UNIVERSIDAD DE BUENOS AIRES

TESIS DE GRADO DE INGENIERÍA EN INFORMÁTICA

Compile-time Deadlock Detection in Rust using Petri Nets

Autor: Horacio Lisdero Scaffino

hlisdero@fi.uba.ar

Director: Ing. Pablo Andrés 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

CLI command-line interface

CPN Colored Petri nets

CPU central processing unit

Creol Concurrent Reflective Object-oriented Language

CTL* Computational Tree Logic*

DBMS Database management systems

FSM Finite-state machine

HIR High-Level Intermediate Representation

IR intermediate representation

ISA instruction set architecture

JIT just-in-time

LHS left-hand side

LIFO last in, first out

LoLA Low-Level Petri Net Analyzer

LTO link time optimization

MCC Model Checking Contest

MIR Mid-level Intermediate Representation

NT-PN Nondeterministic Transitioning Petri nets

OOM out-of-memory

OS operating system

P/T nets place/transition nets

PIPE2 Platform Independent Petri net Editor 2

PN Petri nets

PNML Petri Net Markup Language

RAG Resource Allocation Graph

RAII Resource Acquisition Is Initialization

RFCs Requests for Comments

RHS right-hand side

TAPAAL Tool for Verification of Timed-Arc Petri Nets

THIR Typed High-Level Intermediate Representation

TWF transaction-wait-for

UB Undefined Behavior

WASM WebAssembly

XML Extensible Markup Language

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/cargo-check-deadlock/
[2] https://github.com/hlisdero/netcrab

Chapter 1

Introduction

In order to fully understand the scope and context of this work, it is beneficial 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.

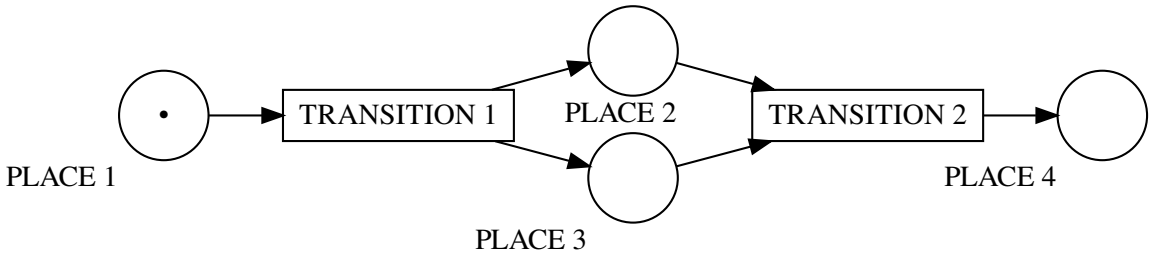

Figure 1.1: Example of a Petri net. PLACE 1 contains a token.

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.

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.

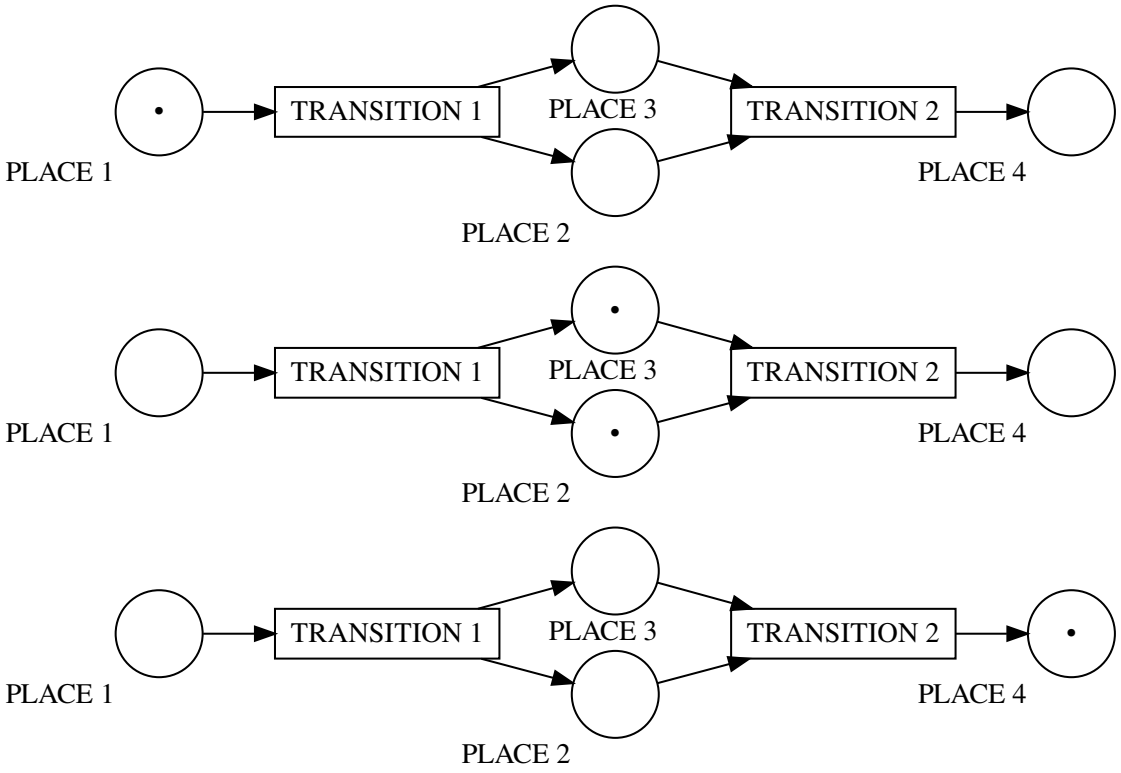

Figure 1.2: Example of transition firing: Transition 1 fires first, then transition 2 fires.

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 noteworthy 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 Online simulators

To familiarize oneself with the dynamics of Petri nets, it is useful to simulate some examples online since seeing a Petri net in action is clearer than any static explanation on paper. We have gathered some tools for this purpose to ease the burden on the reader.

- A simple simulator by Igor Kim can be found on `https://petri.hp102.ru/`. A tutorial video on Youtube and example nets are included in the tool.
- A complement to this is a series of interactive tutorials by Prof. Wil van der Aalst at the University of Hamburg. These tutorials are Adobe Flash Player files (with extension `.swf`) that modern web browsers cannot execute. Luckily, an online Flash emulator like the one found on `https://flashplayer.fullstacks.net/?kind=Flash_Emulator` can be used to upload the files and execute them.
- Another online Petri net editor and simulator is `http://www.biregal.com/`. The user can draw the net, add the tokens, and then manually fire transitions.

1.1.5 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.

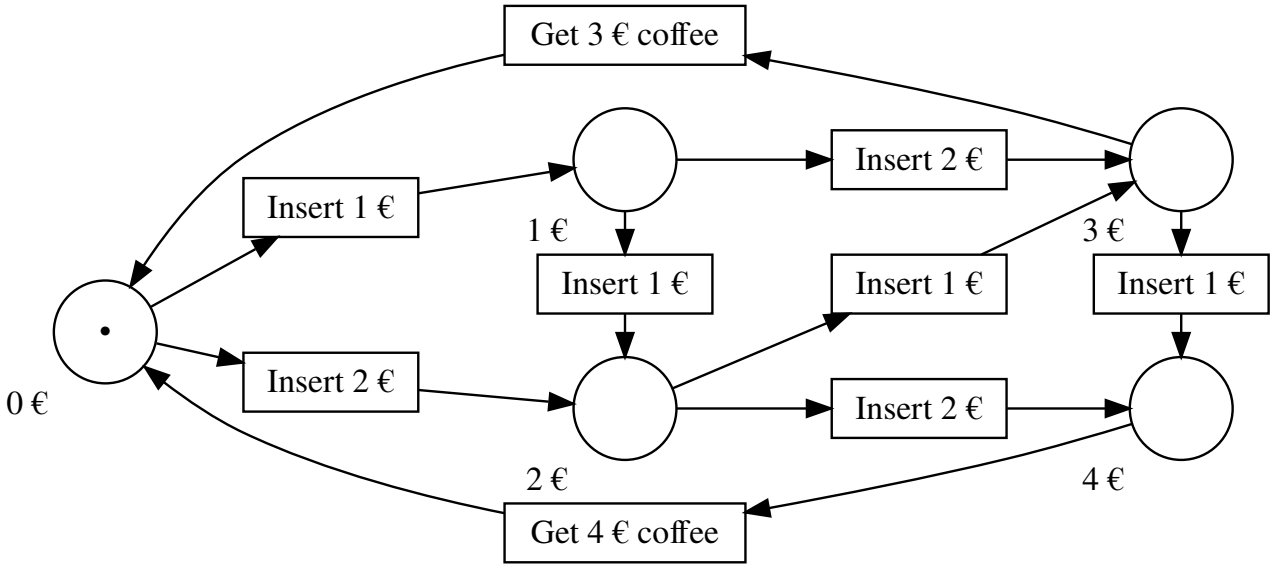

Figure 1.4: The Petri net for a coffee vending machine. It is equivalent to a state diagram.

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 leftmost place labeled "0€" 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 of Fig. 1.4, where the user must select which coin to insert first.

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.

The transition "Fork" will fire before "Task 1" and "Task 2" and that "Join" will only fire after

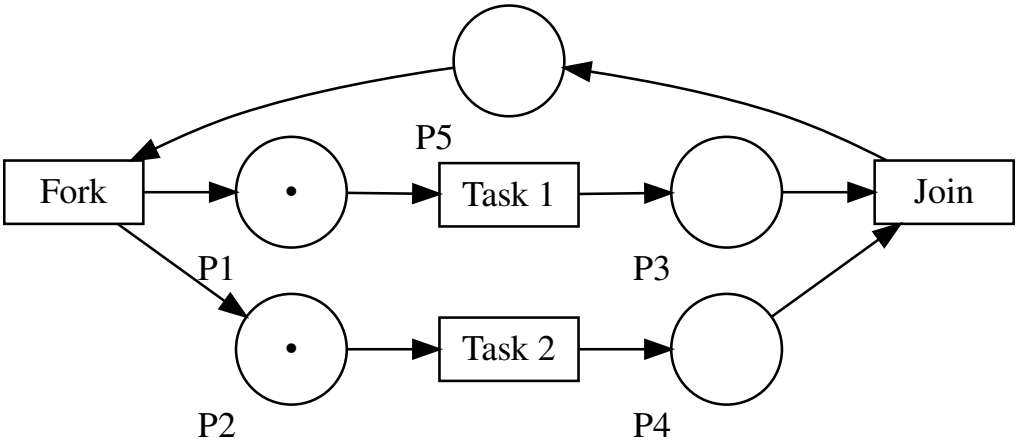

Figure 1.5: The Petri net depicting two parallel activities in a fork-join fashion.

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.

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.

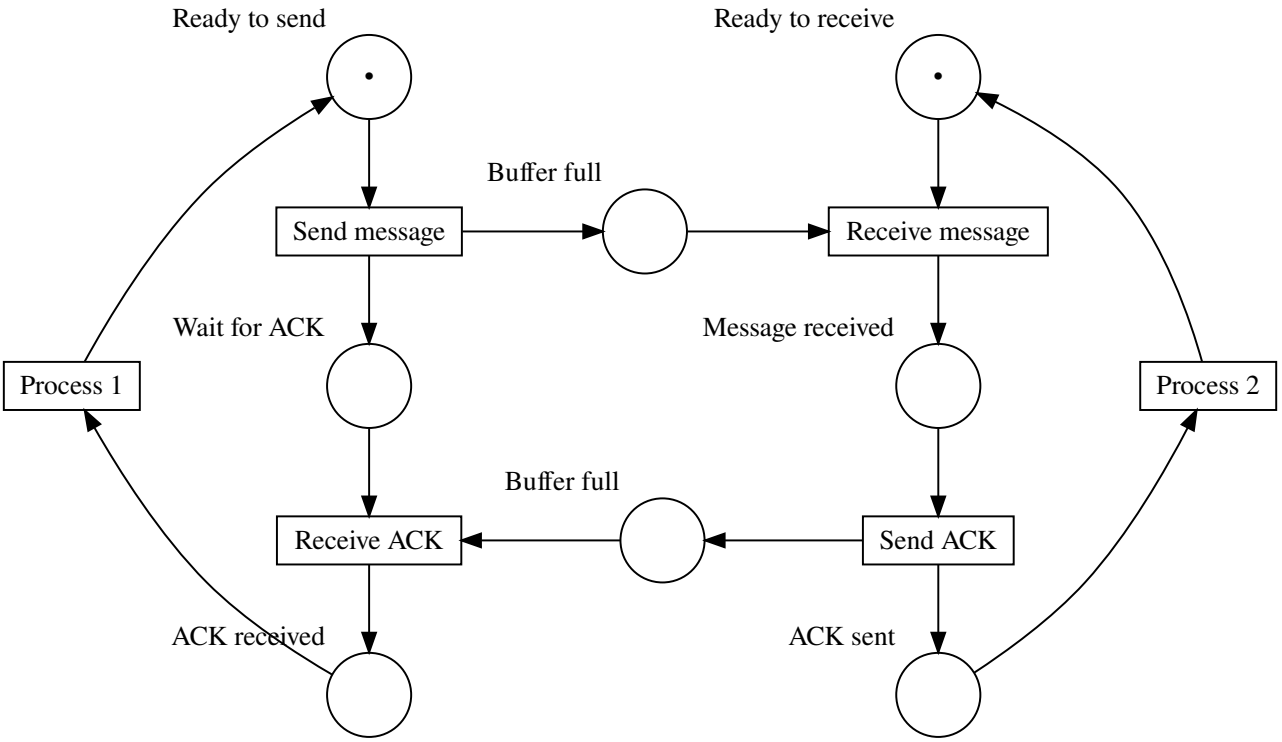

Figure 1.6: A simplified Petri net model of a communication protocol.

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.

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.

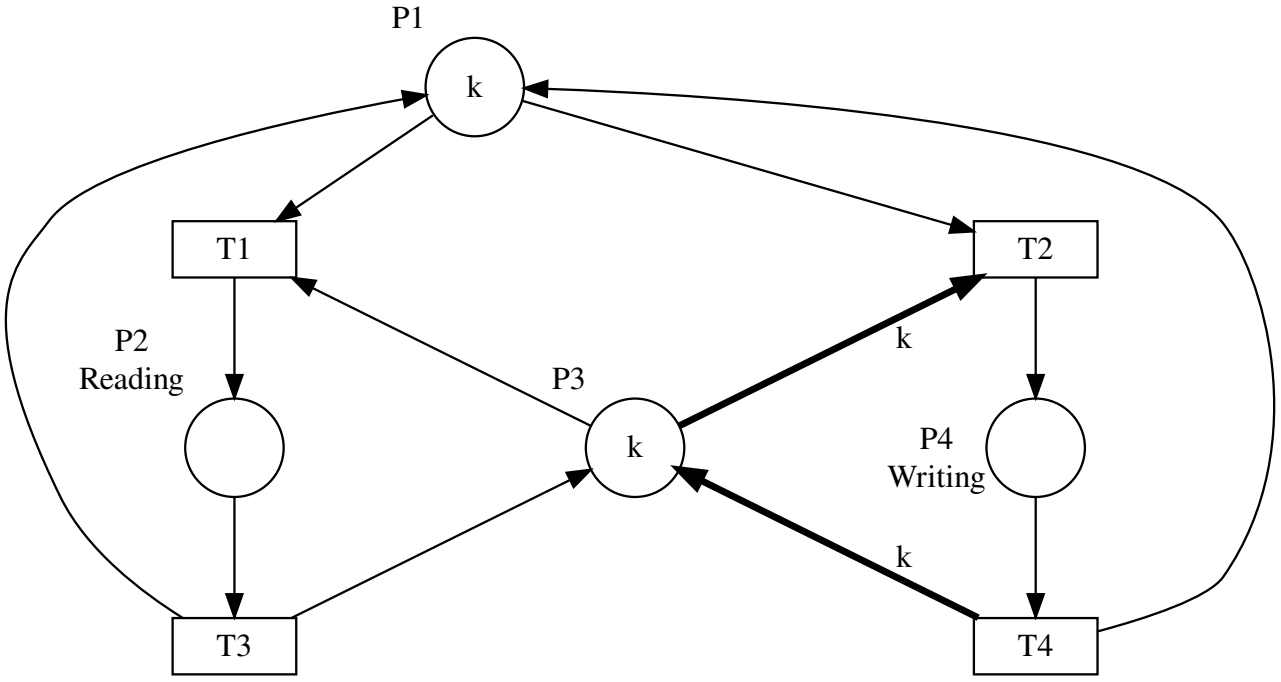

Figure 1.7: A Petri net system with k processes that either read or write.

1.1.6 Important properties

In this subsection, we will look at fundamental 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 cannot 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 applies 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 cannot fire for any marking reachable from M_0. The transition (or a set of transitions) cannot 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.

1.1.7 Reachability Analysis

Having introduced the reachability set $R(N, M_0)$ in Sec. 1.1.6, 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 reachability set, we define a new node in the reachability tree for each reachable marking, which results from firing each transition. An arc, labeled by the transition fired, leads from the initial marking (the root of the tree) to each of the new markings. After this first step (Fig. 1.9), the tree contains all markings that are immediately reachable from the initial marking.

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

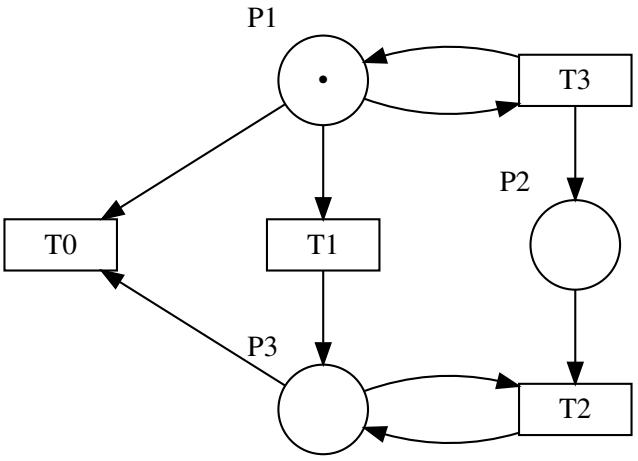

Figure 1.8: A marked Petri net for illustrating the construction of a reachability tree.

