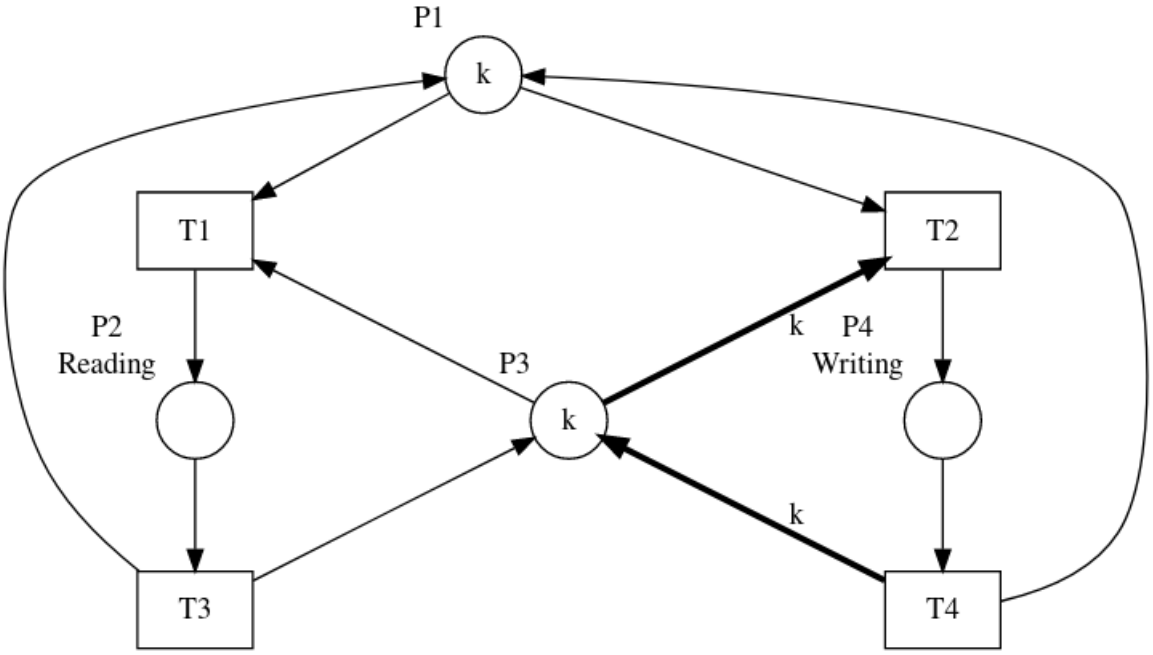

Figure 1.7: A Petri net system with k processes that either read or write.

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. 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

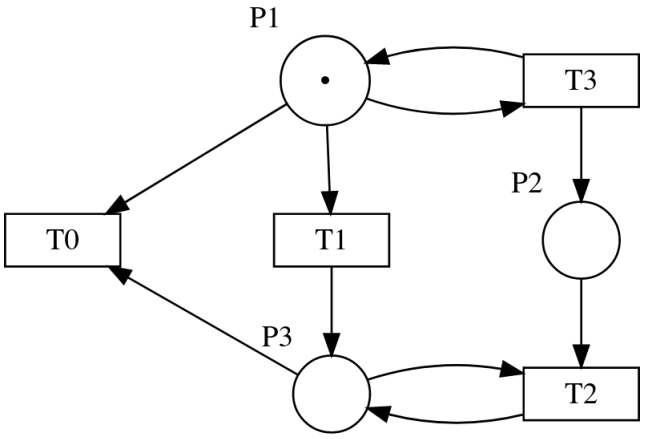

Figure 1.8: A marked Petri net for illustrating the construction of a reachability tree.

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.

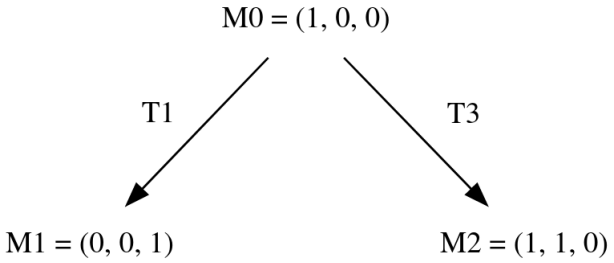

Figure 1.9: The first step building the reachability tree for the Petri net in Fig. 1.8.

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

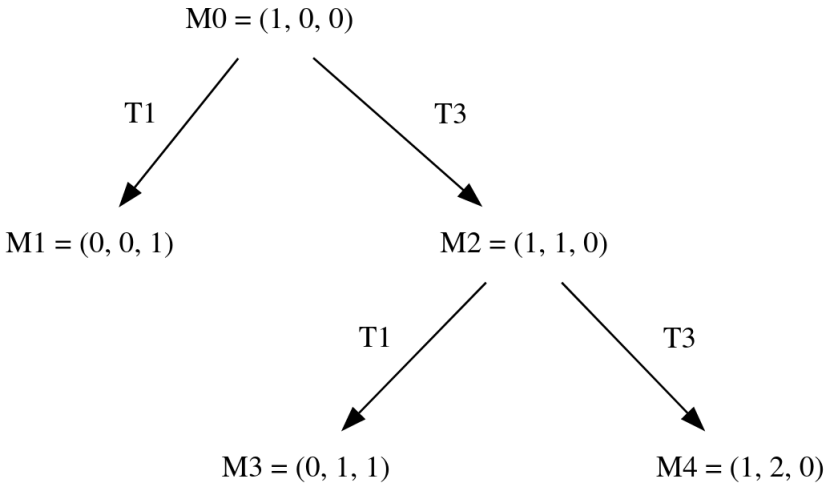

Figure 1.10: The second step building the reachability tree for the Petri net in Fig. 1.8.

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.

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

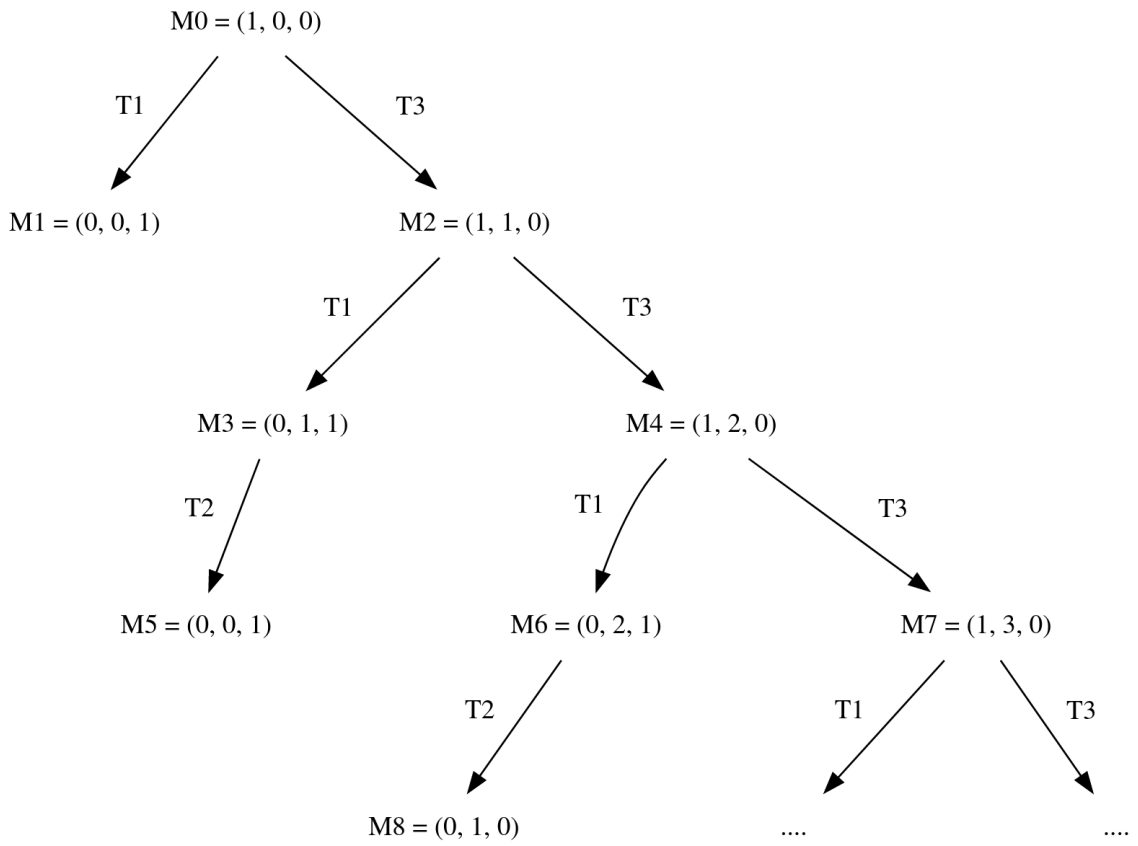

Figure 1.11: The infinite reachability tree for the Petri net in Fig. 1.8.

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.