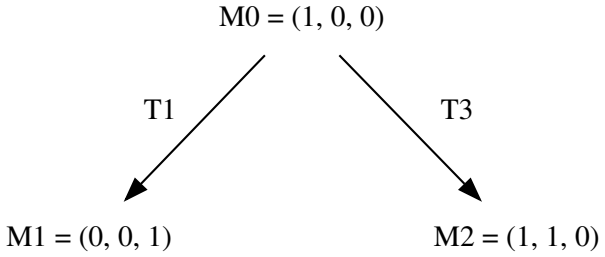

Figure 1.9: The first step building the reachability tree for the Petri net in Fig. 1.8.

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

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

Note that starting with marking $(0, 1, 1)$, only the transition T2 is enabled, which will lead to a marking $(0, 0, 1)$ that was already seen before. If instead we take $(1, 2, 0)$ we have again the same possibilities as starting from $(1, 1, 0)$. It is easy to see that the tree will continue to grow down that path. The tree is therefore infinite and this is because the net in Fig. 1.8 is not bounded. See Fig. 1.11 for the abbreviated final result.

The previously presented method enumerates the elements in the reachability set. Every marking in the reachability set will be produced and so for any Petri net with an infinite reachability set, i.e., an infinite number of possible states, the corresponding tree would also be infinite. Nonetheless, this opposite is not true. A Petri net with a finite reachability set can have an

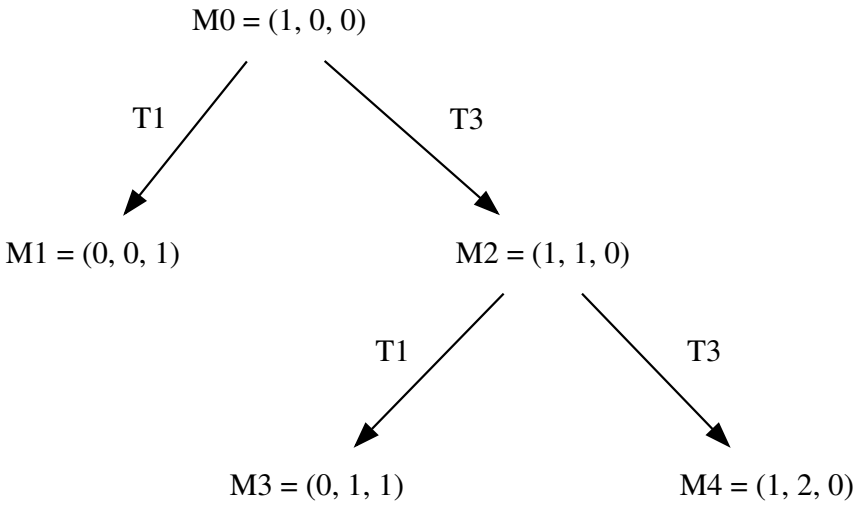

Figure 1.10: The second step building the reachability tree for the Petri net in Fig. 1.8.

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.

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,

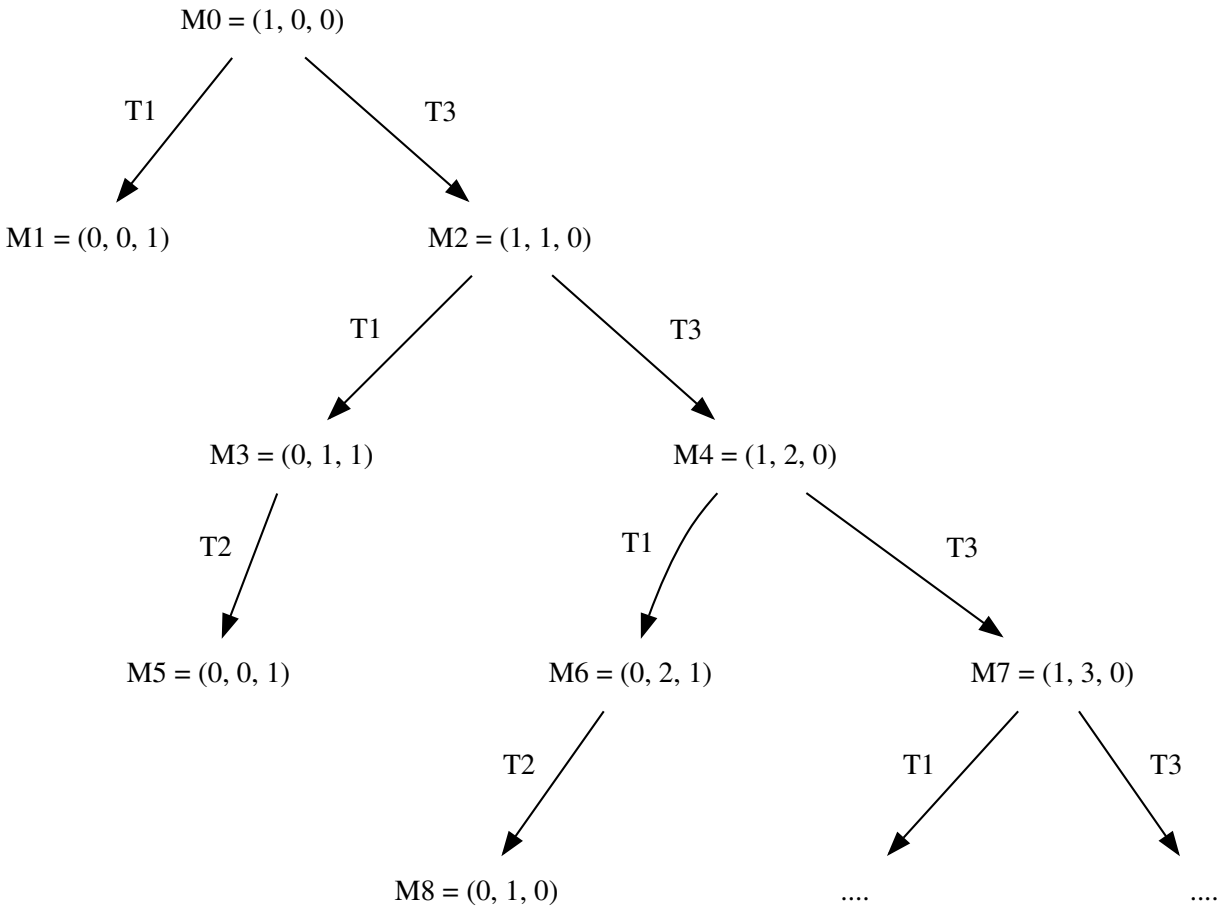

Figure 1.11: The infinite reachability tree for the Petri net in Fig. 1.8.

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.

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].

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

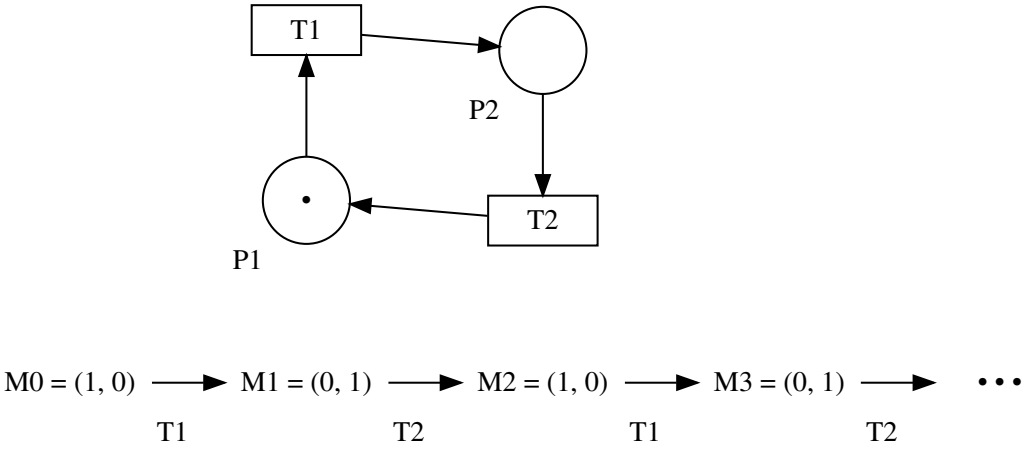

Figure 1.12: A simple Petri net with an infinite reachability tree.

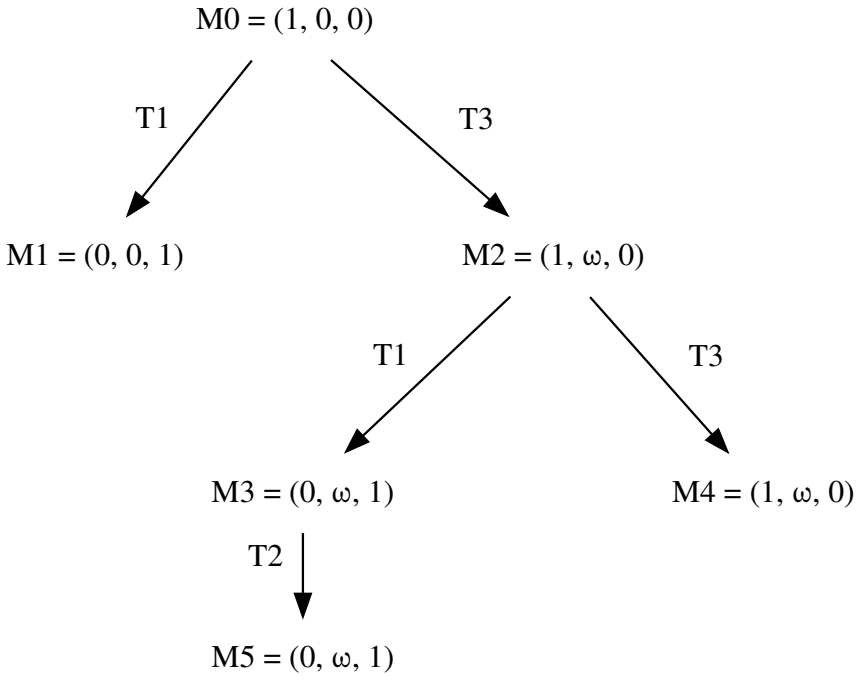

Figure 1.13: The finite reachability tree for the Petri net in Fig. 1.8.

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. 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 lan-

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

guage's development and suggest improvements. The documentation is also open-source and significant decisions are documented in the 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 an essential component of its ownership model, which is designed to ensure memory safety and prevent data races in concurrent code. The borrow checker analyzes Rust code at compile-time and enforces a set of rules to ensure that a program's memory is accessed safely and efficiently.

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

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

- No two mutable references to the same memory location can exist simultaneously. This prevents data races, where two threads try to modify the same memory location at the same time.
- Mutable references cannot exist at the same time as immutable references to the same memory location. This ensures that mutable and immutable references cannot be used simultaneously, preventing inconsistent reads and writes.
- References cannot 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 cannot 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.

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

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.

Error handling enforced by the compiler

Error handling is an essential aspect of programming and is typically addressed in the design of programming languages. The myriad of approaches can be summarized in two separate groups.

One group formed by languages such as C++, Java, or Python employs exceptions, utilizing `try` and `catch` blocks to handle exceptional conditions. When exceptions are thrown and not caught, the program terminates abruptly.

The other group is formed by languages like C or Go, among others, where the convention is to communicate an error either through the return value of functions or through a function parameter specifically dedicated to this purpose. The disadvantage is that the compiler does not enforce error-checking on the programmer, which may lead to error cases not being taken into account when adding new functionality.

Rust takes a different approach by promoting the notion that functions should ideally not fail and that the function signature should reflect if the function may return an error. Instead of exceptions or integer error codes, Rust functions that may encounter errors return a `std::result::Result`[6] type, which can hold either the result of the computation or a custom error type accompanied by a description of the error. *rustc* requires the programmer to write code for both cases and the language provides mechanisms to facilitate error handling [Klabnik and Nichols, 2023, Chap. 9.2].

In Rust, the focus lies on consistently managing the error case. Errors can be propagated to higher-level function calls until a consistent program state can be restored. However, there may be situations where recovery from an error state is not feasible. In such cases, the program can be instructed to panic, resulting in an abrupt and ungraceful shutdown, similar to an uncaught exception in other programming languages. During a panic, program execution is aborted, and the stack is unwound [Klabnik and Nichols, 2023, Chap. 9.1]. An error message containing details of the panic, e.g., the error message itself and its location, is generated. While panics can be caught by parent threads and in specific cases when the programmer so desires[7], they typically lead to the termination of the current program. This structured panic mechanism ensures that the compiler is aware of potential unrecoverable errors, enabling the generation of appropriate code to handle such cases.

[6] https://doc.rust-lang.org/std/result/
[7] https://doc.rust-lang.org/std/panic/fn.catch_unwind.html

Rust also provides a type `std::option::Option`[8] that represents either the presence of a value or the absence of it. The compiler again enforces discipline on the programmer to always handle the `None` case. Thus, Rust eliminates almost completely the need for a NULL pointer as found in other languages like C, C++, Java, Python, or Go.

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 representative 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].

[8] https://doc.rust-lang.org/std/option/

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 are 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 a representative 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[9].

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 noteworthy consideration 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

[9] https://valgrind.org/

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.

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, minimizing the occurrence of errors associated with thread or process synchronization holds significant importance 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 arises 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 would be advantageous to be able to detect and prevent deadlocks. They can happen 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.

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".

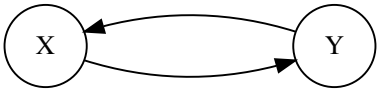

Figure 1.14: Example of a state graph with a cycle indicating a deadlock.

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 materialize 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 manifest. 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 a useful framework 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 cannot 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 arise. 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 take place 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 surface. 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 higher-priority 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[10],
- Rust[11]
- Python[12]
- Go[13]
- Java[14]

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 happens 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`.
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`.