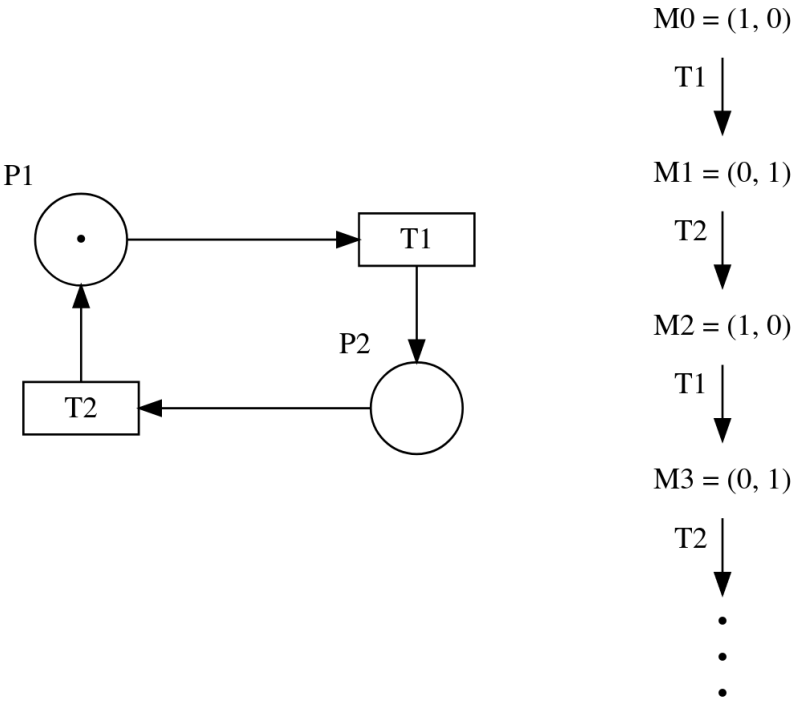

Figure 1.12: A simple Petri net with an infinite reachability tree.

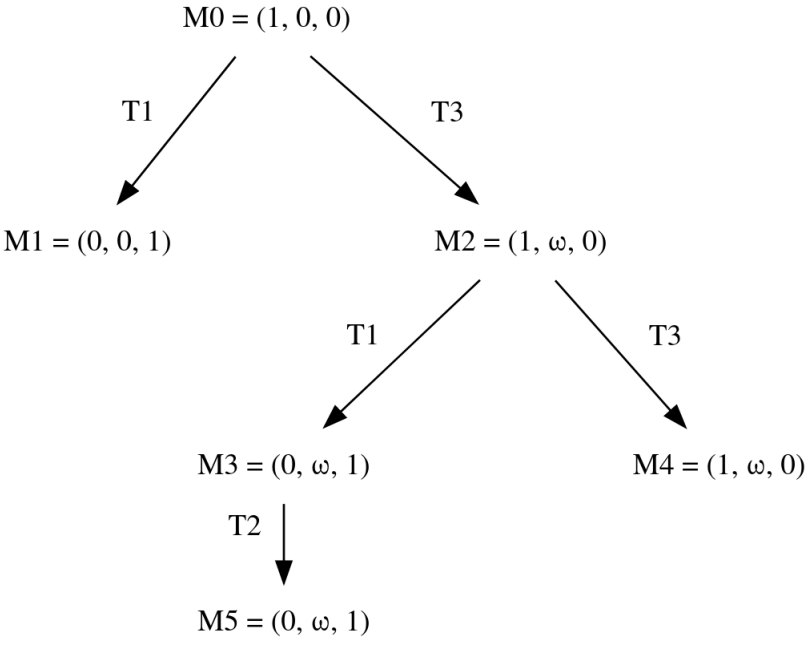

Figure 1.13: The finite reachability tree for the Petri net in Fig. 1.8.

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, Chap. 4.1]
- Use after free [Klabnik and Nichols, 2023, Chap. 4.1]
- Dangling pointers [Klabnik and Nichols, 2023, Chap. 4.2]
- Data races [Klabnik and Nichols, 2023, Chap. 4.2] (with some important caveats explained in [Rust Project, 2023b, Chap. 8.1])
- Passing non-thread-safe variables [Klabnik and Nichols, 2023, Chap. 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.

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

Rust's performance is achieved through a combination of features such as zero-cost abstractions, minimal runtime, and efficient memory management.

- 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:

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

- 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.
- 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 supporting 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]. An example of an important tool in this area, more precisely a dynamic data racer detector for multithreaded programs in C, can be found in [Savage et al., 1997], whose algorithm was later improved in

[Jannesari et al., 2009] and integrated into the Helgrind tool, part of the well-known Valgrind instrumentation framework[6].

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, 2023a, 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.

[6] `https://valgrind.org/`

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.

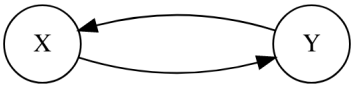

Figure 1.14: Example of a state graph with a cycle indicating a deadlock.

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 separately 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 may 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, Chap. 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), thus 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[7],
- Rust[8]
- Python[9]
- Go[10]
- Java[11]

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.

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:

1. T1 acquires the lock and sets `flag` to `true`.

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

```
// 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.

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 architecture

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, Chap. 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, Chap. 8.5.4],
- loop unrolling [Aho et al., 2014, Chap. 10.5],
- register allocation [Aho et al., 2014, Chap. 8.1.4],
- constant propagation [Aho et al., 2014, Chap. 9],
- liveness analysis [Aho et al., 2014, Chap. 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.

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.

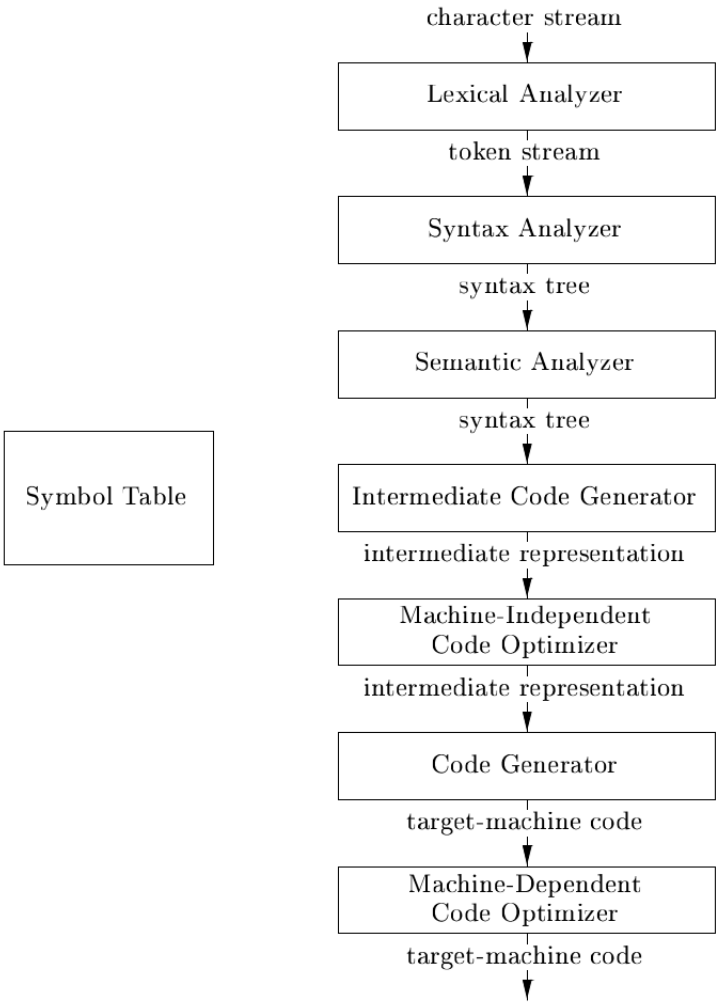

Figure 1.15: Phases of a compiler.

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 promptly during the development process to increase adoption.

The main conclusion from this section is that model checking could bring substantial improvements to the table in terms of increased safety and reliability of software systems. These objectives align with the goals of the Rust programming language and the goals of this work. Detecting deadlocks and missed signals in the source code at compile time could help developers prevent hard-to-find bugs and get quick feedback on the correct use of synchronization primitives, saving time and thus money in the development process. One particular goal of this work is to make the tool user-friendly and easy to get started with so that its adoption benefits the larger community of Rust developers.

Chapter 2

State of the Art

In this chapter, the literature on formal verification of Rust code and Petri net modeling for deadlock detection is briefly reviewed. Some of these previous publications contain approaches that have guided this work.

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

Finally, a survey of the existing Petri net libraries in the Rust ecosystem as of early 2023 is provided to justify the need to implement a library from the ground up.

2.1 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*[1] interpreter developed by the Rust project on GitHub is an experimental interpreter for the intermediate representation of the Rust language (Mid-level Intermediate Representation, commonly known as "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] https://github.com/rust-lang/miri

2.2 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 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 on GitHub[2] as part of the thesis is no longer valid for the new version of *rustc* since the internals of the compiler changed significantly in the last three years.

In a late 2022 pre-print, [Zhang and Liua, 2022] implement a translation of Rust source code to Petri nets for checking deadlocks. The translation focuses on deadlocks caused by two types of locks in the standard library: `std::sync::Mutex` and `std::sync::RwLock`. The resulting Petri net is expressed in the Petri Net Markup Language (PNML) and fed into the model checker Platform Independent Petri net Editor 2 (PIPE2)[3] to perform reachability analysis. Function calls are handled in a very different way compared to this work and missed signals are not modeled at all. The source code of their tool, named TRustPN, is not publicly available as of this writing. Despite these limitations, the authors offer a very detailed and up-to-date survey of static analysis tools for checking Rust code, which could be appealing to the interested reader. Moreover, they list several papers dedicated to formalizing the semantics of the Rust programming language, which are out of the scope of this work.

2.3 Petri nets libraries in Rust

As part of the development of the translation of the source code to a Petri net, it is necessary to use a Petri net library for the Rust programming language. A quick search of the packages available on *crates.io*[4], GitHub, and GitLab revealed that there is unfortunately no well-maintained library.

Some Petri net simulators were found such as:

- pns[5]: Programmed in C. It does not offer the option to export the resulting network to a standard format.
- PetriSim[6]: An old DOS/PC simulator programmed in Bordland Pascal.
- WOLFGANG[7]: A Petri net editor in Java, maintained by the Department of Computer

[2] https://github.com/Skasselbard/Granite
[3] https://pipe2.sourceforge.net/
[4] https://crates.io/
[5] https://gitlab.com/porky11/pns
[6] https://staff.um.edu.mt/jskl1/petrisim/index.html
[7] https://github.com/iig-uni-freiburg/WOLFGANG

Science at the University of Freiburg, Germany.

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

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

- petgraph[8]: The most widely used library for graphs in *crates.io*. It offers an option to export to the DOT format.
- gamma[9]: 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.

[8] https://docs.rs/petgraph/latest/petgraph/
[9] https://github.com/metamolecular/gamma

Chapter 3

Design of the proposed solution

Now that the relevant background topics have been covered, we can proceed to delve into the specifics of the design of the translation process. The design is marked by three crucial architectural choices that will be elaborated on in this chapter:

1. The decision to utilize the Rust compiler as a backend for the translation.
2. Basing the translation on the Mid-level Intermediate Representation (MIR).
3. Inlining function calls in the Petri net.

Throughout this chapter, we will conduct an in-depth analysis of the Rust compiler's internal mechanisms and its relevant compilation stages.

3.1 In search of a backend

To put it succinctly, two approaches for translating Rust code to Petri nets exist. The first option is to create a translator from scratch, while the second option is to build upon an existing tool.

The first option may seem attractive at first, considering that it gives the developer the freedom to shape the tool according to his/her desires. Features can be added as required and data structures can be tailored to the specific purpose. Nonetheless, this flexibility comes at a high price. In order to support a reasonable subset of the Rust programming language, substantial amounts of effort need to be invested into the task. Complex language constructs, such as macros, generics, or the rich type system itself, must be comprehended in their most intricate details to be translated effectively. The result is, essentially, a new compiler for Rust code. Noting that the Rust compiler was developed over many years and with the support of a large community of contributors, it becomes clear that this path is nothing more than work duplication. It is indeed a Herculean labor that would require the full-time dedication of a

whole team to maintain and keep up-to-date with the newest changes in the Rust language and the compiler.

On the other hand, there is the possibility of integrating with the existing Rust compiler, which is available under an open-source license and its documentation is extensive and regularly updated. This frees the implementation partly from having to deal with the changes to the language, giving more time to focus on the features that add value to the users. Hence the compiler plays the role of a backend on which the static analysis relies. Of course, this requires learning the compiler internals but this is not the first time that a tool sets off to do this. For instance, the official Rust linter, *clippy*[1], analyzes the Rust code for incorrect, inefficient, or non-idiomatic constructs. It is an extremely valuable tool for developers that goes beyond the standard checks performed during compilation.

Supporting all the language features from the beginning and collaborating with the community is key to the success of the proposed solution. Therefore, it is advisable to integrate with the existing ecosystem and reuse as much work as possible. Due to the above reasons, this project is based on *rustc*. We will now study the relevant parts of the Rust compiler in more detail.

3.2 Rust compiler: *rustc*

The Rust compiler, *rustc*, is responsible for translating Rust code into executable code. However, *rustc* is not a traditional compiler in the sense that it performs multiple passes over the code, as described in Sec. 1.6. Instead, *rustc* is built on a query-based system that supports incremental compilation.

In *rustc*'s query system, the compiler computes a dependency graph between code artifacts, including source files, crates, and intermediate artifacts, such as object files. The query system then uses this graph to efficiently recompile only those artifacts that have changed since the last compilation[2]. This incremental compilation can significantly reduce the compilation time for large projects, making it easier to develop and iterate on Rust code.

The query system also enables the Rust compiler to perform other optimizations, such as memoization and caching of intermediate results. For example, if a function's return value has been computed before, the query system can return the cached result instead of recomputing it, further reducing compilation time.

Another important design choice in *rustc* is interning. Interning is a technique to store strings and other data structures in a memory-efficient way. Instead of storing multiple copies of the same string or data structure, the Rust compiler stores only one copy in a special allocator called an *arena*. References to values stored in the arena are passed around between different parts of the compiler and they can be compared cheaply by comparing pointers. This can

[1] https://github.com/rust-lang/rust-clippy
[2] https://rustc-dev-guide.rust-lang.org/queries/incremental-compilation.html

reduce memory usage and speed up operations that compare or manipulate strings and data structures.

rustc uses the LLVM compiler infrastructure[3] to perform low-level code generation and optimization. LLVM provides a flexible framework for compiling code to a variety of targets, including native machine code and WebAssembly (WASM). The Rust compiler uses LLVM to optimize code for performance and to generate high-quality code for a variety of platforms. Instead of generating machine code, it only needs to generate the source code's LLVM intermediate representation (IR) and then instruct LLVM to transform this to the compilation target, applying the desired optimizations.

rustc is programmed in Rust. In order to compile the newer version of the compiler and the newer standard library version that goes with it, a slightly older version of *rustc* and the standard library is used. This process is called *bootstrapping* and it implies that one of the major users of Rust is the Rust compiler itself. Considering that a new stable version is released every six weeks, bootstrapping involves a substantial amount of complexity and is described in detail in the documentation[4] and in conferences [Nelson, 2022] and tutorials [Klock, 2022] by members of the Rust team.

3.2.1 Compilation stages

The existence of the query system does not imply that *rustc* does not have compilation phases at all. On the contrary, several stages of compilation are required to transform Rust source code into machine code that can be executed on a computer. These stages involve multiple intermediate representations of the program, each one optimized for a specific purpose. We will now briefly describe these stages. A more complete overview is found in the documentation[5].

Lexing and parsing