[10] https://man7.org/linux/man-pages/man3/pthread_cond_init.3p.html
[11] https://doc.rust-lang.org/std/sync/struct.Condvar.html
[12] https://docs.python.org/3/library/threading.html
[13] https://pkg.go.dev/sync
[14] https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/concurrent/locks/Condition.html

```
// 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.

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 essential 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 unavoidable 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.

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.

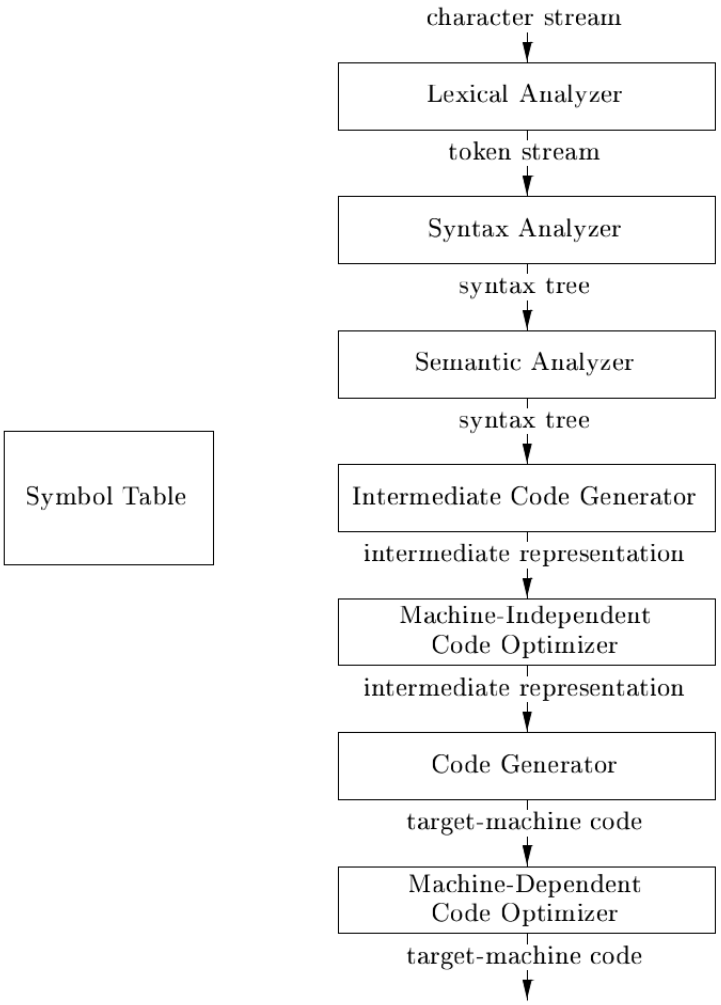

Figure 1.15: Phases of a compiler.

Model checking has become a widely-applied 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 relevant 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.

Afterward, 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.

As the next step, we explore the research community behind the Model Checking Contest (MCC) and the model checkers that participate in it to confirm the potential of these tools to analyze Petri net models of significant size. This is relevant since the model checker acts as the backend to the tool developed in this work.

Finally, three of the existing file formats for exchanging Petri nets are presented and their purpose in the context of this work is explained.

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

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

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.

[Kani Project, 2023] is another well-known tool for the formal verification of Rust code aimed at checking the `unsafe` blocks on a bit level. It offers a proof harness analogous to the test harness provided by Rust. Additionally, a plugin for cargo and VS Code is available.

As the documentation in the repository explains[2], Kani verifies (among others):

- Memory safety, e.g., null pointer dereferences
- User-specified assertions, i.e., `assert!(...)`
- The absence of panics, e.g., `unwrap()` on `None` values
- The absence of some types of unexpected behavior, e.g., arithmetic overflows

However, concurrent programs are currently out of scope[3]. The bottom line is that Kani offers an easy-to-use CLI and a proof harness that seamlessly integrate with the development process. It serves as an illustration of the capabilities of model checking in modern software development.

2.2 Deadlock detection using Petri nets

Deadlock prevention is one of the classic strategies to address this fundamental 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

[2] https://github.com/model-checking/kani
[3] https://model-checking.github.io/kani/rust-feature-support.html

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.

The source code for this library named "C2Petri" is disappointingly not found online, as the publications are fairly old.

In a more recent 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[4] as part

[4] https://github.com/Skasselbard/Granite

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)[5] 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*[6], GitHub, and GitLab revealed that there is, unfortunately, no well-maintained library.

Some Petri net simulators were found such as:

- pns[7]: Programmed in C. It does not offer the option to export the resulting network to a standard format.
- PetriSim[8]: An old DOS/PC simulator programmed in Bordland Pascal.
- WOLFGANG[9]: A Petri net editor in Java, maintained by the Department of Computer Science at the University of Freiburg, Germany.

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

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

[5] https://pipe2.sourceforge.net/
[6] https://crates.io/
[7] https://gitlab.com/porky11/pns
[8] https://staff.um.edu.mt/jskl1/petrisim/index.html
[9] https://github.com/iig-uni-freiburg/WOLFGANG
[10] https://docs.rs/petgraph/latest/petgraph/

- gamma[11]: 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.

2.4 Model checkers

The choice of an appropriate model checker is a vital part of this work, as it is the backend responsible for verifying the absence of deadlocks. Fortunately, several model checkers have been developed for analyzing Petri nets.

The Model Checking Contest (MCC) [Kordon et al., 2021] organized at the Sorbonne University in Paris is a great source for state-of-the-art model checkers. It is an annual competition where submitted model checkers are run on a series of Petri net models from academia and industry[12]. These models have been contributed by many individuals over a period of more than a decade and the total number of benchmarks has grown steadily as new models have been added.

Every year, the benchmarks include place/transition nets (P/T nets), i.e., Petri nets, and Colored Petri nets (CPN). The number of places in the nets can range from a dozen to more than 70000 and transitions can range from less than a hundred to more than a million. This highlights the broad applicability of the model checkers that take part in the competition.

The results are published on the official website (see for instance [Kordon et al., 2022]) and consist of:

1. a list of the qualified tools that participated,
2. the techniques implemented in each of the tools,
3. a section dedicated to detailing the experimental conditions under which the contest took place (the hardware used and the time necessary to complete the runs),
4. the results in the form of tables, plots, and even the execution logs of each program,
5. a list of winners for each category,
6. an analysis of the reliability of the tools based on the comparison of the results.

A brief look at the slides of the 2022 edition[13] reproduced in Fig. 2.1 illustrates that several

[11] https://github.com/metamolecular/gamma
[12] https://mcc.lip6.fr/2023/models.php
[13] https://mcc.lip6.fr/2022/pdf/MCC-PN2022.pdf

model checkers have demonstrated uninterrupted participation, with notable examples including:

- Tool for Verification of Timed-Arc Petri Nets (TAPAAL) maintained by the Aalborg University in Denmark[14], winner of a gold medal in the 2023 edition.
- Low-Level Petri Net Analyzer (LoLA) maintained by the University of Rostock in Germany[15], winner in previous editions and a base for other models.
- ITS-tools [Thierry Mieg, 2015], which was also combined with LoLA and won medals in 2020[16].

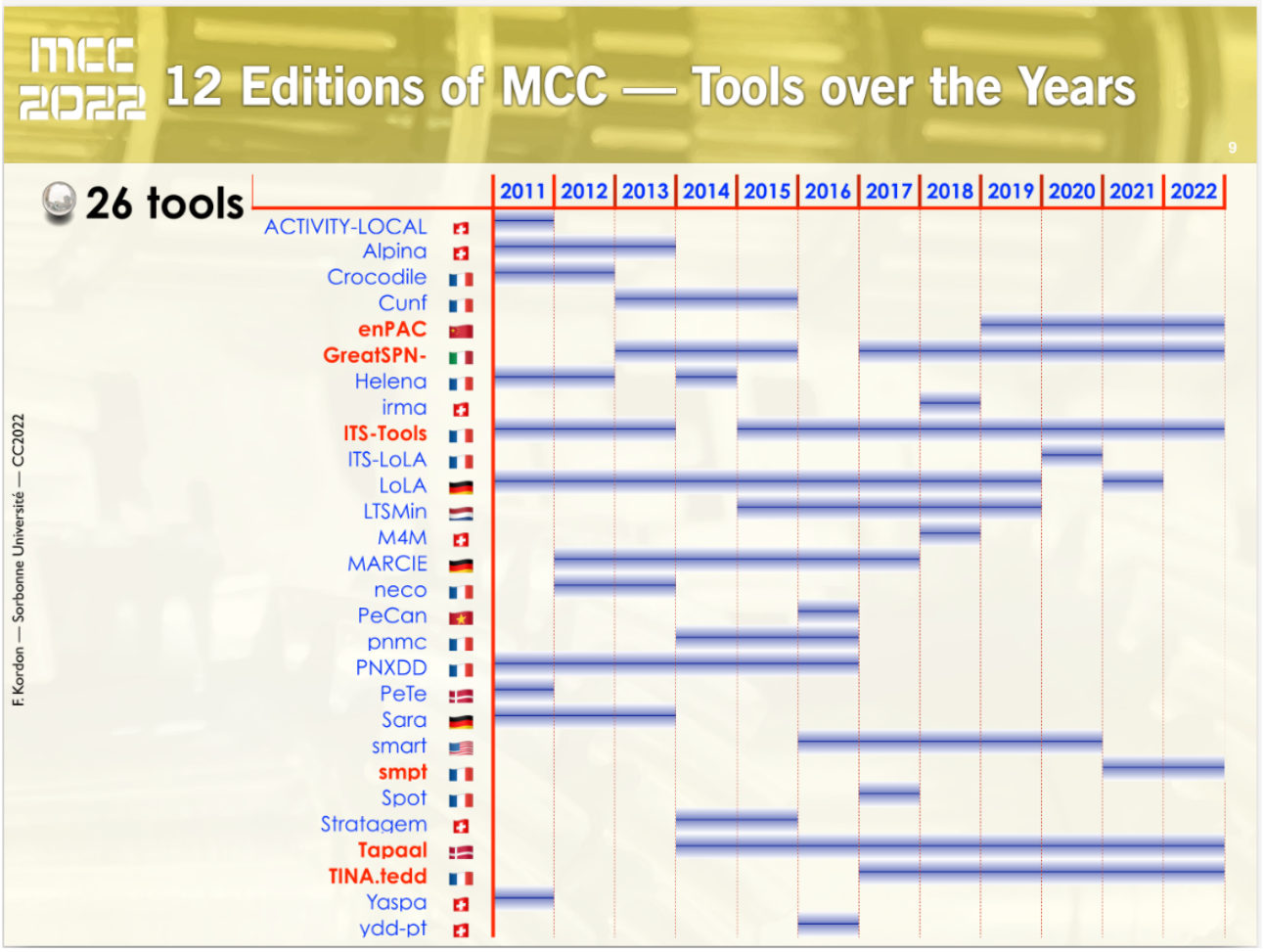

Figure 2.1: Model checker participation in the MCC over the years.

These observations collectively indicate the maturity and vibrancy of the model checker community. The establishment of a well-developed tool landscape, fostered by international col-

[14] https://www.tapaal.net/
[15] https://theo.informatik.uni-rostock.de/theo-forschung/tools/lola/
[16] https://github.com/yanntm/its-lola

laboration and the open-source dissemination of results, benchmarks, and techniques, presents a valuable opportunity for leveraging these tools within the realm of software development. Specifically, in the context of integrating them as backends for a language-specific translator that takes care of automating the Petri net model creation process. By capitalizing on the academic efforts invested in the model checkers, increased safety and reliability in software projects can be achieved.

2.5 Exchange file formats for Petri nets

As observed in the preceding chapter, Petri nets are a widely used tool for modeling software systems. However, due to the different classes of Petri nets (simple Petri nets, high-level Petri nets, timed Petri nets, stochastic Petri nets, colored Petri nets, to name a few), designing a standardized exchange file format compatible with all applications has proven challenging. One reason for this is that Petri nets can be implemented and represented in multiple ways, depending on the specific objectives, given that they are a type of graph.

In order to guarantee a certain degree of interoperability between the tool developed as part of this thesis and other existing and future tools, it is paramount to investigate which file formats would be more convenient to support. The aim is to support file formats that are suited to analysis as well as visualization, allowing for the possibility of extension to additional formats in the future, via a well-defined API in the Petri net library. A literature review led to three relevant file formats that are presented next.

2.5.1 Petri Net Markup Language

The Petri Net Markup Language (PNML)[17] is a standard file format designed for the exchange of Petri nets among different tools and software applications. Its development was initiated at the 'Meeting on XML/SGML based Interchange Formats for Petri Nets' held in Aarhus in June 2000 [Jüngel et al., 2000, Weber and Kindler, 2003], with the goal of providing a standardized and widely accepted format for Petri net models. PNML is an ISO standard consisting, as of 2023, of three parts:

- ISO/IEC 15909-1:2004[18] (and its latest revision ISO/IEC 15909-1:2019[19]) for concepts, definitions, and graphical notation.
- ISO/IEC 15909-2:2011[20] for defining a transfer format based on XML.
- ISO/IEC 15909-3:2021[21] for the extensions and structuring mechanisms.

[17] https://www.pnml.org/
[18] https://www.iso.org/standard/38225.html
[19] https://www.iso.org/standard/67235.html
[20] https://www.iso.org/standard/43538.html
[21] https://www.iso.org/standard/81504.html

It has become a de-facto standard for exchanging Petri net models across different tools and systems. It resulted from many years of hard work to unify the notation as discussed in [Hillah and Petrucci, 2010].

PNML has been designed to be a flexible and extensible format that can represent different classes of Petri nets, including simple Petri nets and high-level Petri nets. It is based on the Extensible Markup Language (XML) which makes it easy to read and parse by humans and machines alike. Additionally, PNML supports the use of metadata to provide additional information about the Petri net models, such as authorship, date of creation, and licensing information.

The development of PNML has significantly improved the interoperability and exchange of Petri net models among different tools and systems. Before the adoption of PNML, exchanging Petri net models was a challenging task, as different tools used proprietary formats that were often incompatible with each other. PNML has greatly simplified this process, enabling researchers and practitioners to share and collaborate on Petri net models with ease. Its use has also facilitated the development of new tools and software applications for Petri nets, as it provides a standard format that can be easily parsed and processed by different systems. For instance, it is the format used in [Zhang and Liua, 2022] and it is supported in [Meyer, 2020].

2.5.2 GraphViz DOT format

The DOT format is a graph description language used for creating visual representations of graphs and networks, which is part of the open-source GraphViz suite[22]. It was created in the early 1990s at AT&T Labs Research as a simple, concise, and human-readable language for describing graphs. The GraphViz suite provides several tools for working with DOT files, including the ability to automatically generate layouts for complex graphs and to export visualizations in a range of formats, including PNG, PDF, and SVG.

DOT can be used to represent Petri nets in a graphical format, which makes it easy to visualize the structure and behavior of the system being modeled. It is particularly useful for visualizing large Petri nets, as the user can navigate through the image to gain an understanding of how the tokens flow through the net.

The DOT format is text-based and easy to use, making it a popular choice for generating visual representations of graphs. This simplicity also means that DOT files can be easily generated by programs and can be read by a wide range of software tools, which is essential for interoperability. Additionally, DOT allows for the specification of various graph properties, such as node shapes, colors, and styles [Gansner et al., 2015], which can be used to represent different aspects of a Petri net, such as places, transitions, and arcs. This flexibility in specifying visual properties also enables users to customize the visualization to their needs and to highlight particular features of the Petri net that are relevant to their analysis.

[22] https://graphviz.org/

2.5.3 LoLA - Low-Level Petri Net Analyzer

Low-Level Petri Net Analyzer (LoLA) [Schmidt, 2000] is a state-of-the-art model checker whose development started in 1998 at the Humboldt University of Berlin. It is currently maintained by the University of Rostock and is published under the GNU Affero General Public License. LoLA is a tool that can check if a system satisfies a given property expressed in Computational Tree Logic* (CTL*). Its particular strength is the evaluation of simple properties such as deadlock freedom or reachability as stated on the website.

This is the model checker used in [Meyer, 2020] and in this work. Therefore, it is necessary to implement the file format required by the tool. Examples are presented in Sec. 5.4.

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 indispensable 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, 2023d] 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 happened 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 Sec. 7.1 and 7.2.

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[8] 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
[8] https://rust-lang.github.io/rfcs/1211-mir.html

```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
    }
```

```
31
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[9] 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
```

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

```
13          _3 = args() -> bb1;          // scope 0 at src/main.rs:2:11: 2:27
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.

3.4.1 MIR components

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.

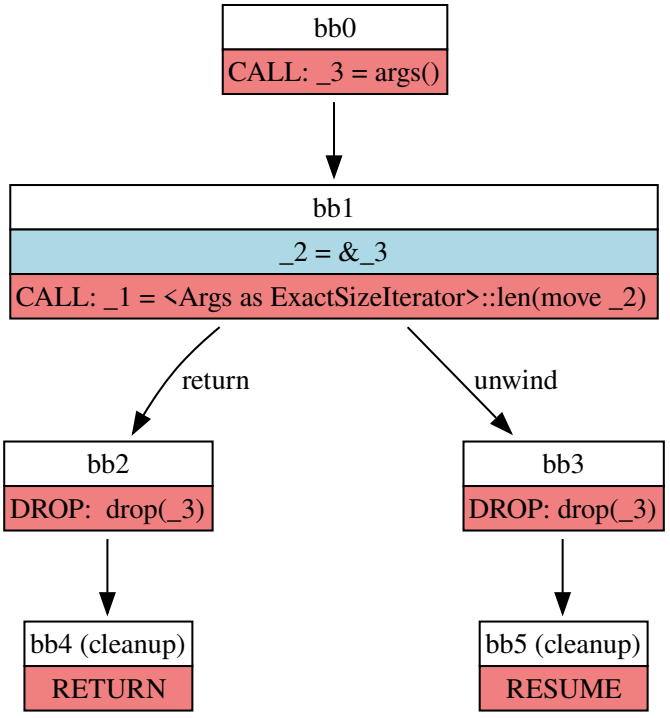

Figure 3.1: The control flow graph representation of the MIR shown in Listing 3.2.

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. A failure 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[10] 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[11]. 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.

3.4.2 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.

[10] https://doc.rust-lang.org/stable/reference/destructors.html
[11] 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"[12], 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.

[12] 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[13]

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 CPU 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.

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

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.

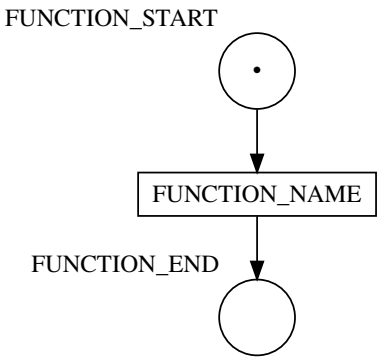

Figure 3.2: The simplest Petri net model for a function call.

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.

```
1 fn simple_function() {}
2
3 pub fn main() {
4     for n in 0..5 {
5         simple_function();
6     }
7 }
```

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

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.

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.

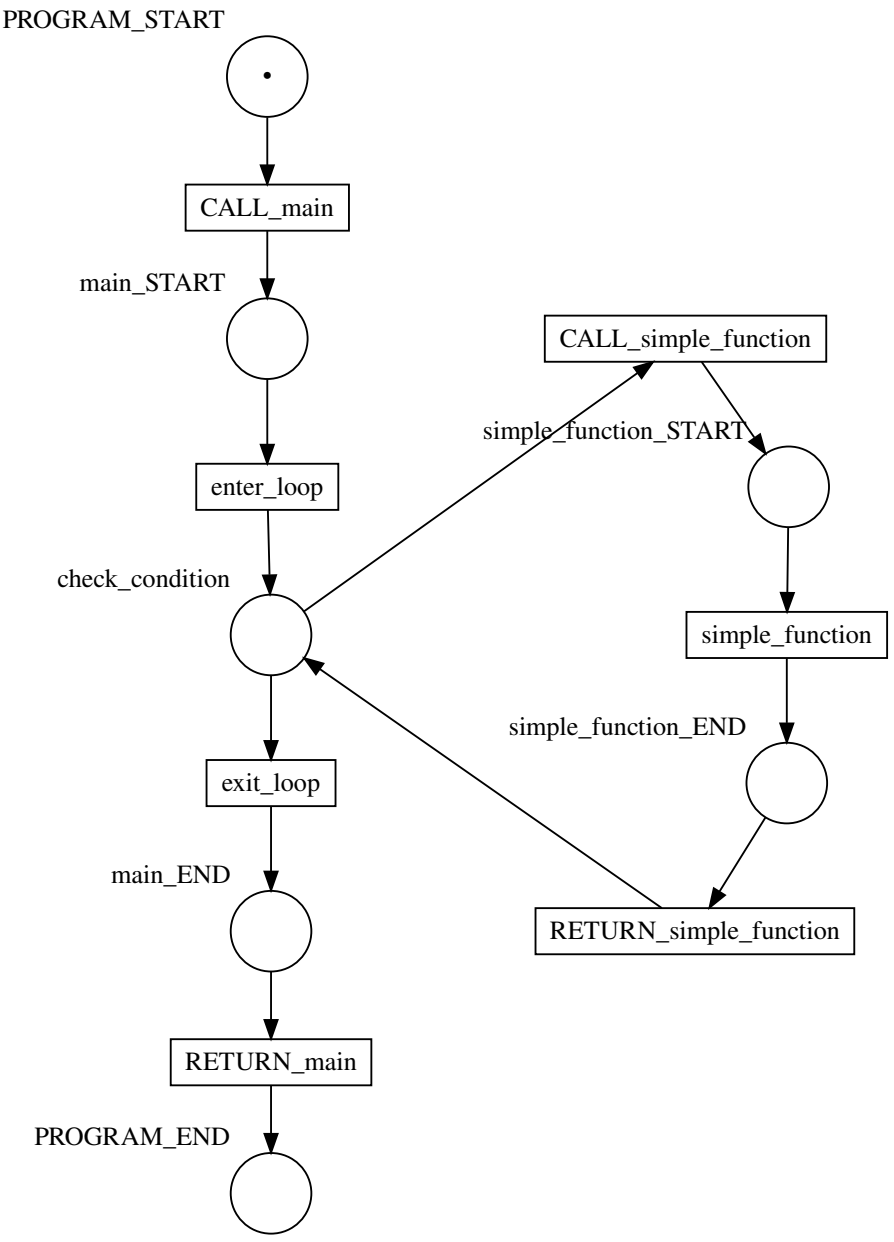

Figure 3.3: A possible PN for the code in Listing 3.4 applying the model of Fig. 3.2.