First, the raw Rust source text is analyzed by a low-level lexer. At this stage, the source text is turned into a stream of atomic source code units known as tokens.

Then parsing takes place. The stream of tokens is converted into an AST. Interning of string values occurs here. Macro expansion, AST validation, name resolution, and early linting also take place during this stage. The resulting intermediate representation from this step is thus the AST.

HIR lowering

Next, the AST is converted to High-Level Intermediate Representation (HIR). This process is known as "lowering". This representation looks like Rust code but with complex constructs

[3] https://llvm.org/
[4] https://rustc-dev-guide.rust-lang.org/building/bootstrapping.html
[5] https://rustc-dev-guide.rust-lang.org/overview.html

desugared to simpler versions. For instance, all `while` and `for` loops are converted into simpler `loop` loops.

The HIR is used to perform some important steps:

1. *type inference*: The automatic detection of a type of an expression, e.g. when declaring variables with `let`.
2. *trait solving*: Ensuring that each implementation block (`impl`) refers to a valid, existing trait.
3. *type checking*: This process converts the types written by the user into the internal representation used by the compiler. It is, in other words, where types are interned. Then, using this information, the type safety, correctness, and coherence are verified.

MIR lowering

In this stage, the HIR is lowered to Mid-level Intermediate Representation (MIR), which is used for *borrow checking*. As part of the process, the Typed High-Level Intermediate Representation (THIR) is constructed, which is a representation that is easier to convert to MIR than the HIR.

The THIR is an even more desugared version of HIR. It is used for pattern and exhaustiveness matching. It is similar to the HIR, but with all types and method calls made explicit. Furthermore, implicit dereferences are included where needed.

Many optimizations are performed on the MIR as it is still a very generic representation. Optimizations are in some cases easier to perform on the MIR than on the subsequent LLVM IR.

Code generation

This is the last stage when producing a binary. It includes the call to LLVM for code generation and the corresponding optimizations. To this effect, the MIR is converted to LLVM IR.

LLVM IR is the standard form of input for the LLVM compiler that all compilers using LLVM, such as the *clang* C compiler, utilize. It is a type of assembly language that is well-annotated and designed to be easy for other compilers to produce. Additionally, it is designed to be rich enough to enable LLVM to perform several optimizations on it.

LLVM transforms the LLVM IR to machine code and applies many more optimizations. Finally, the object files with assembly code in them may be linked together to form the binary.

3.2.2 Rust nightly

Understanding the release model of Rust is crucial for the successful implementation of the tool proposed in this work. The reason is that to use the crates of *rustc* as a dependency in our project, it must be compiled with the *nightly* version.

The nightly Rust compiler refers to a specific build of *rustc* that is updated every night with the latest changes and improvements but also includes experimental or unstable features that are not yet part of the stable release. In Rust, the language and its standard library are versioned using a "release train" model, where there are three main release channels: stable, beta, and nightly[6].

The stable release of the Rust compiler is the most widely used and recommended version for production use. It goes through a rigorous testing and stabilization process to ensure that it provides a stable and reliable experience for developers. The stable release only includes features and improvements that have been thoroughly reviewed, tested, and deemed stable enough for production use.

On the other hand, the nightly Rust compiler is the most bleeding-edge version, where new features, bug fixes, and experimental changes are introduced on a daily basis. It is used by Rust language developers and contributors for testing and development purposes but it is not recommended for production use due to the potential instability and lack of long-term support.

Each feature exclusive to the nightly version is behind a so-called *feature flag*. They may only be used when compiling with the nightly toolchain. Features flags may enable

- syntactic constructs that are not available on the stable version,
- library functions exclusive to the nightly version,
- support for specific hardware instructions of a given ISA or platform,
- additional compiler flags.

The full list of feature flags is found in [Rust Project, 2023c] and contains more than 500 entries in total. In a more concise manner, the Rust language used inside of *rustc* is a superset of the stable Rust language used outside of it. These differences should be taken into account when working on the compiler or building software that directly depends on the compiler.

3.3 Interception strategy: Selecting a suitable starting point for the translation

In this section, a rationale for selecting the Mid-level Intermediate Representation (MIR) as the starting point for the translation to a Petri net is elucidated. This architectural design choice is justified for several reasons.

3.3.1 Benefits

First, the MIR is the lowest machine-independent IR used in *rustc*. It captures the semantics of Rust code after it has undergone a series of optimization passes without relying on the details

[6] https://forge.rust-lang.org/

of any particular machine. By intercepting the translation at this stage, the static analysis tool leverages the benefits of these optimizations, such as constant folding, dead code elimination, and inlining, which results in a more efficient and overall smaller Petri net representation.

Second, intercepting the compilation after the previous stages are completed offers an advantage in terms of efficiency and code reuse. By this stage, the Rust compiler has already performed crucial steps such as borrow checking, type checking, monomorphization of generic code, and macro expansion, among others. These steps are resource-intensive and involve complex analysis of the Rust code to ensure correctness and safety. Re-implementing these steps in our tool from scratch would be redundant and time-consuming. It would require duplicating the efforts of the Rust compiler and may introduce potential inconsistencies or errors. By building on top of the existing MIR, we take advantage of the work already done by *rustc*. This not only saves effort but also ensures that our static analysis tool is aligned with the same level of correctness and safety as the Rust compiler.

Third, it simplifies the maintenance task of staying up-to-date with the ongoing additions to the Rust language and its compiler. Rust is a fast-evolving language and its compiler is constantly updated with new features, bug fixes, and performance optimizations. Repurposing the MIR means our tool can benefit from these updates without having to independently implement and maintain those changes. This provides overall a more robust and reliable static analysis solution.

Furthermore, as it will be explained in the next section, the MIR is based on the concept of a control flow graph (CFG), i.e. a type of graph found in compilers. That means that MIR and Petri nets are both graph representations, which makes the MIR particularly amenable to translation. Both MIR and Petri nets can be seen as graphical models that capture the relationships and interactions between different entities. The MIR graph represents the underlying execution flow within a Rust program, while a Petri net captures the state transitions and event occurrences in a system. Therefore, it becomes easier to convert the MIR to a Petri net, as the graph structure and relationships are already present. This allows for a more straightforward and efficient translation process without having to create a graph structure from thin air, resulting in better integration between the MIR and the Petri net model for deadlock detection.

Finally, working with the MIR synergizes with incremental compilation and modular analysis. In fact, one of the reasons why MIR was introduced in the first place was incremental compilation [Matsakis, 2016]. Even though it is not mandatory in the initial implementation, the tool could profit from incremental compilation and perform analysis on a per-crate/per-module basis, allowing for faster and more efficient analysis of large Rust codebases.

3.3.2 Limitations

There are however some limitations to the approach of basing the translation on the Mid-level Intermediate Representation (MIR).

The most important one is that the MIR is subject to change. No stability guarantees are made

in terms of how the Rust code will be translated to MIR or which its constituent elements MIR are. These are internal details that the compiler developers reserve for themselves. In short, MIR as an interface is not stable. As work on the compiler continues, the MIR undergoes modifications to incorporate new language features, optimizations, or bug fixes, which may require frequent updates and adjustments to the translation process, increasing the maintenance cost.

In the course of this project, this situation occurred numerous times. As an example, in the period between mid-February 2023 and mid-April 2023, the code was modified 7 times to accommodate these changes. They were always a few lines of code in size and detected by tests. We will discuss how tests play a significant role in coping with these changes in Sec. 5.4.

On the same note, [Meyer, 2020] also relied on the MIR but did not incorporate tests to deal with the newer nightly versions. As a result, the toolchain was pinned to an exact nightly version[7] to prevent the implementation from breaking before the publication of the thesis.

Another drawback worth mentioning is that, in some instances, generic code could take the form of a function whose behavior can be modeled by the same Petri net in all cases. Under these circumstances, the MIR could be "condensed" further before translating it to a Petri net. Similarly, some parts of the MIR may be superfluous to the deadlock detection analysis and its translation may enlarge the output, which slows down the reachability analysis done by the model checker. This can be countered with careful optimizations, which will be proposed in Chapter 7.

3.3.3 Synthesis

In conclusion, despite the drawbacks mentioned earlier, intercepting the translation at the MIR level offers significant advantages, including maximizing the utilization of the existing compiler code, reducing the implementation effort, and a more natural mapping to Petri nets. These benefits outweigh the cons and make the MIR a compelling starting point for the translation in the context of building a static analysis tool for detecting deadlocks and missed signals in Rust code.

Both [Meyer, 2020] and [Zhang and Liua, 2022] base their translations on the MIR as well and to the best knowledge of this author, there is no analogous tool that performed a translation to Petri nets starting from a higher-level IR.

3.4 Mid-level Intermediate Representation (MIR)

An overview of the Mid-level Intermediate Representation (MIR) is provided in this section. MIR was introduced in RFC 1211 in August 2015. We will explore its different parts, how different code fragments are mapped to them, and the underlying graph structure.

[7] https://github.com/Skasselbard/Granite/blob/master/rust-toolchain

```rust
fn main() {
    match std::env::args().len() {
        1 => 2,
        3 => 6,
        _ => 0,
    };
}
```

Listing 3.1: Simple Rust program to explain the MIR components.

```
// WARNING: This output format is intended for human consumers only
// and is subject to change without notice. Knock yourself out.
fn main() -> () {
    let mut _0: ();              // return place in scope 0 at src/main.rs:1:11: 1:11
    let mut _1: usize;           // in scope 0 at src/main.rs:2:11: 2:33
    let mut _2: &std::env::Args; // in scope 0 at src/main.rs:2:11: 2:33
    let _3: std::env::Args;      // in scope 0 at src/main.rs:2:11: 2:27

    bb0: {
        _3 = args() -> bb1;      // scope 0 at src/main.rs:2:11: 2:27
                                 // mir::Constant
                                 // + span: src/main.rs:2:11: 2:25
                                 // + literal: Const { ty: fn() ->
                                 //   Args {args}, val: Value(<ZST>) }
    }

    bb1: {
        _2 = &_3;                // scope 0 at src/main.rs:2:11: 2:33
        _1 = <Args as ExactSizeIterator>::len(move _2) -> [return: bb2, unwind: bb4];
                                 // scope 0 at src/main.rs:2:11: 2:33
                                 // mir::Constant
                                 // + span: src/main.rs:2:28: 2:31
                                 // + literal: Const { ty: for<'a> fn(&'a Args) ->
                                 //   usize {<Args as ExactSizeIterator>::len},
                                 //   val: Value(<ZST>) }
    }

    bb2: {
        drop(_3) -> bb3;         // scope 0 at src/main.rs:6:6: 6:7
    }

```

```
32      bb3: {
33          return;                     // scope 0 at src/main.rs:7:2: 7:2
34      }
35
36      bb4 (cleanup): {
37          drop(_3) -> [return: bb5, unwind terminate]; // scope 0 at src/main.rs:6:6: 6:7
38      }
39
40      bb5 (cleanup): {
41          resume;                     // scope 0 at src/main.rs:1:1: 7:2
42      }
43  }
```

Listing 3.2: MIR of Listing 3.1 compiled using rustc 1.71.0-nightly in debug mode.

Consider the example code listed in Listing 3.1, the corresponding MIR[8] is shown in Listing 3.2. Notice the explicit warning at the top of the generated output. It will be omitted in the subsequent listings for simplicity. Moreover, output depends on the following factors:

- The *rustc* version in use, alternatively the release channel (stable, beta, or nightly).
- The build type: *debug* or *release*. By default, the command `cargo build` generates a *debug* build, while `cargo build --release` produces a *release* build.

To illustrate this variability, Listing 3.3 shows the output when compiling the same program in *release* mode. The distinguishing feature found in *release* builds is the presence of the `StorageLive` and `StorageDead` statements. On the other hand, *debug* builds generate shorter and clearer MIR that is closer to what the user wrote. For this reason, unless otherwise stated, the listings in this work contain MIR generated in *debug* builds.