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`

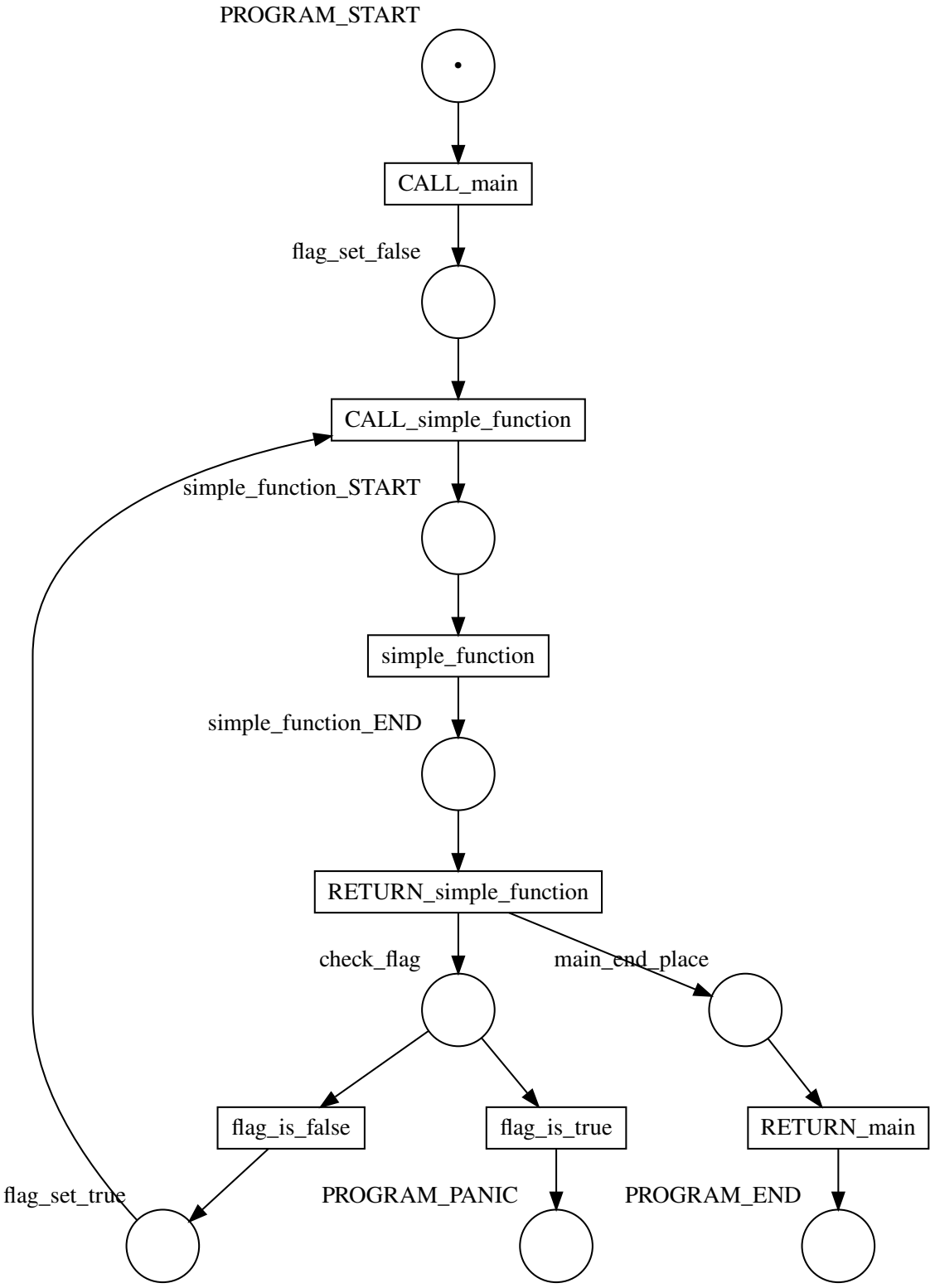

Figure 3.4: A first (incorrect) PN for the code in Listing 3.5.

```
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.

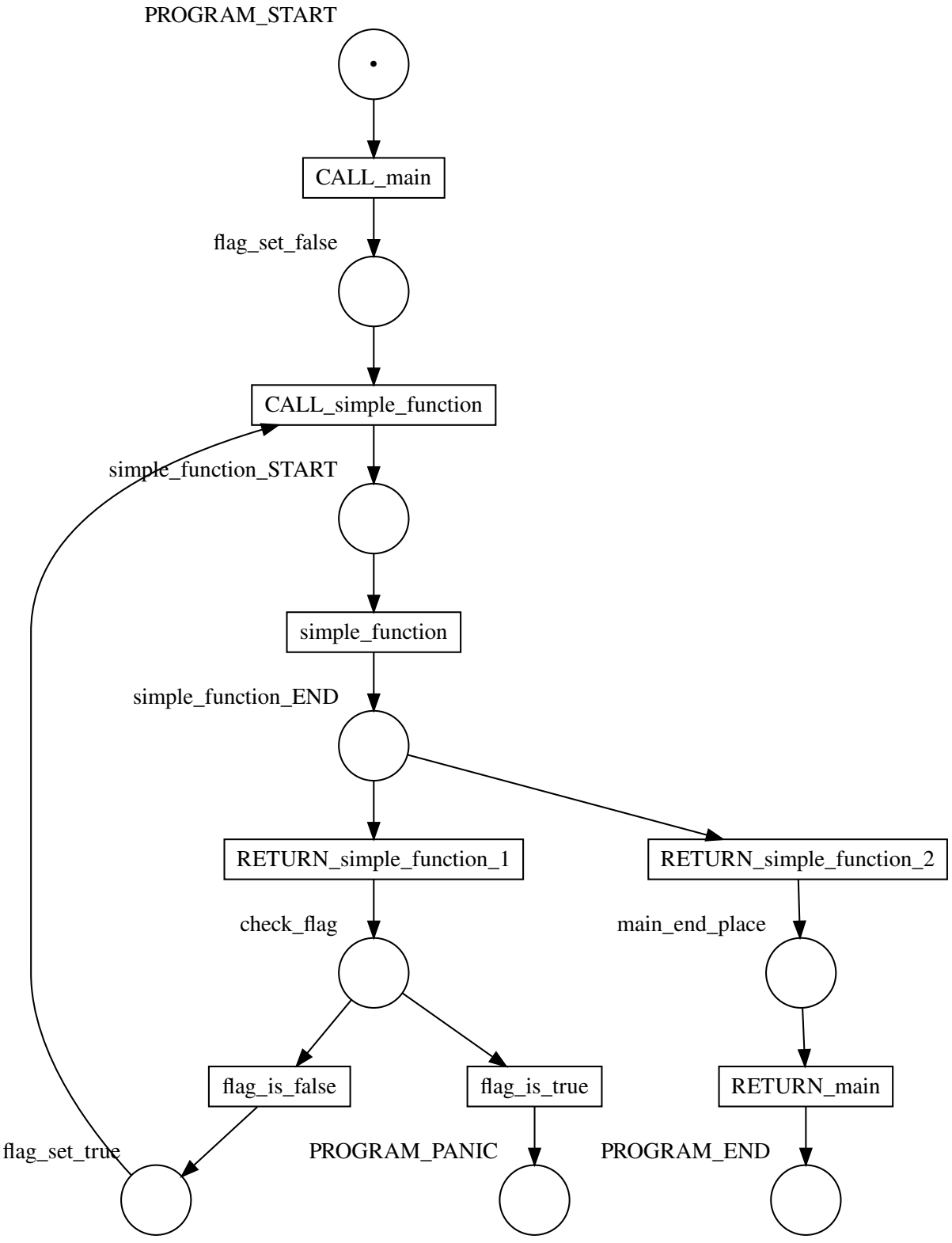

Figure 3.5: A second (also incorrect) PN for the code in Listing 3.5.

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. This was not the case for Fig. 3.4 and 3.5.

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 Sec. 7.1 and 7.2.

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. Sec. 7.6 explores the consequences of this restriction and proposes potential remedies.

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

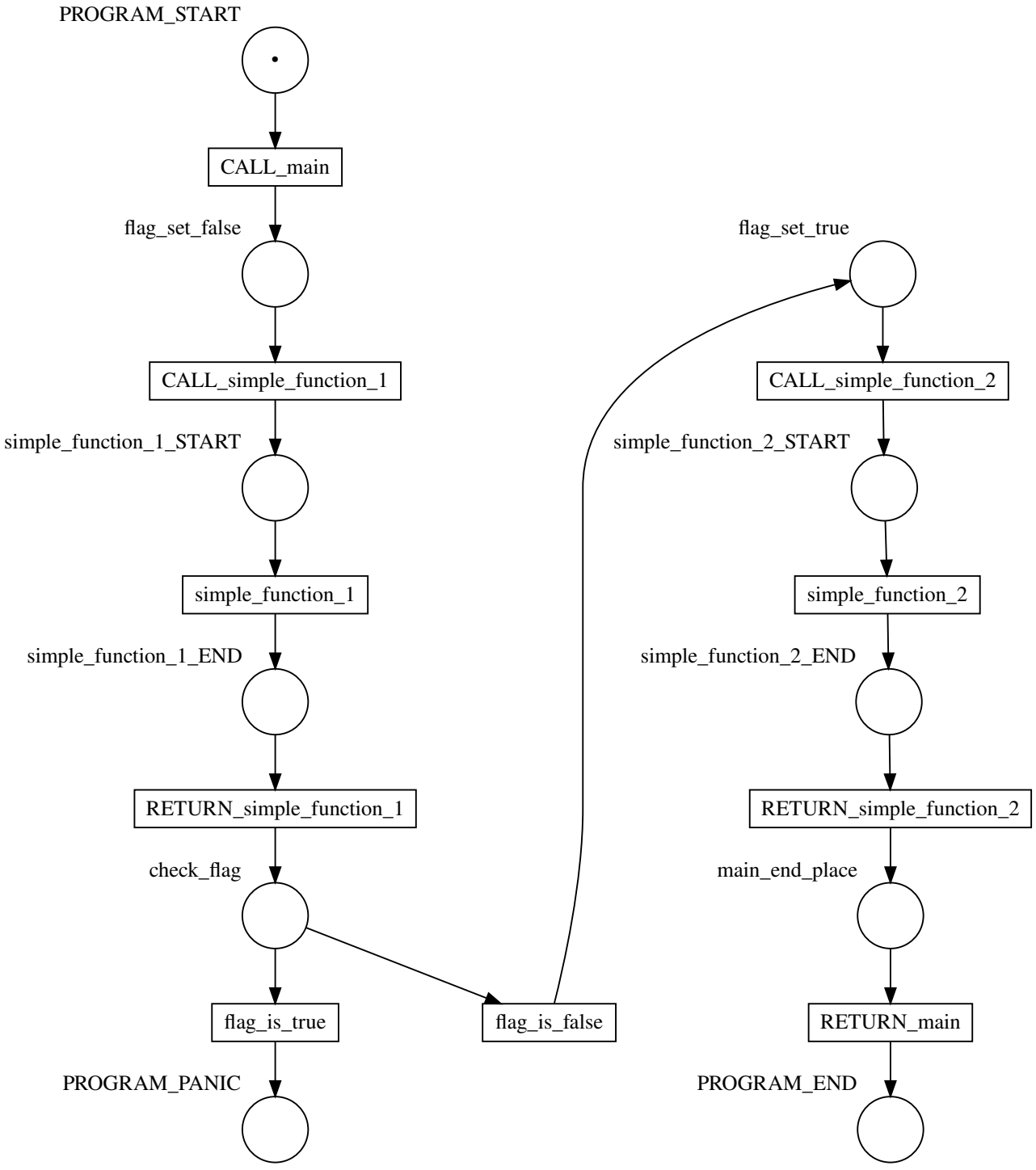

Figure 3.6: A correct PN for the code in Listing 3.5 using inlining.

Chapter 4

Implementation of the translation

This chapter is dedicated to exploring the implementation details of the deadlock detection tool. Its purpose is to provide a high-level view of the code and the data structures. The most important implementation decisions made throughout the development process are examined as well.

In the subsequent sections, we will describe the central components of the deadlock detection tool, including the internal representation of the call stack, the function memory model, and the translation of every constituent of a MIR function.

Later on, a significant portion of the discussion is devoted to explaining the support of multithreading and the modeling of synchronization primitives as Petri nets. Its implementation required careful design considerations to ensure correctness and efficiency.

The tool currently supports the following structures from the Rust standard library to synchronize access to shared resources and provide communication among threads:

- mutexes (`std::sync::Mutex`[1]),
- condition variables (`std::sync::Condvar`[2]),
- atomic reference counters (`std::sync::Arc`[3]).

While the main details are covered, this chapter is not intended to serve as a substitute for the code documentation. The code documentation in the form of comments, unit tests, and integration tests provides comprehensive information on the low-level specifics and usage of the tool. As stated before, the repository is publicly available on GitHub[4][5].

[1] https://doc.rust-lang.org/std/sync/struct.Mutex.html
[2] https://doc.rust-lang.org/std/sync/struct.Condvar.html
[3] https://doc.rust-lang.org/std/sync/struct.Arc.html
[4] https://github.com/hlisdero/cargo-check-deadlock
[5] https://github.com/hlisdero/netcrab

4.1 Initial considerations

4.1.1 Basic places of a Rust program

The basic Petri net model for a Rust program generated by the tool can be seen in Fig. 4.1. The place labeled `PROGRAM_START` contains a token and represents the initial state of the Rust program. This token will "move" from statement to statement and can thus be interpreted as the program counter of the CPU.

Correspondingly, the place labeled `PROGRAM_END` models the end state of the program after normal program termination, i.e., returning from the `main` function, regardless of the specific exit code. In other words, a main function that returns an error code because of invalid parameters or an internal program error is still considered a "normal" program termination. In other instances, however, the program may never reach this state if `main` never returns. These are known in Rust as "diverging functions"[6] and are supported by the tool.

Lastly, the place labeled `PROGRAM_PANIC` models the *abnormal* program termination, which happens when the program calls the `panic!()` macro.

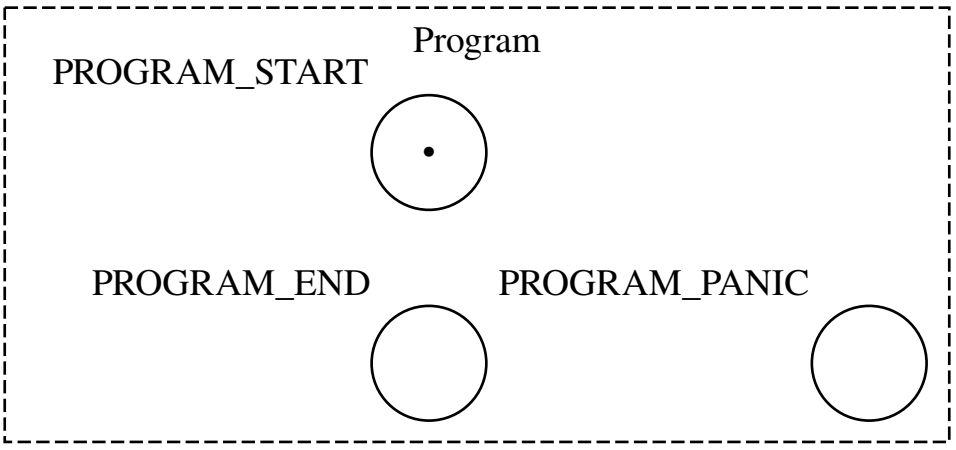

Figure 4.1: Basic places in every Rust program.

Two reasons for considering a separate panic end-state place can be argued. First, it is helpful for formal verification to distinguish the panic case from the normal termination case. A program may panic when detecting a possible violation of its memory safety guarantees. This is in most circumstances a wiser choice than simply ignoring the error and continuing. Therefore, such programs should not be flagged in principle as erroneous or defective but it is advisable to record the end state for troubleshooting and debugging purposes. Second, even if the user's code does not resort to `panic!()` as an error-handling mechanism, numerous functions

[6] https://doc.rust-lang.org/rust-by-example/fn/diverging.html

in the Rust standard library may panic under extraordinary circumstances, e.g., due to out-of-memory (OOM) or hardware errors, or when the OS fails to allocate a new thread, mutex, etc. Consequently, it is essential to capture this eventual failure in the PN model.

There is one last subtle point that needs to be addressed. The program's start place is not as trivial as it seems. Although the `main` function is typically perceived as the first function to be executed, this is in reality not the case. Instead, Rust programs have a runtime that executes before the `main` function is called, in which language-specific features and static memory are initialized. One usually hears of interpreted languages such as Java or Python having a runtime but low-level languages such as Rust or C have a small runtime as well. It is simply thinner and less sophisticated. For interested readers, a guided tour of the journey before `main` was presented recently at a Rust conference [Levick, 2022].

Considering this, we are faced with the question of whether to include this runtime in the PN translation. On one hand, the runtime code is indeed part of the binary executed by the CPU. Nonetheless, it is platform-dependent code (the runtime is slightly different for every OS) and independent of the program's semantics, i.e., of the specific meaning of the program the user wrote. Since the user does not have any influence on this part of the binary, synchronization problems cannot be attributed to him/her. As such, this code does not add value to the translation and can be safely abstracted away, reducing in the process the size of the PN. In conclusion, the decision is to skip the runtime code; the translation starts at the `main` function.

4.1.2 Argument passing and entering the query

The tool is designed around a simple command-line interface (CLI). After parsing the command-line arguments using the well-known library `clap` library[7], the program enters a query to the *rustc* compiler to start the translation process. The majority of the work from that point on is coordinated by the struct of type `Translator`[8].

The query system was described briefly in Sec. 3.2. Two examples of the use of this mechanism are provided in the documentation[9][10]. They have proved extremely useful as a starting point, since they provide an excellent short working example of how to interact with *rustc*. In simpler terms, they are the "Hello, World!" of working side-by-side with the Rust compiler.

4.1.3 Compilation requirements

As briefly mentioned in Sec. 3.2.2, the tool must be compiled with the nightly version of *rustc* to access its internal crates and modules. The decisive section in the file *lib.rs*[11] is depicted in Listing 4.1.

[7] https://docs.rs/clap/latest/clap/
[8] https://github.com/hlisdero/cargo-check-deadlock/blob/main/src/translator.rs
[9] https://rustc-dev-guide.rust-lang.org/rustc-driver-interacting-with-the-ast.html
[10] https://rustc-dev-guide.rust-lang.org/rustc-driver-getting-diagnostics.html
[11] https://github.com/hlisdero/cargo-check-deadlock/blob/main/src/lib.rs