```
1   // WARNING: This output format is intended for human consumers only
2   // and is subject to change without notice. Knock yourself out.
3   fn main() -> () {
4       let mut _0: ();                 // return place in scope 0 at src/main.rs:1:11: 1:11
5       let mut _1: usize;              // in scope 0 at src/main.rs:2:11: 2:33
6       let mut _2: &std::env::Args;    // in scope 0 at src/main.rs:2:11: 2:33
7       let _3: std::env::Args;         // in scope 0 at src/main.rs:2:11: 2:27
8
9       bb0: {
10          StorageLive(_1);            // scope 0 at src/main.rs:2:11: 2:33
11          StorageLive(_2);            // scope 0 at src/main.rs:2:11: 2:33
12          StorageLive(_3);            // scope 0 at src/main.rs:2:11: 2:27
13          _3 = args() -> bb1;         // scope 0 at src/main.rs:2:11: 2:27
```

[8]The comments in the MIR have been slightly modified to improve the output

```
14                                  // mir::Constant
15                                  // + span: src/main.rs:2:11: 2:25
16                                  // + literal: Const { ty: fn() ->
17                                  //   Args {args},
18                                  //   val: Value(<ZST>) }
19      }
20
21      bb1: {
22          _2 = &_3;               // scope 0 at src/main.rs:2:11: 2:33
23          _1 = <Args as ExactSizeIterator>::len(move _2) -> [return: bb2, unwind: bb4];
24                                  // scope 0 at src/main.rs:2:11: 2:33
25                                  // mir::Constant
26                                  // + span: src/main.rs:2:28: 2:31
27                                  // + literal: Const { ty: for<'a> fn(&'a Args) ->
28                                  //   usize {<Args as ExactSizeIterator>::len},
29                                  //   val: Value(<ZST>) }
30      }
31
32      bb2: {
33          StorageDead(_2);        // scope 0 at src/main.rs:2:32: 2:33
34          drop(_3) -> bb3;        // scope 0 at src/main.rs:6:6: 6:7
35      }
36
37      bb3: {
38          StorageDead(_3);        // scope 0 at src/main.rs:6:6: 6:7
39          StorageDead(_1);        // scope 0 at src/main.rs:6:6: 6:7
40          return;                 // scope 0 at src/main.rs:7:2: 7:2
41      }
42
43      bb4 (cleanup): {
44          drop(_3) -> [return: bb5, unwind terminate]; // scope 0 at src/main.rs:6:6: 6:7
45      }
46
47      bb5 (cleanup): {
48          resume;                 // scope 0 at src/main.rs:1:1: 7:2
49      }
50  }
```

Listing 3.3: MIR of Listing 3.1 compiled using rustc 1.71.0-nightly in release mode.

The specific formatting when converting MIR to a string has changed only slightly over time. See [Meyer, 2020, Section 3.3] for an example of older output from mid-2019.

As stated in Sec. 3.3, the MIR is derived from a previously existing control flow graph (CFG) in the Rust compiler. Fundamentally, a CFG is a graph representation of a program that exposes the underlying control flow.

The MIR is formed by functions. Each function is represented as a series of basic blocks (BB) connected by directed edges. Each BB contains zero or more *statements* (usually abbreviated as "STMT") and lastly one *terminator statement*, for short *terminator*. The terminator is the only statement in which the program can issue an instruction that directs the control flow to another basic block inside the same function or to call another function. Branching as in Rust's `match` or `if` statements can occur only in terminators. Terminators play the role of mapping the high-level constructs for conditional execution and looping to the low-level representation in machine code as simple conditional or unconditional `branch` instructions.

In Fig. 3.1, the graph representation for the MIR shown in Listing 3.2 is presented as an example. The statements are colored in light blue and the terminators in light red. To make the kind of terminator statement clearer, extra annotations as in `CALL:` or `DROP:` were added.

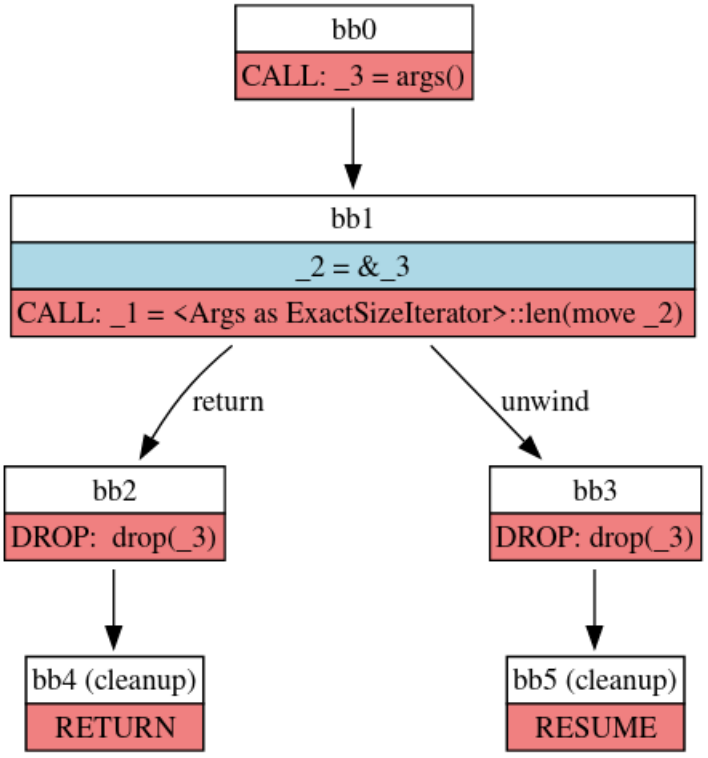

Figure 3.1: The control flow graph representation of the MIR shown in Listing 3.2.

It should be noted that the function call to `std::env::args().len()` in Line 2 in Listing 3.1 may return successfully or fail. The latter triggers an unwinding of the stack, ending the program and reporting an error. This is represented by the branching at the end of BB1 where the code execution may take the left path or the right path down the graph. The left branch (BB4 and

BB5) corresponds to the correct execution of the program, while the right branch relates to the abnormal termination of the program.

There are different kinds of terminators and these are specific to the Rust semantics. We will introduce some of them to clarify the meaning of the example presented.

- As expected, a terminator of type `CALL:` calls a function, which returns a value, and continues execution to the next BB.
- A terminator of type `DROP:` frees up the memory of the variable passed in. It executes the destructors[9] and performs all the necessary cleanup tasks. From that point on, the variable cannot be used anymore in the program.
- `RETURN:` returns from the function. The return value is always stored in the local variable `_0`, as we will see shortly.
- `RESUME:` indicates that the process should continue unwinding. Analogously to a return, this marks the end of this invocation of the function. It is only permitted in cleanup blocks.

The complete list of terminator kinds can be found in the nightly documentation[10]. Other kinds of terminators will be discussed in detail in Sec. 4.4.3.

Regarding the variables, the data in MIR can be divided into two categories: *locals* and *places*. It is critical to observe that these "places" are *not* related to the places in Petri nets. Places are used to represent all types of memory locations (including aliases), while locals are limited to stack-based memory locations (i.e. local variables of a function). In other words, places are more general and locals are a special case of a place, therefore places are not always equivalent to locals. Conveniently, all the places are also locals in Fig. 3.1.

Locals are identified by an increasing non-negative index and are emitted by the compiler as a string of the form "`_<index>`". In particular, the return value of the function is always stored in the first local `_0`. This matches closely the low-level representation on the stack.

3.4.1 Step-by-step example

In this subsection, we will give a short explanation of what happens in each basic block of Fig. 3.1 to ensure that all necessary information is covered. Moreover, this illustrates how the MIR output represents higher-level constructs often encountered when programming in Rust.

BB0

- The `main()` function starts at BB0.

[9] https://doc.rust-lang.org/stable/reference/destructors.html
[10] https://doc.rust-lang.org/stable/nightly-rustc/rustc_middle/mir/enum.TerminatorKind.html

- A function is called (`std::env::args()`) to obtain an iterator over the arguments provided to the program.
- The return value of the function, the iterator, is assigned to the local `_3`.
- Execution continues in BB1.

BB1

- A reference to the iterator stored in `_3` is generated and stored in the local `_2` (similar to the "&" operator in C). This is necessary for calling methods because methods receive a reference to a struct of the same type (`&self`) as their first argument.
- The reference stored in `_2` is passed to the method `std::env::Args::len()` by moving and the function is called.
- The return value of the function, the number of arguments passed to the function, is assigned to the local `_1`.
- Execution continues in BB2 if successful, in BB4 in case of panic.

BB2

- The variable `_3`, whose value is the iterator over the arguments, is *dropped* since it is no longer needed.
- Execution continues in BB3.

BB3

- The function returns. The return value (local `_0`) is of type "unit"[11], which is similar to a `void` function in C, i.e. it does not return anything. This is how `main()` was defined in Listing 3.1.

BB4

- The variable `_3`, whose value is the iterator over the arguments, is *dropped* since it is no longer needed.
- If the drop is successful, execution continues in BB5, otherwise terminate the program immediately.

[11] https://doc.rust-lang.org/std/primitive.unit.html

BB5

- Continue unwinding the stack. This is the standard protocol defined for handling catastrophic error cases that cannot be handled by the program. Implementation details can be found in the documentation[12]

3.5 Function inlining in the translation to Petri nets

In this section, a thorough analysis and motivation for the third design decision listed at the beginning of the chapter, namely inlining function calls, is presented.

Modeling functions in PN is a crucial aspect of the translation because it is the basic unit of the MIR. By representing the functions in the MIR as PN and connecting them accordingly, the control flow and data shared between the threads in the program can be captured in a formal framework. Afterward, the Petri net is analyzed by a model checker in order to identify potential deadlocks or lost signals. This approach is especially useful when working with large and complex systems that may have many interrelated threads and functions, where the deadlock situation may not be evident even to an experienced code reviewer.

When translating MIR functions to PN, one key question that arises is whether to reuse the same representation for every call to a specific function or to "inline" the corresponding representation every time the function is called. Expressed differently, each function maps to a subnet in the final PN obtained after the translation, i.e. a connected subgraph formed by the places and transitions that model the behavior of the specific function. This smaller part of the net can either be present only once in the PN and all calls to this function connect to it, or be repeated for every instance of a call to the function in the Rust code.

Reusing the same model for every function seems at first glance more efficient, as the PN obtained is smaller. However, this approach can also lead to invalid states that were not present in the original Rust program. These can be the source of false positives during deadlock detection, as these extraneous states may violate the safety guarantees offered by the compiler.

On the other hand, inlining the model every time a function is called results in a larger PN, which requires more memory and processor time to be analyzed, but it can also improve the accuracy of the analysis by ensuring that each function call is represented by a separate Petri net structure that captures its specific data dependencies in the context in which the function call occurs in the code.

3.5.1 The basic case

The impact of these subtle details can only be fully comprehended with an appropriate example. Therefore, consider first the most simple abstraction of a function call in the language of Petri nets, formed by a single transition and two places representing the start and end of the function.

[12] https://rustc-dev-guide.rust-lang.org/panic-implementation.html.