```
13 // This feature gate is necessary to access the internal crates of the compiler.
14 // It has existed for a long time and since the compiler internals will never be
   ↪  stabilized,
15 // the situation will probably stay like this.
16 // <https://doc.rust-lang.org/unstable-book/language-features/rustc-private.html>
17 #![feature(rustc_private)]
18
19 // Compiler crates need to be imported in this way because they are not published on
   ↪  crates.io.
20 // These crates are only available when using the nightly toolchain.
21 // It suffices to declare them once to use their types and methods in the whole crate.
22 extern crate rustc_ast_pretty;
23 extern crate rustc_const_eval;
24 extern crate rustc_driver;
25 extern crate rustc_error_codes;
26 extern crate rustc_errors;
27 extern crate rustc_hash;
28 extern crate rustc_hir;
29 extern crate rustc_interface;
30 extern crate rustc_middle;
31 extern crate rustc_session;
32 extern crate rustc_span;
```

Listing 4.1: Excerpt of the file *lib.rs* showcasing how to use the *rustc* internals.

The `rustc_private` is a feature flag that controls access to the compiler's private crates. These crates are not installed by default when installing the Rust toolchain using *rustup*[12]. Hence, it is necessary to install the additional components *rustc-dev*, *rust-src*, and *llvm-tools-preview*. The purpose of each component is detailed in [Rust Project, 2023c]. Straightforward instructions to set up a development environment are also found in the `README`[13] of the repository.

To the best knowledge of this author, an alternative method of accessing the internals of the Rust compiler does not exist. Tools such as Clippy[14] or Kani [15], or kernels like Redox[16] and RustyHermit[17] use this mechanism as well.

[12] https://rustup.rs/
[13] https://github.com/hlisdero/cargo-check-deadlock/blob/main/README.md
[14] https://github.com/rust-lang/rust-clippy/blob/master/rust-toolchain
[15] https://github.com/model-checking/kani/blob/main/rust-toolchain.toml
[16] https://gitlab.redox-os.org/redox-os/redox/-/blob/master/rust-toolchain.toml
[17] https://github.com/hermitcore/rusty-hermit/blob/master/rust-toolchain.toml

4.2 Function calls

4.2.1 The call stack

A Rust program is composed, as in other programming languages, of functions. The program begins (except for the caveats seen in Sec. 4.1.1) with a call to the `main` function, which then may call other functions. It should be emphasized that function calls may be placed at any point within the code. A function can be called from another function or even from within itself, resulting in recursive calls.

Function calls are stored in memory in a data structure called the *call stack*. When a function is called in Rust, it gets pushed onto the call stack, creating a new stack frame. A stack frame contains important information such as the function's local variables, arguments, and the return address indicating where the program should resume once the function finishes its execution.

The call stack operates based on the principle of last in, first out (LIFO). As functions are called, each new stack frame is placed on top of the previous one. This allows the program to execute the most recently called function first. Once a function completes its execution, it is popped off the stack, and the program continues from the point where it left off in the previous function.

Hence, the call stack plays an essential role in managing function calls and returns, since it keeps track of the flow of function calls and maintains the necessary information for the program to return to the correct execution point after a function completes its task.

For the same reasons, mirroring the call stack in the translator is the most suitable approach for tracking function calls to be translated because it aligns with the logical flow of program execution. As functions are translated, they are pushed and popped from the call stack of the `Translator`, reflecting the order in which they are called at runtime. This enables us to handle nested function invocations and follow the control flow from one function to the other during the translation process.

4.2.2 MIR functions

In the implementation, the `Translator` has a stack that supports the usual operations `push`, `pop`, and `peek`. This stack stores structures of type `MirFunction`[18]. Later, we will see that not all functions are translated as MIR functions since not all functions have a representation in MIR and, in other cases, it is convenient to handle them differently. Nevertheless, MIR functions are the "common case" in the translation process, the default case for the majority of user-defined functions.

The available interface provided by *rustc* allows for querying the MIR body of only one function

[18] https://github.com/hlisdero/cargo-check-deadlock/blob/main/src/translator/mir_function.rs

at a time, which can be done using the `optimized_mir`[19] method. This implies that it is not possible to get the MIR of the whole program initially and the translator must obtain the MIR from each function as it reaches them in the code. But how to identify each function? It is known from experience that functions in distinct modules may have the same name, making the name unsuitable as an identifier. Luckily, this problem is already solved in the compiler. The functions are uniquely identified by the compiler type `rustc_hir::def_id::DefId`[20]. This ID is valid for the crate currently being compiled and it is already present in the HIR. The high-level algorithm can be described as follows.

When the translation starts:

1. Query the id of the entry point of the program (the `main` function).
2. Create a `MirFunction` with the necessary information.
3. Push it to the stack.
4. If necessary, modify the MIR function contents using `peek`.
5. Translate the top of the call stack.
6. When `main` finishes, remove it (`pop`) from the call stack.

When a terminator of type "call" (see Sec. 3.4.1) is encountered:

1. Query the id of the called function.
2. Create a `MirFunction` with the necessary information.
3. Push it to the stack.
4. If necessary, modify the MIR function contents using `peek`.
5. Translate the top of the call stack.
6. When the function finishes, remove it (`pop`) from the call stack.

As seen, the approach is consistent for every MIR function and thus is easier to implement.

The use of a call stack in the translation process enables context switching between MIR functions and facilitates the ability to return to the specific basic block from which a function was called. This allows for the translation of the program to be performed function by function, in a linear fashion, ensuring that the structure and order of the original program are maintained.

However, employing the call stack approach does come with certain limitations. First, if the same function is called multiple times within the program, it will be translated multiple times as well. This is related to the inlining strategy elaborated in Sec. 3.5. Although this can

[19] https://doc.rust-lang.org/stable/nightly-rustc/rustc_middle/ty/context/struct.TyCtxt.html#method.optimized_mir

[20] https://doc.rust-lang.org/stable/nightly-rustc/rustc_hir/def_id/struct.DefId.html

potentially be mitigated through the use of some sort of cache, it is out of the scope of this thesis. This optimization will be discussed in Sec. 7.3.

The more severe implication of using the call stack approach is the inability to handle recursive functions. When encountering a recursive function within the translation process, the process becomes trapped in an endless loop where the stack grows indefinitely as new stack frames are pushed to it, leading to a stack overflow and a subsequent crash of the translation process. This problem is addressed in Sec. 7.4 too. For now, it is necessary to accept the limitation that recursive functions cannot be translated using this framework.

4.2.3 Foreign functions and functions in the standard library

In Rust, the compiler includes by default the standard library in all compiled binaries, effectively linking it statically. To override this behavior, the crate-level attribute `#![no_std]` is used to indicate that the crate will link to the core-crate instead of the std-crate. See [Rust on Embedded Devices Working Group, 2023] for more details.

This means that the standard library's functionality becomes an integral part of the resulting executable. Function calls to the standard library appear in various contexts in Rust code, such as when accessing command line arguments, invoking iterators, utilizing traits like `std::clone::Clone`, `std::deref::Deref::deref`, or employing standard library types like `std::result::Result` or `std::option::Option`. Given the prevalence of these function calls throughout Rust programs, it becomes essential to handle them separately in the translation process. It is evident that these standard library functions, due to their purpose, cannot lead to a deadlock. Therefore, it is more practical to treat them as black boxes within the translation process, bypassing the need to translate their MIR. This approach is indispensable in order to avoid generating an excessively large and convoluted Petri net that would hinder readability and comprehension.

The focus of the translation effort lies primarily on the user code, specifically the functions that developers write to implement their desired functionalities. By directing attention to the user code and excluding the translation of standard library functions, the resulting Petri net remains more manageable, facilitating the analysis and verification of potential deadlocks within the user's codebase. The calls to the standard library constitute, in other words, the "frontier" or "boundary" of the translation, the point at which we stop translating the MIR accurately and rely instead on a simplified model.

Petri net model for a function with cleanup block

The model presented in Fig. 3.2 is the first approximation. There is, however, an implementation detail that requires careful attention. Numerous functions in the standard library contain not only an end place ("target block", in the *rustc* parlance) but also a cleanup place ("cleanup block"). This second execution path is taken when the function explicitly panics or more generally fails to achieve its goal for whatever reason. In this case, the control flow continues to a

different basic block, where variables are freed and eventually the program ends with a panic error code. Stated differently, the unwind of the stack begins as soon as a function encounters a non-recoverable failure.

Considering that the translator cannot tell if this abnormal situation could lead to a deadlock later in the translation process, it is imperative to translate this alternative execution path whenever possible. Only in counted exceptions, all related to the synchronization primitives and discussed in the respective sections, this cleanup block is ignored explicitly. The complete model for an abridged function call with a cleanup block can be seen in Fig. 4.2.

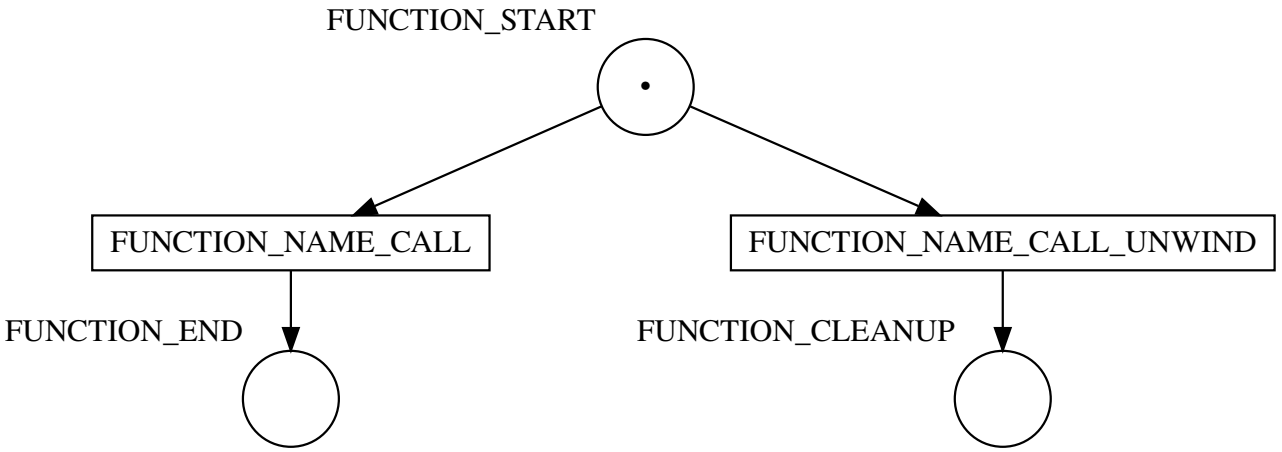

Figure 4.2: The Petri net model for a function with a cleanup block.

Functions translated with the abridged Petri net model

Having discussed the exclusion of standard library functions from the translation process, we now shift our focus towards the functions that indeed require translation using the model we presented earlier. Surprisingly, they include a considerable number of functions.

- Functions part of the standard library (the std-crate[21]), save for the `std::sync::Condvar::wait` function detailed in Sec. 4.8.3.
- Functions part of the core library (the core-crate[22]).
- Functions in the alloc-crate: the core allocation and collections library[23].
- Functions without a MIR representation This can be checked with the `is_mir_available` method[24].

[21] https://doc.rust-lang.org/std/
[22] https://doc.rust-lang.org/core/
[23] https://doc.rust-lang.org/alloc/
[24] https://doc.rust-lang.org/stable/nightly-rustc/rustc_middle/ty/context/struct.TyCtxt.html#method.is_mir_available

- Functions which are a foreign item i.e., linked via `extern { ... }`. This can be checked with the `is_foreign_item` method[25].

In the future, calls to functions in dependencies, i.e., in other crates, should also be handled in this fashion. In conclusion, the default case for functions that are *not* user-defined is to treat them as a foreign function and use an abridged Petri net model to translate them.

4.2.4 Diverging functions

Diverging functions are a special case that is relatively easy to support. It is simply a function that never returns to the caller. Examples of this are a wrapper around an infinite `while` loop, a function that exits the process, or a function that starts an OS. It suffices to connect the start place of the function to a sink transition (Definition 7) as seen in Fig. 4.3.

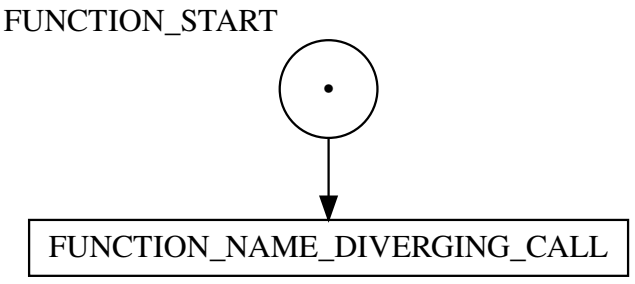

Figure 4.3: The Petri net model for a diverging function (a function that does not return).

Note that this special case does not constitute a deadlock and must *not* be treated as such. An infinite loop, i.e., a "busy wait", is in its inherent nature distinct from the infinite wait that characterizes a deadlock as seen in Sec. 1.4.1. In other words, detecting infinite loops is closer to the problem of detecting livelocks, which are out of the scope of this thesis. Besides, the translator cannot tell ahead of time if the diverging call is benign like a call to `std::process::exit` or a call to some kind of function carefully designed to block the program.

In the current PN model, the token is consumed and the net is left in an end state without tokens in the places `PROGRAM_END` or `PROGRAM_PANIC` shown in Fig. 4.1. Consequently, the model checker is able to distinguish this end state from the other cases and conclude that a diverging function has been called.

4.2.5 Explicit calls to panic

The `panic!()` macro can be seen as a special case of a divergent function where the transition representing the function call is connected to the place labeled `PROGRAM_PANIC` described in Sec. 4.1.1. The translator detects an explicit call to panic, which is one of the following functions:

[25] https://doc.rust-lang.org/stable/nightly-rustc/rustc_middle/ty/context/struct.TyCtxt.html#method.is_foreign_item

- `core::panicking::assert_failed`
- `core::panicking::panic`
- `core::panicking::panic_fmt`
- `std::rt::begin_panic`
- `std::rt::begin_panic_fmt`

The documentation[26] elaborates on why panic is defined in the core-crate and the std-crate and how it is implemented.

See Listing 4.2 for a simple program that panics. The corresponding Petri net model is depicted in Fig. 4.4. This is one of the illustrative examples included in the repository.

```
1 fn main() {
2     panic!();
3 }
```

Listing 4.2: A simple Rust program that calls `panic!()`.

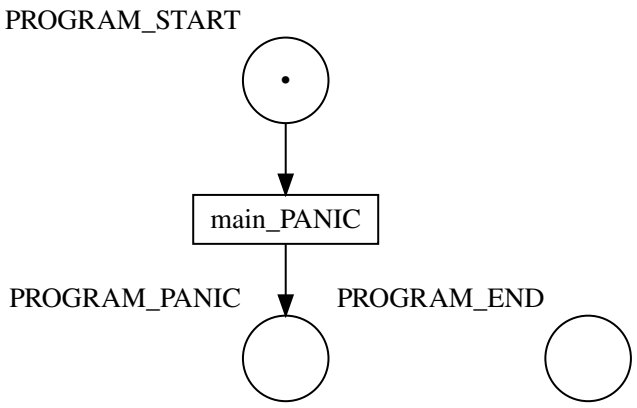

Figure 4.4: The Petri net model for Listing 4.2.

4.3 MIR visitor

This section is dedicated to a pivotal component that serves as the backbone of the translator: the MIR Visitor trait[27]. This trait enables straightforward navigation of the MIR of the Rust source code. In other words, it acts as the glue that seamlessly binds the various components of the translator together.

[26] https://rustc-dev-guide.rust-lang.org/panic-implementation.html
[27] https://doc.rust-lang.org/stable/nightly-rustc/rustc_middle/mir/visit/trait.Visitor.html

The MIR Visitor trait plays a fundamental role in the translation process by providing a structured approach to traverse and analyze the MIR. It offers a set of methods that can be implemented to perform specific actions at different points during the traversal. By employing this trait, the translator gains the ability to systematically explore the MIR and extract the necessary information for generating the corresponding Petri net.

The implemented methods within the MIR Visitor trait serve as entry points for handling different elements encountered during the traversal. These methods allow for customized processing of specific MIR constructs, e.g., basic blocks, statements, terminators, assignments, constants, etc. By defining appropriate behavior for each method, the translator can efficiently extract relevant data and make informed decisions based on the encountered MIR elements.

It was not required to implement all possible methods. If not defined, the methods in MIR Visitor simply call the corresponding `super` method and continue the traversal. For instance, `visit_statement` calls `super_statement` if no custom implementation is present. In the case of the translator, the implemented methods are:

- `visit_basic_block_data` for keeping track of the basic block currently being translated.
- `visit_assign` for keeping track of assignments of synchronization variables (mutexes, mutex guards, join handles, and condition variables).
- `visit_terminator` for processing the terminator statement of each basic block, that is, connecting the basic blocks.

To start visiting the MIR, the method `visit_body` must be used. Listing 4.3 shows the corresponding function in the translator.

In conclusion, the MIR Visitor trait simplifies remarkably the translation as it is not necessary to implement a traversal mechanism thanks to the provided compiler interfaces. This also makes the translator more robust and resistant to changes in *rustc*. If the string representation of the MIR changes, the translator is left unaffected. As long as the internal interfaces for accessing the MIR stay the same, the translator can navigate the MIR semantically and not based on how it is printed to the user.

As a last remark, similar traits exist for other intermediate representations:

- AST: https://doc.rust-lang.org/stable/nightly-rustc/rustc_ast/visit/trait.Visitor.html
- HIR: https://doc.rust-lang.org/stable/nightly-rustc/rustc_hir/intravisit/trait.Visitor.html
- THIR: https://doc.rust-lang.org/stable/nightly-rustc/rustc_middle/thir/visit/trait.Visitor.html

Various components in the compiler implement these traits to navigate the intermediate representations. To put it roughly, they are analogous to iterators for collections.

```rust
/// Main translation loop.
/// Translates the function from the top of the call stack.
/// Inside the MIR Visitor, when a call to another function happens, this method will be
↪ called again
/// to jump to the new function. Eventually a "leaf function" will be reached, the functions
↪ will exit and the
/// elements from the stack will be popped in order.
fn translate_top_call_stack(&mut self) {
    let function = self.call_stack.peek();
    // Obtain the MIR representation of the function.
    let body = self.tcx.optimized_mir(function.def_id);
    // Visit the MIR body of the function using the methods of
    ↪ `rustc_middle::mir::visit::Visitor`.
    //
    ↪ <https://doc.rust-lang.org/stable/nightly-rustc/rustc_middle/mir/visit/trait.Visitor.html>
    self.visit_body(body);
    // Finished processing this function.
    self.call_stack.pop();
}
```

Listing 4.3: The method in the `Translator` that starts the traversal of the MIR.

4.4 MIR function

In the following section, we will delve into the translation process of a MIR function. This section aims to provide a comprehensive understanding of the translation techniques applied to specific MIR elements, namely basic blocks (BB), statements, and terminators. These components were introduced previously in Sec. 3.4.1.

The implementation in the repository is accordingly named `MirFunction`[28]. This type stores the start place and the end place of the function. These must be supplied to the MIR function because they also represent where the function call took place and where it should return to. The end place is in simpler terms the return place in the Petri net. See Fig. 3.2 for an illustration.

The start place of the function overlaps with the place that models the first basic block in the function. This matches more closely the MIR as the code only lives inside of basic blocks, so the function call begins at the first basic block (BB0).

The `MirFunction` also stores the ID that identifies it. This is necessary for performing function calls from this function. Moreover, the function requires a name that is different for every

[28] https://github.com/hlisdero/cargo-check-deadlock/blob/main/src/translator/mir_function.rs

function call, so it receives a name with an index appended, making it unique across the whole Petri net.

We will now explain how each component is expressed in the language of Petri nets. Through a detailed exploration of the translation techniques employed for basic blocks, statements, and terminators, we will develop a formal model that accurately captures the behavior of a MIR function for deadlock detection.

4.4.1 Basic blocks

One aspect of the translation process involves transforming basic blocks into Petri nets, which serve as a fundamental building block for modeling the control flow within the MIR function. As seen in Fig. 3.1, a basic block in MIR acts as a container that houses a sequence of zero or more statements, as well as a mandatory terminator statement.

As nodes in a graph, the main property of basic blocks is their ability to direct the flow of control within a program. Each basic block may have one or more basic blocks pointing to it, indicating the potential paths from which the control flow can reach it. Similarly, a basic block can point to one or more other basic blocks, signifying the possible paths the control flow may take after executing the current basic block. It is worth mentioning that isolated basic blocks with no connections do not make sense since they would never be executed, i.e., they are dead code.

This branching behavior allows for dynamic control flow within the program, as multiple basic blocks can continue the control flow to the same target basic block (for instance to a block that performs cleanup tasks). Conversely, a basic block can branch and determine the next basic block based on specific conditions or program logic, e.g., in an `if`, `while`, `match` or other control structures. This versatility in control flow provides the foundation for modeling complex program behavior.

The Petri net model used in the implementation relies on a single place to model each BB. We can abstract away the inner workings of the BB and work with a single place. The rationale behind it is that the connections to other BB depend solely on the terminators and statements are not modeled at all as we will see shortly. Additionally, the implementation[29] keeps track of the function name to which the BB belongs and the BB number to generate unique labels.

4.4.2 Statements

MIR statements are intentionally *not* incorporated into the Petri net model. Considering the reasons for this and the benefits may not be immediately apparent, we will provide a detailed explanation for this implementation decision.

[29] https://github.com/hlisdero/cargo-check-deadlock/blob/main/src/translator/mir_function/basic_block.rs

The approach that was previously implemented did include the modeling of statements. It was based on the approach seen in [Meyer, 2020]. However, it was observed that this led to the creation of a long chain of places and transitions that did not significantly contribute to the detection of deadlocks or missed signals. Furthermore, it unnecessarily inflated the size of the Petri net representation, making it more difficult to debug and understand. Consequently, this approach was later revised and removed in a later commit[30].

In all the programs we had tested so far, the statements did not perform any action that may justify their addition to the Petri net. On the contrary, the nets that include the statements were larger and more difficult to read. In order to facilitate the use and adoption of the tool, it is crucial to optimize the Petri net for the purpose of the tool. The decision was therefore to eliminate all the code related to the modeling of the statements and fix the tests accordingly to match the new output.

The alternative was to disable the statements with a compile flag but that would complicate testing and since there is no use case for modeling the MIR statements anyway, this option was discarded.

For illustrative purposes, we can refer to Fig. 4.5, which showcases a comparison between the old model and the new model. The differences between these two representations are evident, highlighting the removal of statements from the model and the subsequent simplification achieved in the resulting Petri net.

4.4.3 Terminators

As seen in Sec. 3.4.1, terminator statements come in different shapes. The documentation for the enum `TerminatorKind`[31] lists as of this writing 14 different variants. The implementation is required to support most of them, since they appear sooner or later in the test programs included in the repository and their translation directly influences the connections between the basic blocks. The remaining terminators that are not implemented are not present when querying the `optimized_mir`, i.e., they are used only in previous compiler passes.

The implementation of the MIR Visitor[32] includes the `visit_terminator` method as seen before. This is where the edges connecting one BB to another are created. In the next paragraphs, the high-level details of each handler are discussed. Some implementation details are omitted as they do not affect the Petri net.

Goto

This is an elementary terminator kind. The end place of the currently active BB is connected to the start place of the target BB through a new transition with an appropriate label.

[30] https://github.com/hlisdero/cargo-check-deadlock/commit/b27403b6a5b2bb020a5d7ab2a9b1cacefb48be82

[31] https://doc.rust-lang.org/stable/nightly-rustc/rustc_middle/mir/enum.TerminatorKind.html

[32] https://github.com/hlisdero/cargo-check-deadlock/blob/main/src/translator/mir_visitor.rs

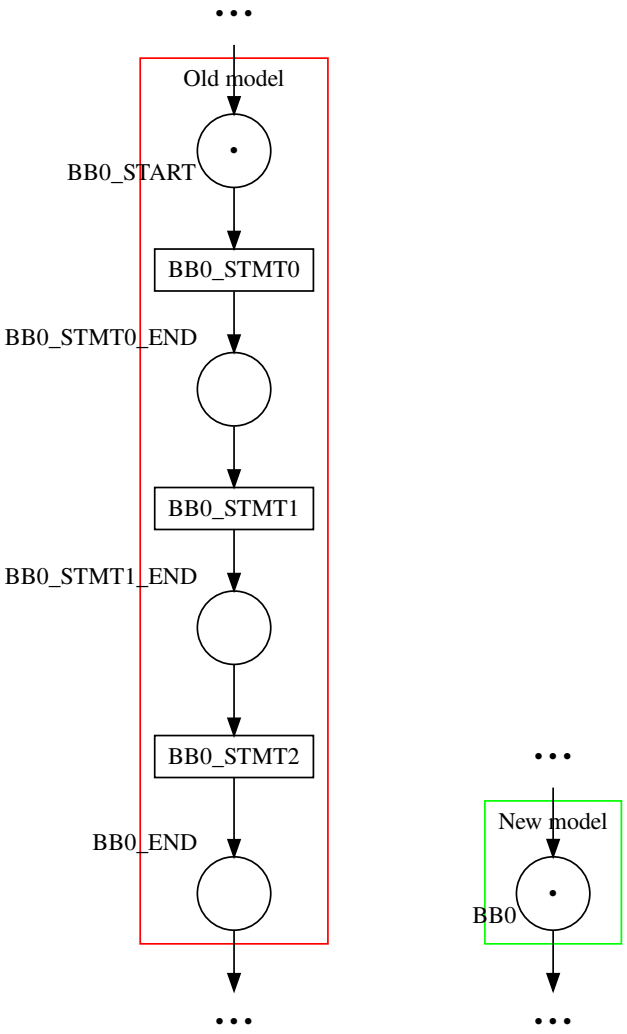

Figure 4.5: A side-by-side comparison of two possibilities to model the MIR statements.

SwitchInt

This terminator kind comes with a collection of target basic blocks. For each target BB, we connect the end place of the currently active BB to the start place of the target BB through a new transition with an appropriate label. This creates a *conflict* as defined in Sec. 1.1.3.

The label must also contain some kind of unique identifier of the block from where the jump starts. This is a precondition to correctly translate multiple basic blocks with a `SwitchInt` that jumps to the same block.

Resume or Terminate

These are terminators that model respectively an unwinding of the stack and the immediate abort of the program. Both are treated in the same way: Connecting the end place of the currently active BB to the `PROGRAM_PANIC` place seen in Fig. 4.1.

Return

This is the terminator that causes the MIR function to return. This is where the end place of the function is used. The end place of the currently active BB is connected to it.

Unreachable

This is a border case that appears in some `match`, `while` loops, or other control structures. The documentation states: *Indicates a terminator that can never be reached.* To handle this case, the decision was to connect the end place of the currently active BB to the `PROGRAM_END` place seen in Fig. 4.1. See the comments in the repository for more details.

Drop

The `std::ops::Drop` trait is used to specify code that should be executed when the type goes out of scope [Klabnik and Nichols, 2023, Chap. 15.3]. It is equivalent to the concept of destructors found in other programming languages.

The drop terminator behaves like a function call with a cleanup transition. Therefore, we apply the model shown in Fig. 4.2 with modified transition labels.

An important check happens here too, namely checking if a mutex guard is being dropped. If that is the case, then the corresponding mutex should be unlocked as part of the transition firing. Precise details are explained in Sec. 4.7.3.

Call

This is the terminator kind for executing function calls. The presence of a cleanup block and the particular `UnwindAction`[33] as well as the function name and type is analyzed to handle it according to the strategy elaborated in Sec. 4.2.

Note that `UnwindAction` is a refactor of *rustc* that was introduced on April 7th, 2023. It is a good example of a regression that required significant changes to accommodate. The interested reader is referred to the corresponding commit[34].

[33] https://doc.rust-lang.org/stable/nightly-rustc/rustc_middle/mir/syntax/enum.UnwindAction.html

[34] https://github.com/hlisdero/cargo-check-deadlock/commit/8cf95cd54b29c210801cae2941abcbbb85051b92

Assert

This terminator kind is related to the `assert!()`[35] macro and the default overflow checks that *rustc* incorporates when performing arithmetic operations.

The implementation does not model the condition for the assert. It simply connects the end place of the currently active BB to the start place of the target BB through a new transition with an appropriate label.

In some cases, a cleanup block is present too. For this, a second transition is needed, analogously to the `Drop` case.

4.5 Function memory

We will now proceed to explore the memory characteristics of the MIR function in detail. It is important to acknowledge that the need to record values being assigned between memory locations in the MIR arises from the requirements of the deadlock and missed signals detection. In simpler teams, we are forced to model the memory only because the supported synchronization variables need to be tracked during the translation process.

The translator must track variables of the following types:

- Mutexes (`std::sync::Mutex`).
- Mutex guards (`std::sync::MutexGuard`).
- Join handles (`std::thread::JoinHandle`).
- Condition variables (`std::sync::Condvar`).
- Aggregates, i.e., wrappers such as `std::sync::Arc` or types that contain multiple values like tuples or a structured type (`struct`).

Before calling methods on these types of synchronization variables, immutable or mutable references to the original memory location are created. The translator must somehow know which specific synchronization variable is behind a given reference. Knowing the type of the memory location is *not* enough, the *value* must be readily available to the translator to operate on the Petri net model of the specific synchronization variable.

4.5.1 A guided example to introduce the challenges

To illustrate the situation described previously, consider the Rust program shown in Listing 4.4. It is again one of the example programs found in the repository. As it should be evident to the reader, this program deadlocks when executed. The reason is that the `std::sync::Mutex::lock`

[35] https://doc.rust-lang.org/std/macro.assert.html

method is being called twice on the same mutex. To detect this deadlock, the translator must be able to at the very least identify that the invocation of `lock` takes place on the same mutex.

```
fn main() {
  let data = std::sync::Mutex::new(0);
  let _d1 = data.lock();
  let _d2 = data.lock(); // cannot lock, since d1 is still active
}
```

Listing 4.4: A deadlock caused by calling `lock` twice on the same mutex.

Observe now an excerpt of the MIR of the same erroneous program in Listing 4.5. The comments have been removed for clarity. In BB0 the mutex is created through a call to `std::sync::Mutex::new`. The new mutex is the return value of the function. It is assigned to the local variable `_1`. Then the execution continues in BB1. Focus on the first statement of BB1: An immutable reference to the local variable `_1` is stored in `_3`. Next, the reference is moved to the function `std::sync::Mutex::lock`. This reference is consumed by lock, that is to say, the local variable `_3` is not used anywhere else in the MIR because, from that point on, the ownership of the reference is transferred to the `std::sync::Mutex::lock` function.

Immediately after the statement, the translator encounters the terminator of BB1. It contains a call to `std::sync::Mutex::lock`. How would the translator know, when translating this call, that `_3` is indeed the mutex stored in `_1`? This is the problem that the modeling of the function's memory aims to solve.

The problem goes even further. The local variable `_2` contains a mutex guard after the call to lock, which should be recorded too. Notice how BB2 repeats the same operations as BB1 but uses different local variables, `_5` and `_4`. The translator should know that `_5` is an alias for `_1` as well. Furthermore, the mutex guards in `_2` and `_4` will eventually be dropped, which indirectly unlocks the mutex. There has to be a link from the mutex guard in `_2` and `_4` to the mutex in `_1`. More concisely, the translator should monitor which mutex is behind each mutex guard.

To make matters more complex, each MIR function has its own stack memory, with its separate local variables `_0`, `_1`, `_2`, `_3`, and so on. Thus, the mapping of memory locations to synchronization variables cannot be a single global structure. It is instead dependent on the context of the current function being translated. Lastly, a synchronization variable could migrate from one function to another and the translator must be able to re-map them correctly.

This suffices as a brief practical example of the challenges of memory modeling. We can now introduce the solution that has been implemented.

```
fn main() -> () {
  let mut _0: ();
  let _1: std::sync::Mutex<i32>;
  let mut _3: &std::sync::Mutex<i32>;
  let mut _5: &std::sync::Mutex<i32>;
  scope 1 {
      debug data => _1;
      let _2: std::result::Result<std::sync::MutexGuard<'_, i32>,
      ↪ std::sync::PoisonError<std::sync::MutexGuard<'_, i32>>>;
      scope 2 {
          debug _d1 => _2;
          let _4: std::result::Result<std::sync::MutexGuard<'_, i32>,
          ↪ std::sync::PoisonError<std::sync::MutexGuard<'_, i32>>>;
          scope 3 {
              debug _d2 => _4;
          }
      }
  }

  bb0: {
      _1 = Mutex::<i32>::new(const 0_i32) -> bb1;
  }

  bb1: {
      _3 = &_1;
      _2 = Mutex::<i32>::lock(move _3) -> bb2;
  }

  bb2: {
      _5 = &_1;
      _4 = Mutex::<i32>::lock(move _5) -> [return: bb3, unwind: bb6];
  }
```

Listing 4.5: An except of the MIR of the program from Listing 4.4.

4.5.2 A mapping of `rustc_middle::mir::Place` to shared counted references

The implementation is suitably named `Memory`[36]. As anticipated in the previous section, there is one instance of `Memory` per `MirFunction`. The memory is tightly connected to the context of

[36] https://github.com/hlisdero/cargo-check-deadlock/blob/main/src/translator/mir_function/memory.rs

the MIR function.

Rather than moving values between different memory locations, as observed in the MIR, our solution relies on the simpler concept of "linking". This entails associating a specific `rustc_middle::mir::Place` with the corresponding value. This association is not removed when moving the variable to a different function. It also does not differentiate a shallow copy of the value from taking a reference or a mutable reference. To put it shortly, it is an all-encompassing mapping between places and values.

To accommodate the possibility of linking the same value to multiple places, particularly when multiple memory locations hold an immutable reference to the value, it becomes necessary for the stored value to be a reference to the synchronization variable. To clarify, this introduces a second level of indirection. In order to facilitate the required cloning operations, we have opted to utilize `std::rc::Rc`, which is a smart pointer provided by the Rust standard library. The ownership of the referenced value (the synchronization variable) is shared and every time that the value is cloned, an internal counter is incremented. When the count reaches zero, the value is freed [Klabnik and Nichols, 2023, Chap. 15.4].

The `Memory` utilizes a `std::collections::HashMap` data structure that establishes a mapping between `rustc_middle::mir::Place` instances and an enum with 5 variants corresponding to the 5 types mentioned previously that the translator tracks. 4 of these 5 variants enclose a `std::rc::Rc` reference to the synchronization variable. The aggregate case instead contains a vector of `Value`. This enables nesting aggregate values inside of each other, which is a critical requirement for supporting more complex programs with nested `structs`.

Using a hash map allows for efficient retrieval and management of the associated values during the translation process. The `Memory` also takes care of providing typedefs for the different references to synchronization variables. Listing 4.6 depicts an excerpt of the source file with the essential type definitions used in the implementation. Improvements to the current implementation are discussed in Sec. 7.5.

4.5.3 Intercepting assigments

The missing piece in the puzzle of the memory model is where to link the memory locations exactly. There are three separate places in the code in which this takes place.

On the one hand, the translator functions responsible for processing the methods of mutexes, condition variables, and threads create new synchronization variables that are linked to the return value of the corresponding method. This is where the lifetime of each synchronization variable starts. The specifics are expanded upon in Sec. 4.7.3 and 4.8.3.

On the other hand, the synchronization variable may be assigned in any other BB. For this reason, the translator incorporates a custom implementation of the method `visit_assign` to intercept every assignment in the MIR. Listing 4.7 shows precisely that all cases of copying, moving, or referencing the right-hand side (RHS) are handled by the same mechanism: The left-

```rust
#[derive(Default)]
pub struct Memory<'tcx> {
    map: HashMap<Place<'tcx>, Value>,
}

/// ...