```rust
fn simple_function() {}

pub fn main() {
    for n in 0..5 {
        simple_function();
    }
}
```

Listing 3.4: A simple Rust program with a repeated function call.

This is seen in Fig. 3.2. The function call is treated as a black box, all details are abstracted away in the transition. We care only about where the function starts and where it ends.

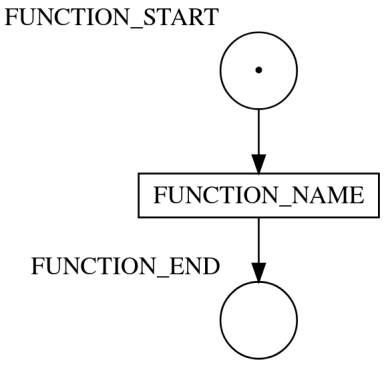

Figure 3.2: The simplest Petri net model for a function call.

Observe now such a function in the context of a Rust program. Listing 3.4 provides a simple example in which one function is called five times consecutively in a `for` loop. A possible PN that models the program is found in Fig. 3.3. It should be emphasized that this net and the subsequent ones in this section do *not* result from a translation of the MIR. They are simplifications to showcase the difficulties of dealing with functions called in various places in the code.

3.5.2 A characterization of the problem

The troublesome scenario has not emerged so far. It manifests only when a function is called in at least two different places in the code or, in simpler terms, the expression `simple_function()` appears twice or more. Listing 3.5 satisfies this condition and is designed to exhibit the extraneous behavior described at the beginning of the section.

As stated before, the first approach to modeling the program consists in reusing the function model for both calls. This is shown in Fig. 3.4.

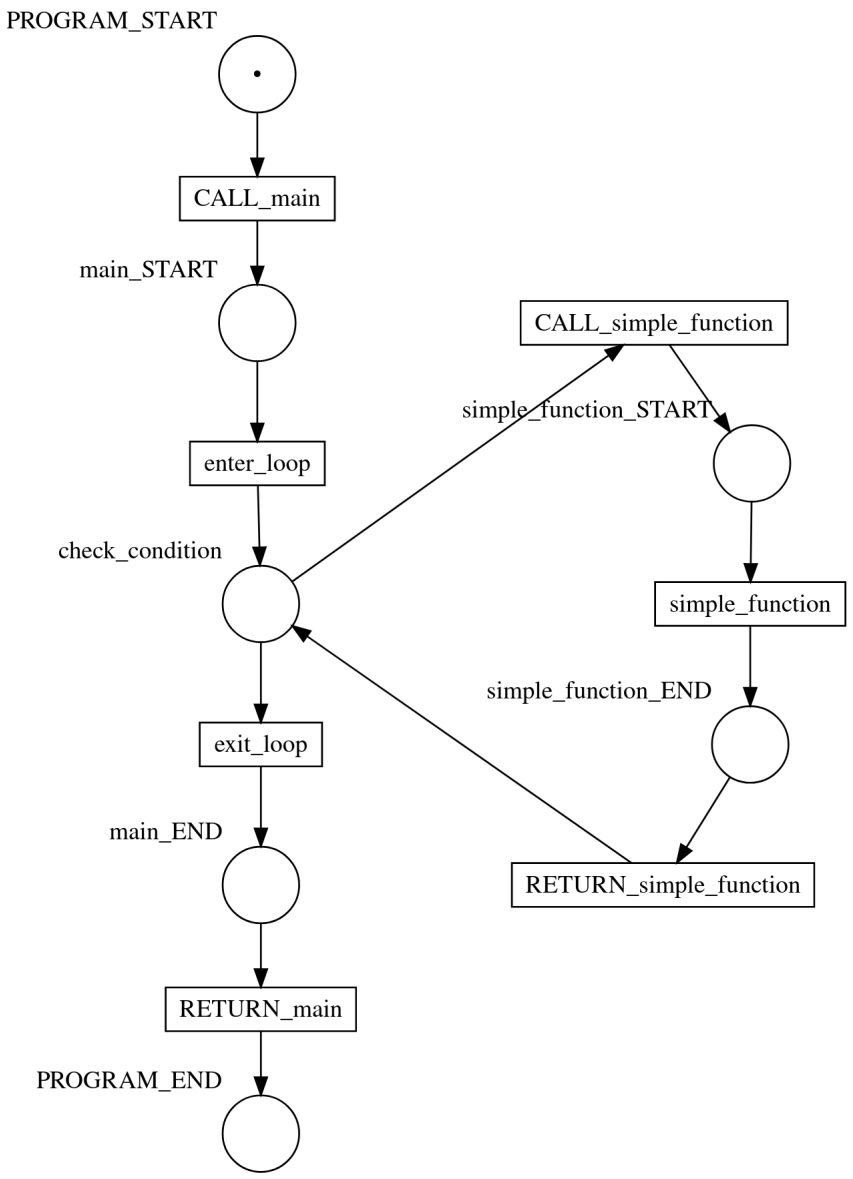

Figure 3.3: A possible PN for the code in Listing 3.4 applying the model of Fig. 3.2.

It is evident to the reader that the program in Listing 3.5 never calls the `panic!()` macro and always terminates successfully, given that the variable `second_call` is never `true` before line 9.

Yet, the PN depicted in Fig. 3.4. is conspicuously flawed, making it unsuitable as a model for the program. The reason is that after firing the transition labeled `RETURN_simple_function` a token is placed in `check_flag` but *also* in `main_end_place`. The token in `main_end_place` will eventually appear in `PROGRAM_END`, which indicates a normal termination of the program. This is technically correct since we know that the program terminates successfully.

Nonetheless, there are concerning issues regarding the second token. The token in `check_flag`