/// Possible values that can be stored in the `Memory`.
/// A place will be mapped to one of these.
#[derive(PartialEq, Clone)]
pub enum Value {
    Mutex(MutexRef),
    MutexGuard(MutexGuardRef),
    JoinHandle(ThreadRef),
    Condvar(CondvarRef),
    Aggregate(Vec<Value>),
}

/// ...

/// A mutex reference is just a shared pointer to the mutex.
pub type MutexRef = std::rc::Rc<Mutex>;

/// A mutex guard reference is just a shared pointer to the mutex guard.
pub type MutexGuardRef = std::rc::Rc<MutexGuard>;

/// A condvar reference is just a shared pointer to the condition variable.
pub type CondvarRef = std::rc::Rc<Condvar>;

/// A thread reference is just a shared pointer to the thread.
pub type ThreadRef = std::rc::Rc<Thread>;
```

Listing 4.6: A summary of the type definitions of the `Memory` implementation.

hand side (LHS) is linked to the right-hand side (RHS) if the type of the variable is a supported synchronization variable. The listing also shows how the compiler uses nested enums to model its data. Inside the variants of a right-hand side value (`rustc_middle::mir::Rvalue`), one can find operands (`rustc_middle::mir::Operand`). These operands also appear when passing function arguments.

The most peculiar case is the aggregate assignment. It materializes from assignments in Rust source code that create tuples, closures, or `structs`. It necessitates special handling as the value

```rust
fn visit_assign(
    &mut self,
    place: &rustc_middle::mir::Place<'tcx>,
    rvalue: &rustc_middle::mir::Rvalue<'tcx>,
    location: rustc_middle::mir::Location,
) {
    match rvalue {
        rustc_middle::mir::Rvalue::Use(
            rustc_middle::mir::Operand::Copy(rhs) | rustc_middle::mir::Operand::Move(rhs),
        )
        | rustc_middle::mir::Rvalue::Ref(_, _, rhs) => {
            let function = self.call_stack.peek_mut();
            link_if_sync_variable(place, rhs, &mut function.memory, function.def_id,
            ↪ self.tcx);
        }
        rustc_middle::mir::Rvalue::Aggregate(_, operands) => {
            let function = self.call_stack.peek_mut();
            handle_aggregate_assignment(
                place,
                &operands.raw,
                &mut function.memory,
                function.def_id,
                self.tcx,
            );
        }
        // No need to do anything for the other cases for now.
        _ => {}
    }

    self.super_assign(place, rvalue, location);
}
```

Listing 4.7: The custom implementation of `visit_assign` to track synchronization variables.

to be linked in memory must be assembled from the constituents of the aggregated value that are a synchronization variable. This implies that the `Memory` solely retains the portion of the aggregated value formed by the synchronization variables.

Tracking the assignments of synchronization variables at the moment they are returned from functions is another crucial mechanism. Fortunately, this can be accomplished by implementing a consistent check on all functions, regardless of whether they are modeled using the simple model (Fig. 3.2) or the function with cleanup model (Fig. 4.2). As a benefit, this uniform

design readily supports `std::arc::Arc` without requiring any additional effort.

In every instance, the handling of assignments has no impact on the Petri net. No places or transitions are added when intercepting assignments.

Finally, some memory locations are passed to a new thread when calling `std::thread::spawn` and mapped again to the memory of the thread's function. The next section will demonstrate the method used to accomplish this.

4.6 Multithreading

Multithreading support is a prerequisite for deadlock and missed signal detection. In order to support real-world programs where deadlocks or missed signals are possible in the first place, it becomes essential to support having several threads that share resources. First, the basics will be presented to later devise a PN model that captures the behavior of threads in Rust code.

4.6.1 Thread lifetime in Rust

The lifetime of a thread begins when it is started by invoking the `std::thread::spawn`[37] function. It receives a closure or function as an argument, representing the code that the new thread will execute concurrently with the other threads of the program. The spawned thread may start running immediately after it is spawned but there is no guarantee that it will do so.

Contrary to other programming languages like C, C++, or Java, Rust does not have the notion of a thread variable initialized previously to the start of the thread. Instead, the function `std::thread::spawn` returns a `std::thread::JoinHandle`, which is, as the name suggests, a handle to call `join` at the end of the thread's lifetime.

During its existence, a thread can independently execute its designated code and perform various operations concurrently with other threads. It can access shared resources and communicate with other threads through synchronization mechanisms like mutexes, condition variables, channels, or atomic operations. This enables concurrent processing and parallelism in Rust programs.

To ensure proper coordination between threads, Rust provides a mechanism to join threads. The `std::thread::JoinHandle::join`[38] method allows the main thread or another thread to wait for the completion of a different thread. By calling join on a join handle, the calling thread blocks until the spawned thread finishes its execution. Once a thread completes its execution and is joined by another thread, its lifetime ends, and the corresponding system resources are released. Otherwise, threads that were not joined correctly may potentially leak resources.

[37] https://doc.rust-lang.org/std/thread/fn.spawn.html
[38] https://doc.rust-lang.org/std/thread/struct.JoinHandle.html#method.join

If the join handle is dropped, the thread may no longer be joined and it implicitly becomes *detached*. A detached thread refers to a thread without a valid join handle. It will continue its execution independently until it completes or the program terminates. They are useful in scenarios where the spawning thread does not need to wait for the thread to complete its task. For example, in long-running background tasks or when the main thread terminates independently of the detached thread's progress. However, it's important to stress that the execution of detached threads may continue *even* after the main thread exited.

4.6.2 Petri net model for a thread

To incorporate additional threads into the PN model, a distinct subnet is appended to the main net to represent each thread. This subnet encapsulates the execution path of the newly spawned thread and operates as an isolated context. It establishes precise interfaces that connect back to the main net. The closure provided to the `spawn` function, being a MIR function, can invoke other functions that in turn require translation. Therefore, processing a thread's function follows a similar approach to the usual translation logic.

The concurrency aspect of the execution of the new thread is modeled by the generation of a new token at the transition that represents the call to `spawn`. This token can be interpreted, in the same manner as the token in `PROGRAM_START`, as the instruction counter of the new thread. Essentially, the spawn operation constitutes a "fork" in the token flow: One token enters the transition and two tokens exit from it. The first proceeds along the main thread's path to execute the subsequent statement, while the second is directed to the first BB of the function passed to the thread.

Each thread identified in the source code possesses designated start and end places labeled `THREAD_<index>_START` and `THREAD_<index>_END`, respectively. The index is mandatory to preserve the label uniqueness property across the entire program. It should be emphasized that this mimics the basic places for the program detailed in Sec. 4.1.1.

Threads lack a separate panic place as invoking `panic!()` inside a thread only terminates that specific thread's execution. We are not interested in differentiating between thread end states; the main requirement is to determine whether a thread has finished or not. A single end place for both cases suffices here.

The joining behavior serves as the inverse operation of the spawn. The transition corresponding to the `join` call consumes two tokens but generates only one token. As a result, the waiting condition is modeled straightforwardly: The main thread can continue, i.e, the `join` transition can fire, if and only if the thread to be joined has finished execution, reaching its respective `THREAD_END` place.

To recapitulate, the thread is translated to a separate subnet that interfaces with the main net only at three places:

- The `spawn` transition where the thread starts.

- The (optional) `join` transition where the join handle is utilized.
- The connections due to synchronization variables, analyzed later in the dedicated sections of this chapter.

4.6.3 A practical example

Observe Listing 4.8 and its corresponding PN model in Fig. 4.6. This is one of the test programs found in the repository. Note the "fork" at the spawn transition described in the previous subsection. The left branch is the thread, while the right branch is the main thread. It is clear that the paths split at the `spawn` and merge at the `join`. Notice also that there is no separate panic place for the thread, indicating that a failure in one thread does not impact the other threads.