```
1  fn simple_function() {}
2
3  pub fn main() {
4      let mut second_call = false;
5      simple_function();
6      if second_call {
7          panic!()
8      }
9      second_call = true;
10     simple_function();
11 }
```

Listing 3.5: A simple Rust program that calls a function in two different places.

could be consumed either by the transition `flag_is_false` or `flag_is_true`. If it is consumed by the latter, a token will be placed in `PROGRAM_PANIC`, signaling an erroneous termination of the program. This is absurd because it means that the program could panic but also *always* ends normally, as seen in the previous paragraph.

The situation becomes worse if we follow the path of firing `flag_is_false`. In that case, the token triggers another function call, which is in principle correct, but nothing prevents it from doing this over and over again. The conclusion is that an infinite amount of tokens could accumulate in `main_end_place` or `PROGRAM_END` in the circumstance that, by pure chance, the transition `flag_is_true` does not fire.

It has become clear that we must discard this model and look for a better solution. One possibility is to split the transition labeled `RETURN_simple_function` in two separate transitions depending on the function call order as illustrated in Fig. 3.5.

This second attempt unfortunately comes with its own set of extraneous states. First, the program may now exit after calling the function only once. Nothing prevents the transition `RETURN_simple_function_2` from firing first. This is equivalent to saying that the execution flow jumps from line 5 to line 11 in Listing 3.5, which is obviously not a property present in the original Rust code.

On the other hand, the problem of the infinite loop persists. The PN may continue firing indefinitely as long as `flag_is_true` and `RETURN_simple_function_2` do not fire. There is no guarantee that the transitions fire in a specific order. As seen in Sec. 1.1.3, the transition firing is non-deterministic.

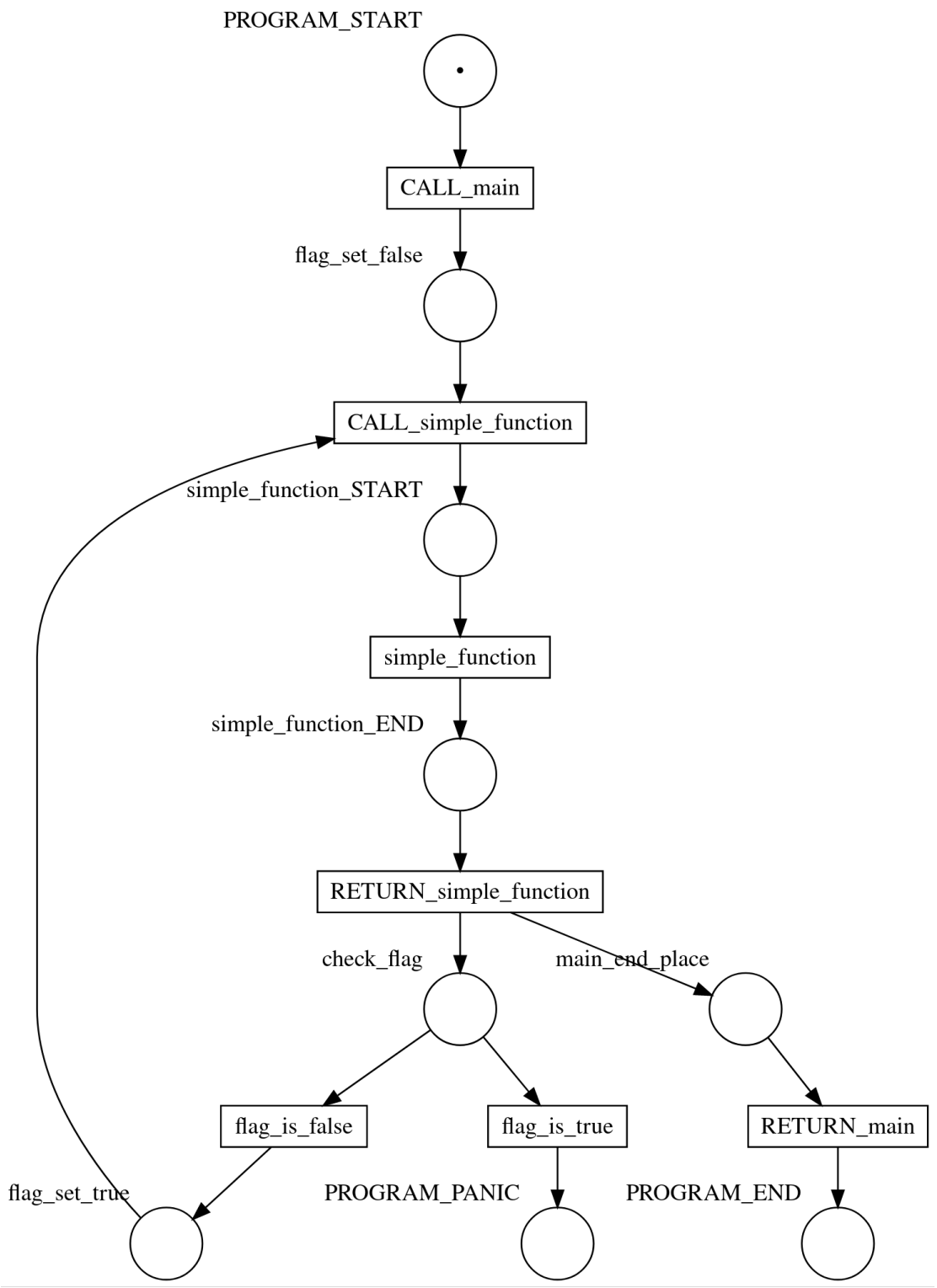

Figure 3.4: A first (incorrect) PN for the code in Listing 3.5.

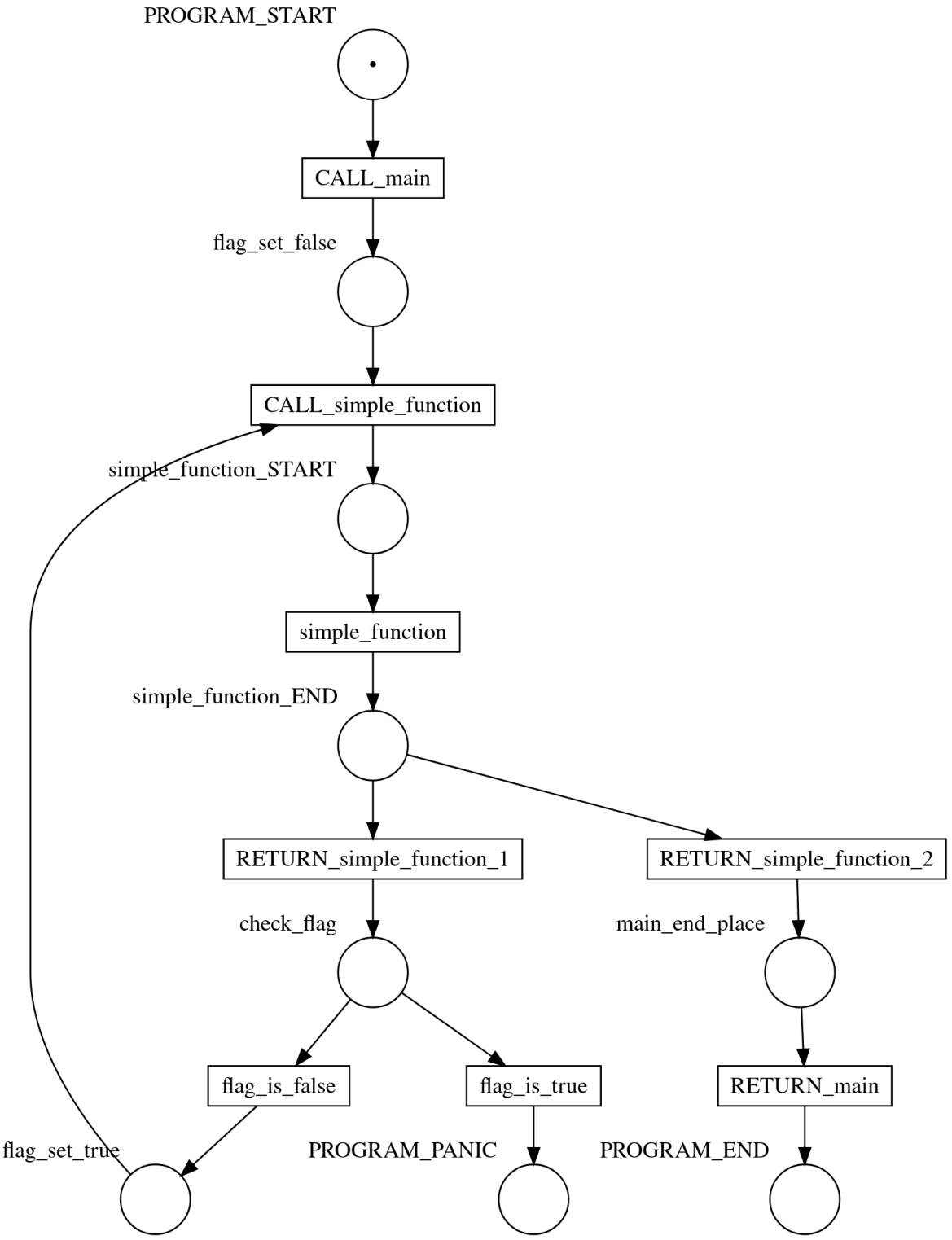

Figure 3.5: A second (also incorrect) PN for the code in Listing 3.5.

3.5.3 A feasible solution

Having observed the difficulties of modeling function calls, we turn our attention to the other approach to modeling function calls: Inlining the PN representation. Some of the lessons learned from the preceding subsection are:

- Creating a loop in the net where there is no loop in the original program opens the door to infinite sequences of transition firings. This could in turn break the *safety* property of the PN.
- As the token symbolizes the program counter, there must be only one token in the PN at any given time.
- The program state may change between function calls. Accordingly, separate places should model these states. Put differently, the state when calling a function the first time may not be the same as when calling the function a second time.

Fig. 3.6 introduces the inlining approach implemented in the tool. The PN therein is correct. It matches the structure of the Rust code more closely. It does not contain any loops nor it creates additional tokens when firing transitions, i.e. none of the transitions has two outputs. It is worth mentioning that the resulting PN is a state machine (Definition 8), as expected for a single-threaded program.

A significant advantage of the inlining approach is that every function call is unequivocally identified. This proves helpful when interpreting the output of the model checker or error messages during the translation of a given program. The use of an incremental non-negative id is arbitrary but convenient. Moreover, the accuracy of deadlock detection is increased because certain classes of extraneous states such as those in the PN shown in the previous section are not present. Minimizing the number of false positives plays an important role when considering which approach to implement for a tool that aims to be user-friendly and easy to set up.

One disadvantage mentioned earlier is that the size of the resulting net is larger. The exact penalty in the number of additional places and transitions depends on the frequency with which functions are reused on average in the codebase. It is reasonable to assume that functions are called from several places. However, certain optimizations can be applied, which can reduce the size of the net considerably, thus compensating for the effect of using inlining. These optimizations are discussed in detail in Chapter 7.

Lastly, an attentive reader may notice that the analysis of the PN in Fig. 3.6 leads to the conclusion that the program may call `panic!()` and terminate abruptly, which does not match the execution of the Rust program. This is correct but it is a limitation of low-level Petri nets that cannot be solved in the framework of the model and goes beyond the scope of this work. Chapter 7 explores the consequences of this restriction and proposes potential remedies.

Armed with these insights and knowledge about the design choices, we are now able to fully describe the implementation.

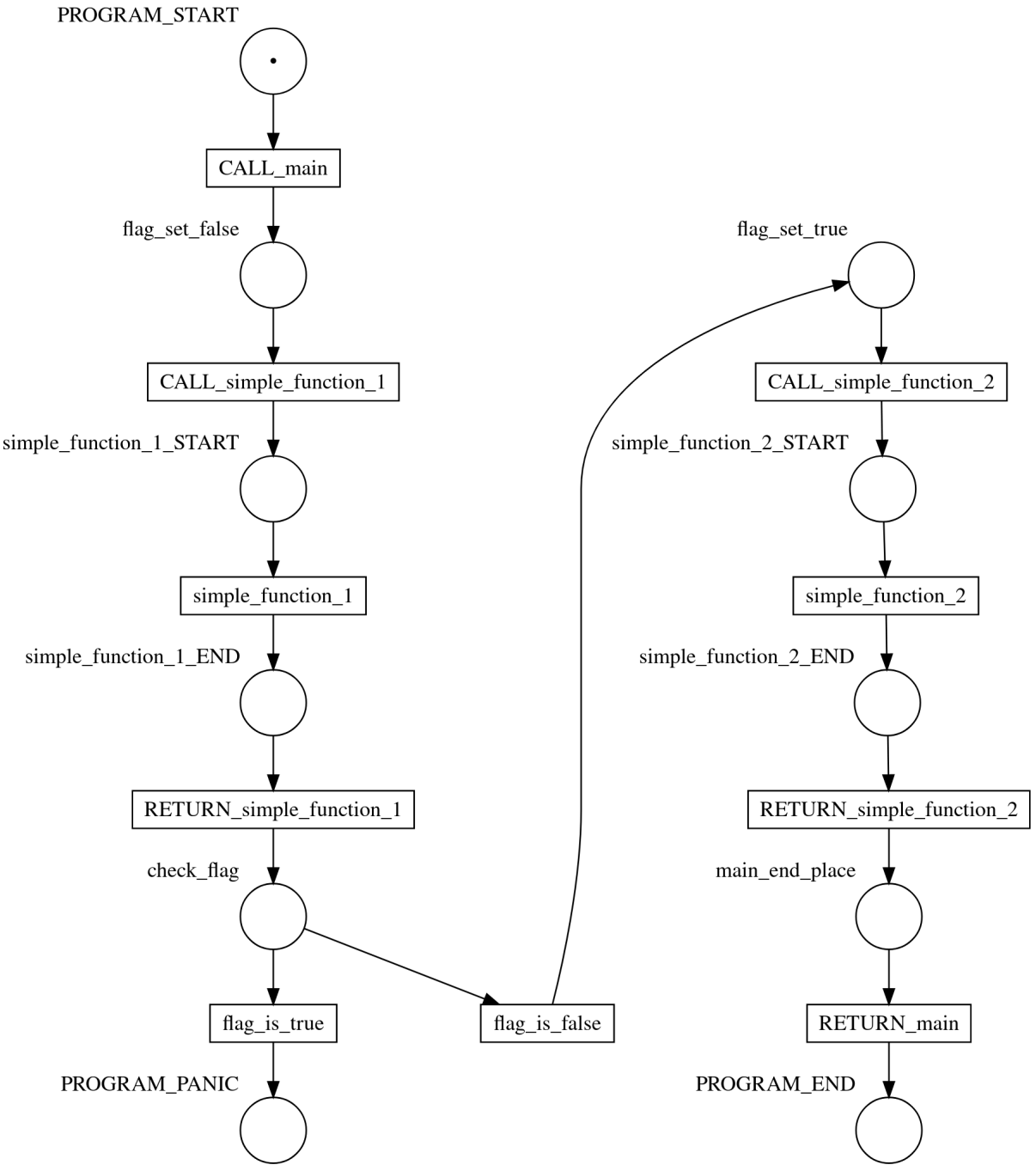

Figure 3.6: A correct PN for the code in Listing 3.5 using inlining.

Chapter 4

Implementation of the translation

4.1 Entry point for the translation

4.2 Function calls

4.3 Function memory

4.4 MIR function

4.4.1 Basic blocks

4.4.2 Statements

4.4.3 Terminators

4.5 Panic handling

4.6 Multithreading

4.7 Emulation of Rust synchronization primitives

4.7.1 Mutex (`std::sync::Mutex`)

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

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

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

Chapter 5

Testing the implementation

5.1 Generating the MIR

5.2 Visualizing the result

5.3 Unit tests

5.4 Integration tests

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

Chapter 6

Conclusions

Chapter 7

Future work

Chapter 8

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-PN), 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.

In [De Boer et al., 2013], a translation from a formal language to Petri nets for deadlock detection in the context of active objects and futures is presented. The formal language chosen is Concurrent Reflective Object-oriented Language (Creol). It is an object-oriented modeling

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

language designed for specifying distributed systems. In this paper, the program is made of asynchronously communicating active objects where futures are used to handle return values, which can be retrieved via a lock detaining `get` primitive (blocking) or a lock releasing `claim` primitive (non-blocking). After translating the program to a Petri net, reachability analysis is applied to detect deadlocks. This paper shows that a translation of asynchronous communication strategies to Petri nets with the goal of detecting deadlocks is also possible.

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.

[De Boer et al., 2013] De Boer, F. S., Bravetti, M., Grabe, I., Lee, M., Steffen, M., and Zavattaro, G. (2013). A petri net based analysis of deadlocks for active objects and futures. In *Formal Aspects of Component Software: 9th International Symposium, FACS 2012, Mountain View, CA, USA, September 12-14, 2012. Revised Selected Papers 9*, pages 110–127. Springer.

[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.

[Jannesari et al., 2009] Jannesari, A., Bao, K., Pankratius, V., and Tichy, W. F. (2009). Helgrind+: An efficient dynamic race detector. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–13. 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/.

[Klock, 2022] Klock, F. S. (2022). Contributing to Rust: Bootstrapping the Rust Compiler (rustc). https://www.youtube.com/watch?v=oG-JshUmkuA. Accessed on 2023-04-08.

[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.

[Matsakis, 2016] Matsakis, N. (2016). Introducing MIR. https://blog.rust-lang.org/2016/04/19/MIR.html. Accessed on 2023-04-14.

[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. https://github.com/Skasselbard/Granite/blob/master/doc/MasterThesis/main.pdf.

[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.

[Nelson, 2022] Nelson, J. (2022). RustConf 2022 - Bootstrapping: The once and future compiler. https://www.youtube.com/watch?v=oUIjG-y4zaA. Accessed on 2023-04-08.

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

[Rust Project, 2023b] Rust Project (2023b). The Rustonomicon. https://doc.rust-lang.org/nomicon/. Accessed on 2023-04-19.

[Rust Project, 2023c] Rust Project (2023c). The Unstable Book. https://doc.rust-lang.org/unstable-book/the-unstable-book.html. Accessed on 2023-04-14.

[Savage et al., 1997] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. (1997). Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411.

[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.

[Zhang and Liua, 2022] Zhang, K. and Liua, G. (2022). Automatically transform rust source to petri nets for checking deadlocks. *arXiv preprint arXiv:2212.02754*.