```
1 fn main() {
2   let thread_join_handle = std::thread::spawn(move || {
3       // some work here
4   });
5   // some work here
6   let _res = thread_join_handle.join();
7 }
```

Listing 4.8: A basic program with two threads to demonstrate multithreading support.

4.6.4 Algorithms for thread translation

To close this section, we will briefly describe the algorithms used for translating threads. Initially, it is worth mentioning that since the translation is carried out by a single thread (the tool does not support multiple threads translating the source code), a decision has to be made concerning when to translate spawned threads:

- Immediate translation: Translate the thread as soon as it is encountered. The translator "switches" to the spawned thread.
- Delayed translation: Store all the relevant information about the new thread and translate it after the main thread.

The current solution takes the latter approach.

When a call to `std::thread::spawn` is encountered:

1. Translate the function call using the model seen in Fig. 4.2.
2. Retrieve the first argument passed to the function: An aggregate value that holds the variables captured by the closure and the function to be executed by the thread.

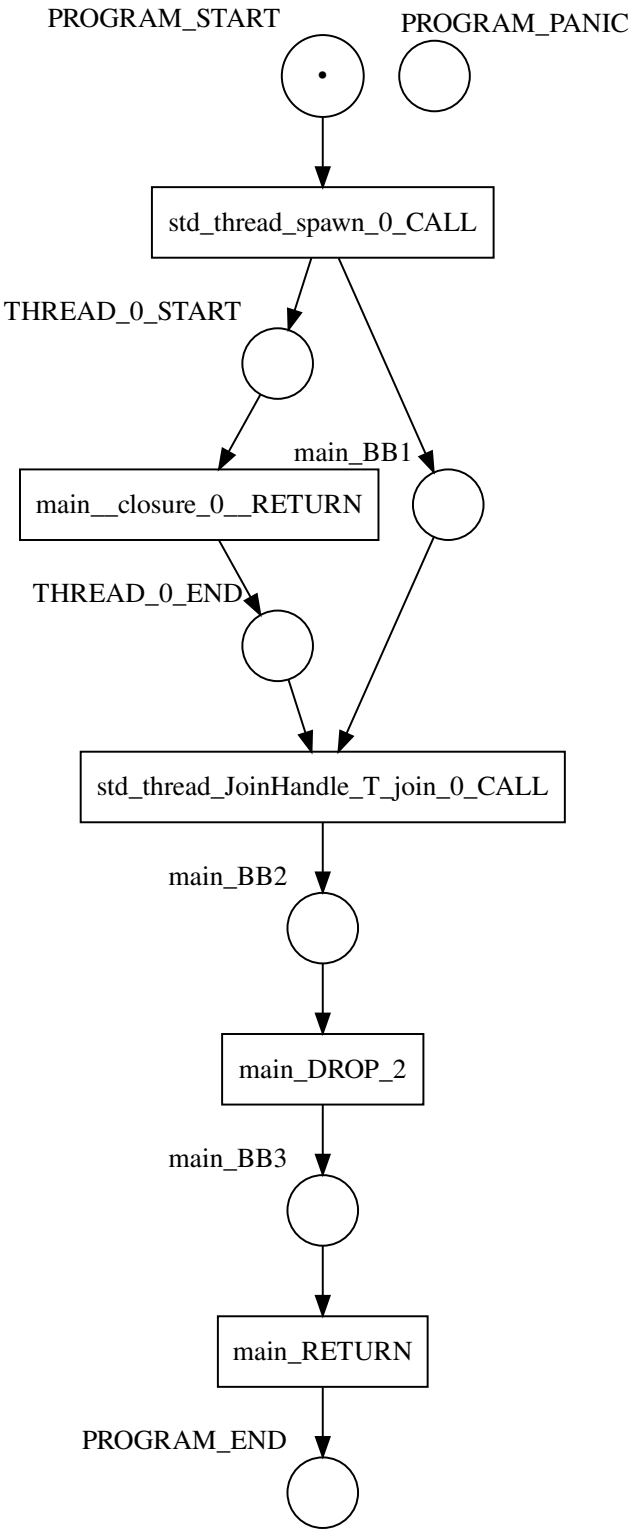

Figure 4.6: The Petri net model for the program in Listing 4.8.

3. Extract the ID of the function to be executed by the thread.
4. Extract the values captured by the closure.
5. Create a new `Thread`[39] to store the information required for the delayed translation.
6. Link the return value of `std::thread::spawn`, the new join handle, to the `Thread`.
7. Push the thread to a queue of detected threads in the `Translator`.

When a call to `std::thread::JoinHandle::join` is encountered:

1. Translate the function call using the model seen in Fig. 3.2. Ignore the cleanup place since we must force the PN to "wait" for the thread to exit. This is equivalent to assuming that the `join` function never fails.
2. Retrieve the first argument passed to the function: The join handle. The memory location is linked to the corresponding thread thanks to the assignment interception explained in Sec. 4.5.3.
3. Set the join transition of the underlying `Thread` behind the join handle.

When the main thread finishes translating, that is, when the `main` function has already been processed, the `Translator` enters a loop to translate the threads discovered so far in order.

1. Create a new start and end place for the thread.
2. Connect the spawn transition to the start place.
3. If a join transition was found, connect the end place to it.
4. Replace the place `PROGRAM_PANIC` with the place `THREAD_<index>_END` to translate terminators like `Unwind` correctly (Sec. 4.4.3).
5. Push the thread function to the call stack.
6. Move the synchronization variables to the memory of the thread function, i.e., map the aggregate value and its fields to the memory of the thread function.
7. Translate the top of the call stack.

As anticipated before, the algorithm exhibits resemblances to the overall procedure for function calls outlined in Sec. 4.2. Lastly, the implementation is capable of handling programs where threads spawn their own threads in a nested manner. The threads are simply added to the queue and as the loop advances, the nested threads are translated too.

[39] https://github.com/hlisdero/cargo-check-deadlock/blob/main/src/translator/sync/thread.rs

4.7 Mutex (std::sync::Mutex)

A mutex, short for mutual exclusion, is a synchronization mechanism used to control access to a shared resource in a concurrent program. It allows multiple threads to access the shared resource in a mutually exclusive manner, ensuring that only one thread can access the resource at a time.

In this section, the PN model for a mutex in Rust is explained, then a practical example is presented to ease comprehension and finally the algorithms used for the translation of mutex functions are outlined.

4.7.1 Petri net model

In Rust, a mutex is created by wrapping the shared data in a `Mutex<T>` type, where `T` is the type of the shared resource. The `std::sync::Mutex` type exposes a method named `lock` to acquire the lock on the shared resource. If the mutex is currently unlocked, the thread successfully acquires the lock and can proceed with accessing the resource. If the mutex is already locked by another thread, the thread attempting to acquire the lock will be blocked until the lock becomes available. The `lock` method returns a mutex guard (`std::sync::MutexGuard`) that grants exclusive access to the resource until it is dropped.

Contrary to the `unlock` semantics present in C or C++, the mutex included by the Rust standard library is unlocked implicitly, i.e., without calling a function. The mutex implements Resource Acquisition Is Initialization (RAII) and releases the lock automatically when it goes out of scope, preventing deadlocks. Alternatively, dropping a local variable of type `std::sync::MutexGuard` is equivalent to unlocking the corresponding mutex.

A mutex can be modeled in PN as a single place that represents the state of the mutex, indicating whether it is locked or unlocked. The place is labeled to reflect its purpose as a mutex. Besides, the place is marked with a token initially to signify that the mutex starts in the unlocked state.

Transitions that lock the mutex consume the token from the mutex place. If the token is absent, the transition may not fire. The mutex must be in the unlocked state to enable the locking transition, which is the desired behavior.

Transitions that unlock the mutex produce a token at the mutex place. The transition can fire as long as the program reached that point in the execution. After the transition fires, the mutex place holds again a token that can be consumed by a locking transition. Two types of transitions may unlock the mutex:

1. A `Drop` terminator (Sec. 4.4.3) when the dropped place is of type `std::sync::MutexGuard`.
2. The transition for a call to `std::mem::Drop`, which frees the memory occupied by the value passed in explicitly.

By connecting the mutex place to the locking and unlocking transitions using input and output arcs, we establish the relationship between the mutex state and the actions that manipulate it. This modeling approach allows for the representation of the mutex's behavior in a PN and facilitates the analysis of its interactions with other parts of the system.

The PN model presented here is well-known in the literature and has been applied successfully in other tools. It can be found among others in [Kavi et al., 2002, Moshtaghi, 2001, Meyer, 2020, Zhang and Liua, 2022].

4.7.2 A practical example

Consider the PN model shown in Fig. 4.7 corresponding to the program in Listing 4.4. The MIR is depicted in 4.5. This test program is one of the examples included in the repository.

Observe that there are two locking transitions mapping the two calls to `lock` in the source code. The indexing system mirrors the order of their appearance in the program, which justifies the labels `std_sync_Mutex_T_lock_0_CALL` and `std_sync_Mutex_T_lock_1_CALL`. Both have an incoming arc from the mutex place `MUTEX_0`.

As explained before, `Drop` terminators may unlock a mutex. No matter if they fail or not (the error case includes the suffix `_UNWIND`), an outgoing arc flows back to the mutex place to replenish the token.

One should take note that there are more incoming arcs to the mutex place than outgoing arcs, which highlights the importance of following the mutex guards throughout the MIR using the strategy explained in Sec. 4.5.3.

4.7.3 Algorithms for mutex translation

Concluding this section, we will provide a brief overview of the algorithms employed in the translation of mutex functions.

When a call to `std::sync::Mutex::new` is encountered:

1. Translate the function call using the model seen in Fig. 4.2.
2. Create a new `Mutex`[40] structure with an index to identify it unequivocally across the PN.
3. Link the return value of `std::sync::Mutex::new`, the new mutex, to the `Mutex` structure.

When a call to `std::sync::Mutex::lock` is encountered:

1. Translate the function call using the model seen in Fig. 3.2. Ignore the cleanup place since we must force the PN to "wait" for the mutex place to be marked. This is equivalent to assuming that the `lock` function never fails.
2. Retrieve the `self` reference to the mutex on which the function is called.

[40] https://github.com/hlisdero/cargo-check-deadlock/blob/main/src/translator/sync/mutex.rs

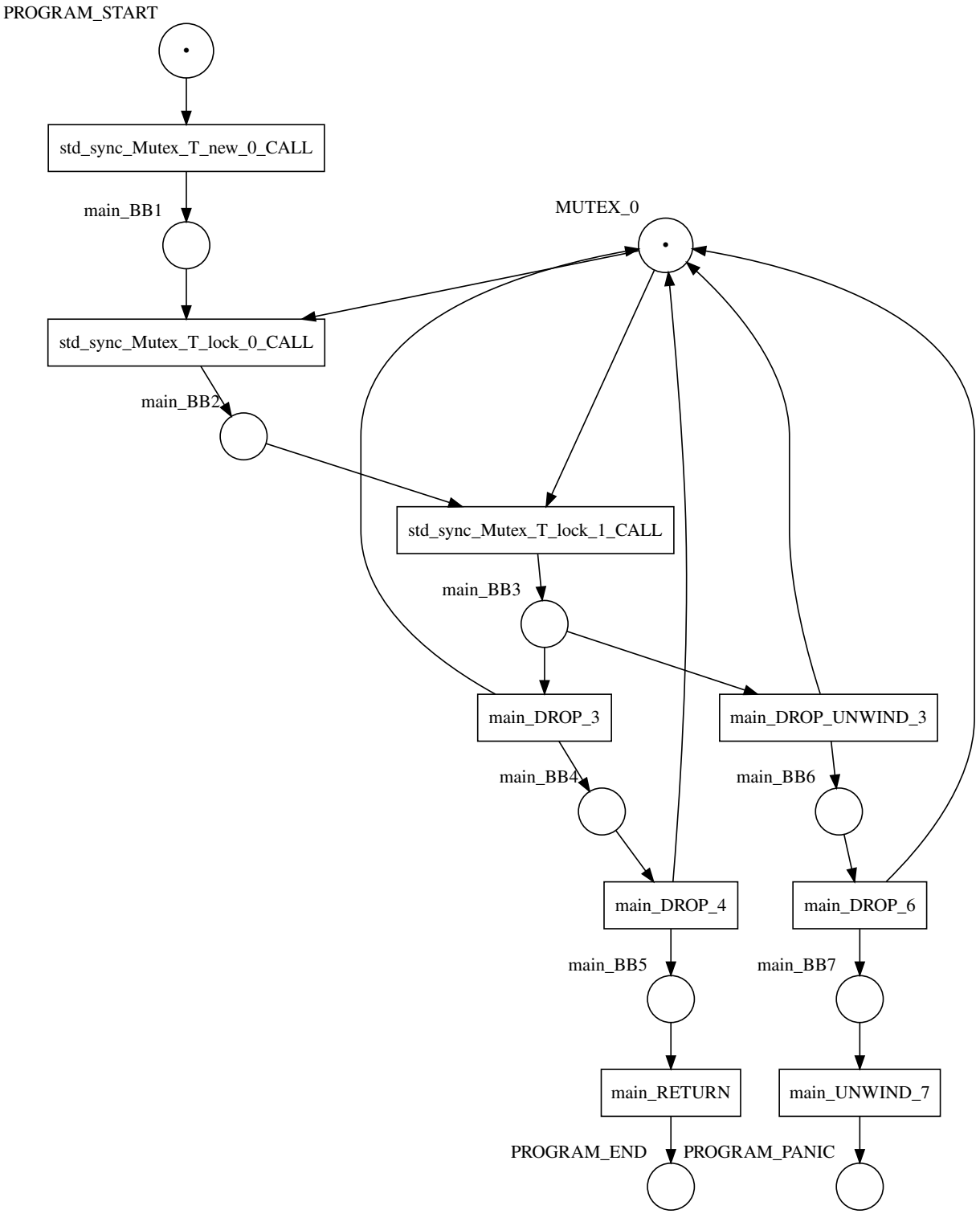

Figure 4.7: The Petri net model for the program in Listing 4.4.

3. Add an arc from the underlying mutex place to the transition representing the function call.
4. Create a new `MutexGuard` with a reference to the `Mutex`.
5. Link the return value of `std::sync::Mutex::lock`, the new mutex guard, to the `MutexGuard` structure.

When a call to `std::mem::drop` is encountered:

1. Translate the function call using the model seen in Fig. 4.2.
2. Extract the variable passed into the function.
3. If the variable is linked to a mutex guard, add an arc from the transition of the function call to the mutex place.
4. If a cleanup place was provided, add an unlock arc from the cleanup transition to the mutex place too.

When a terminator of kind `rustc_middle::mir::TerminatorKind::Drop` is encountered:

1. If the variable to be dropped is linked to a mutex guard, add an arc from the transition of the function call to the mutex place.
2. If a cleanup place was supplied, add an unlock arc from the drop unwind transition to the mutex place too.

In the upcoming section, we will delve into the necessary adjustments of these algorithms to establish a unified model for condition variables, which is essential for detecting missed signals. Given that these modifications are better comprehended within the framework of condition variables, we will elucidate them in that specific context.

4.8 Condition variable (`std::sync::Condvar`)

A condition variable is a synchronization primitive used in concurrent programming to enable threads to wait for a certain condition before proceeding with their execution. Threads wait until they are notified by another thread that the desired condition has been met.

Condition variables are typically associated with a mutex, which ensures exclusive access to the shared data that the condition depends on. When a thread waits on a condition variable, it releases the associated mutex, allowing other threads to make progress. When the condition becomes true or some event occurs, a notifying thread signals the condition variable, allowing one or more waiting threads to resume their execution.

The semantics of condition variables, as well as examples in pseudocode, were introduced in Sec. 1.5. The understanding of the precise behavior of condition variables in all circumstances is a prerequisite for this section.

This section provides an elaborate explanation of the PN model used to represent condition variables from the Rust standard library. It is followed by a practical example that aims to enhance the clarity of the concepts. Finally, the algorithms for the translation of condition variable functions are outlined.

4.8.1 Petri net model

In this particular case, the PN model must be examined carefully, as it involves not only the condition variable itself but also the variable that holds the condition on which the blocked thread is waiting *and* the mutex that synchronizes access to that condition.

This interaction can be extremely complex in general. For instance, the same condition variable could be used to signal an arbitrary number of distinct conditions. Accordingly, different mutexes may be passed as an argument to the `wait` call. Furthermore, an arbitrary number of threads may block on a condition variable and Rust supports the broadcast operation to awaken all waiting threads at once through the method `notify_all`[41] (see Sec. 1.5). Most importantly, the condition itself could be of any type and could take a long sequence of values during the execution, depending on which the waiting threads could act in diverse ways for every scenario.

For the above reasons, it is unavoidable to make assumptions regarding the supported use cases of condition variables in order to reduce the complexity of the task. Embracing and dealing with every possibility is beyond the scope of this thesis.

Assumptions

1. *Single call*: There is only one call to `wait` per condition variable, i.e., `condvar.wait()` appears in a single place in the source code for a given `condvar`. For example, it can be inside of a loop but it cannot be in two different functions.
2. *Single-element queue*: There is at most one waiting thread per condition variable.
3. *Boolean condition*: The condition is a boolean flag. It is either set or not set. Waiting on a condition that may take 3 or more values is *not* supported by this model.
4. *Mandatory set-condition / No "false notify"*: If a thread locks the mutex and accesses the shared condition mutably, then it always sets it to a different value. In simpler terms, threads that look at the value, do not change it and immediately notify the condition variable are *not* supported.
5. *Broadcast exclusion*: The method `std::sync::Condvar::notify_all` is out of scope.

Support for multiple calls to `wait` and multiple waiting threads could be implemented but a considerable implementation effort is required. Therefore, assumptions 1 and 2 can be overcome with the proposed model.

[41] https://doc.rust-lang.org/std/sync/struct.Condvar.html#method.notify_all

Supporting non-boolean conditions and detecting which value is set necessitates a thorough reconsideration of the modeling approach for representing concrete data values in simple Petri nets. Consequently, assumptions 3 and 4 are particularly challenging and could be the subject of future research into higher-lever models. See Sec. 7.6 for some thoughts to this effect.

Analysis of the proposed model

Fig. 4.8 depicts the PN model used in the implementation. The same diagram in DOT, PNG, and SVG format can be found in the repository as documentation.

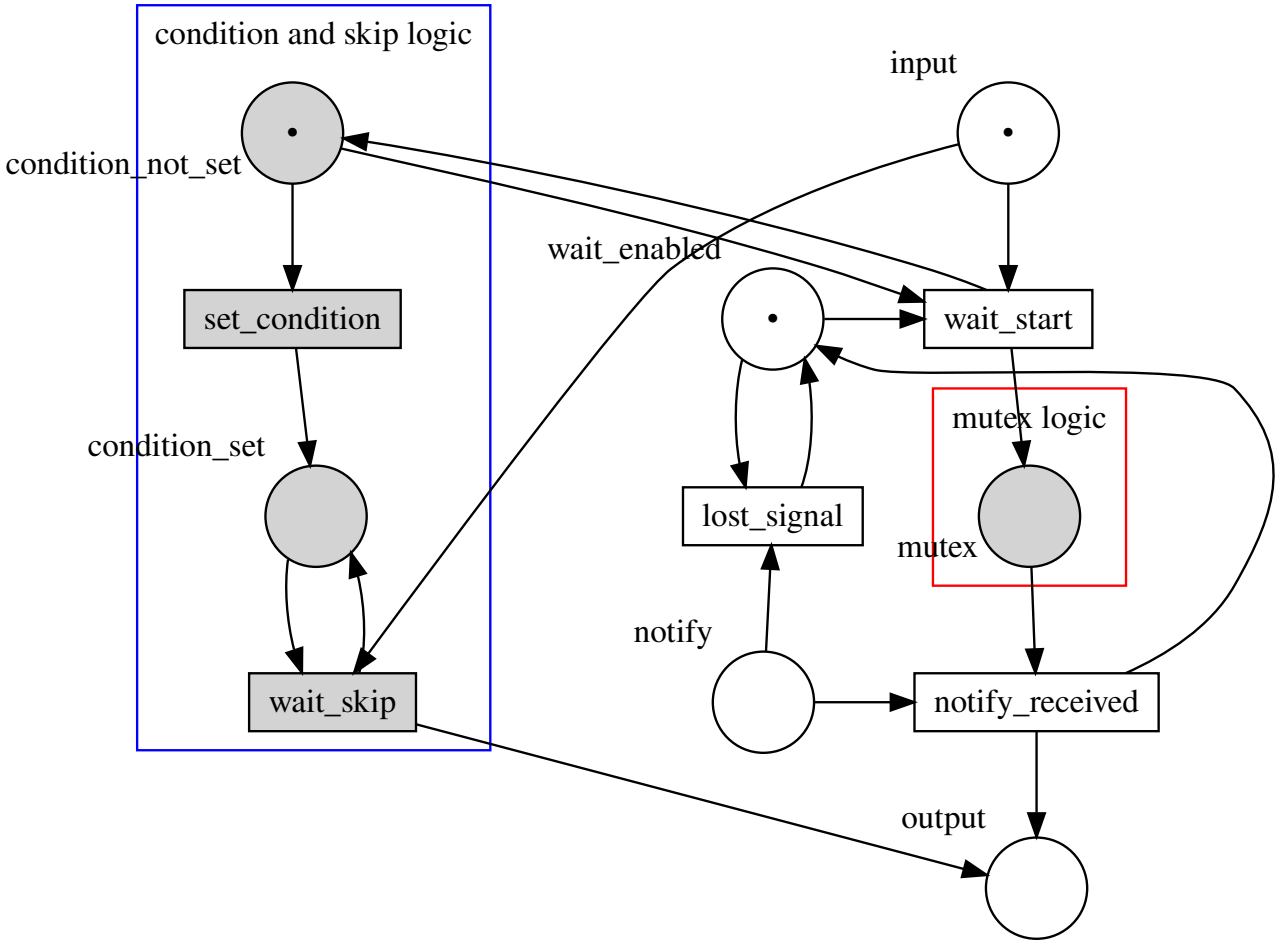

Figure 4.8: The Petri net model for condition variables.

The input places are:

- `input`: The start place of the wait function. The model supports the methods from the standard library `std::sync::Condvar::wait` and its variation `std::sync::Condvar::wait_while`.
- `condition_not_set`: The place is marked when the condition is `false`.

- `condition_set`: The place is marked when the condition is `true`.
- `notify`: The place where the notifying thread places a token to wake up the waiting thread.

The `output` place is the end place of the function call to `wait` or `wait_while`. The execution of the thread continues from there.

Two possible ways exist to go from `input` to `output`, represented by two transitions:

- `wait_start`: This is the "common case", the thread blocks and waits for the signal.
- `wait_skip`: This is the alternative path that the token takes when the condition was already set. The thread does not wait, instead, it skips the wait and reaches `output` in a single jump.

It is essential to notice that the greyed-out part on the left of Fig. 4.8 controls which transition is enabled and which transition is disabled. As soon as a token is set in `condition_set`, `wait_start` is disabled. Before that, the opposite is true: `wait_start` may fire but `wait_skip` may not.

Note the arcs between `condition_not_set` and `wait_start`. The token is regenerated every time that `wait_start` fires. The same is true for `condition_set` and `wait_skip`. These arcs restore the condition places to their previous state. Modeling more than 2 values as a PN would entail a more convoluted net, Besides, it would be impossible to know at compile time, how many possible values the condition takes to generate the correct number of places. This limitation is the justification for Assumptions 3 and 4.

Now focus on the right side of Fig. 4.8. In the middle of the condition variable, we find the place for the mutex. As expected, it is unlocked when the wait starts and it is locked when the notify is received.

The place labeled `wait_enabled` plays an important role. On the one hand, it consumes the token from `notify` if `wait_start` was not fired. This is the archetypical missed signal case that we would like to detect. On the other hand, the token in `wait_enabled` is consumed when `wait_start` fires, This prevents the condition variable from "accepting" other threads (Assumption 2) and preserves the token in `notify`, ensuring that the missed signal cannot occur.

Finally, the `notify_received` transition combines the requisites for the thread to leave the wait: The mutex must be unlocked and `notify_one` was called. To restore the initial state of the condition variable, it regenerates the token in `wait_enabled`.

Table of possible inputs and expected outputs

As a complement to the explanation in the previous subsection, here is a table summarizing the expected output for a given input. The reader may compare Fig. 4.8 to verify that the model produces the correct output for each scenario.

Row #	Input			Output
	`condition_set`	`wait_enabled`	`notify`	*where the initial token at* ***`input`*** *ends*
R1	False	False	False	`waiting` (waiting for a `notify`)
R2	False	False	True	`output` (correct wait end condition)
R3	False	True	False	`input` (initial state)
R4	False	True	True	*lost signal (transient state, goes to R1)*
R5	True	False	False	`waiting` (condition set, needs `notify`)
R6	True	False	True	`waiting` (correct wait end condition)
R7	True	True	False	`output` (skip the wait)
R8	True	True	True	`output` (skip the wait, with lost signal)

Table 4.1: A summary of the possible states of the Petri net model for condition variables.

4.8.2 A practical example

Due to the size constraints of the resulting PN, we are compelled to select a small-scale program for demonstration purposes. It would be unfeasible to embed within a single page the complete PN of a realistic program with condition variables and multiple threads. For more comprehensive examples, readers are encouraged to explore the repository, which contains a collection of more intricate programs included as part of the integration tests.

Despite the space limitations, this example in Listing 4.9 comprises the core elements of the model presented before. The complete PN can be seen in Fig. 4.9.

Observe the following sequence of transitions:

1. The mutex is locked in `std_sync_Mutex_T_lock_0_CALL`.
2. `std_sync_Condvar_notify_one_0_CALL` sets a token in `CONDVAR_0_NOTIFY`.
3. The token flow continues to `main_BB5` right before `CONDVAR_0_WAIT_START`.

It should be emphasized that no deadlock arises if the transition `CONDVAR_0_WAIT_START` fires *before* `CONDVAR_0_LOST_SIGNAL`. In short, a conflict exists between `CONDVAR_0_WAIT_START` and `CONDVAR_0_LOST_SIGNAL` for the token in `CONDVAR_0_NOTIFY`. Nevertheless, the model checker verifies *all* possible firings and it will uncover the missed signal case without difficulties.

Another noteworthy observation is that this program illustrates the effect that the cleanup paths would have on missed signal detection. If there were a second transition at the same level as `CONDVAR_0_WAIT_START` or `std_sync_Condvar_notify_one_0_CALL`, the token could "escape" to the `PROGRAM_PANIC` place and the deadlock would remain undetected.

It is indispensable for the translation to "force" the Petri net to stay blocked and not open alternative paths that could be used by the model checker to come to the conclusion that the PN never deadlocks.

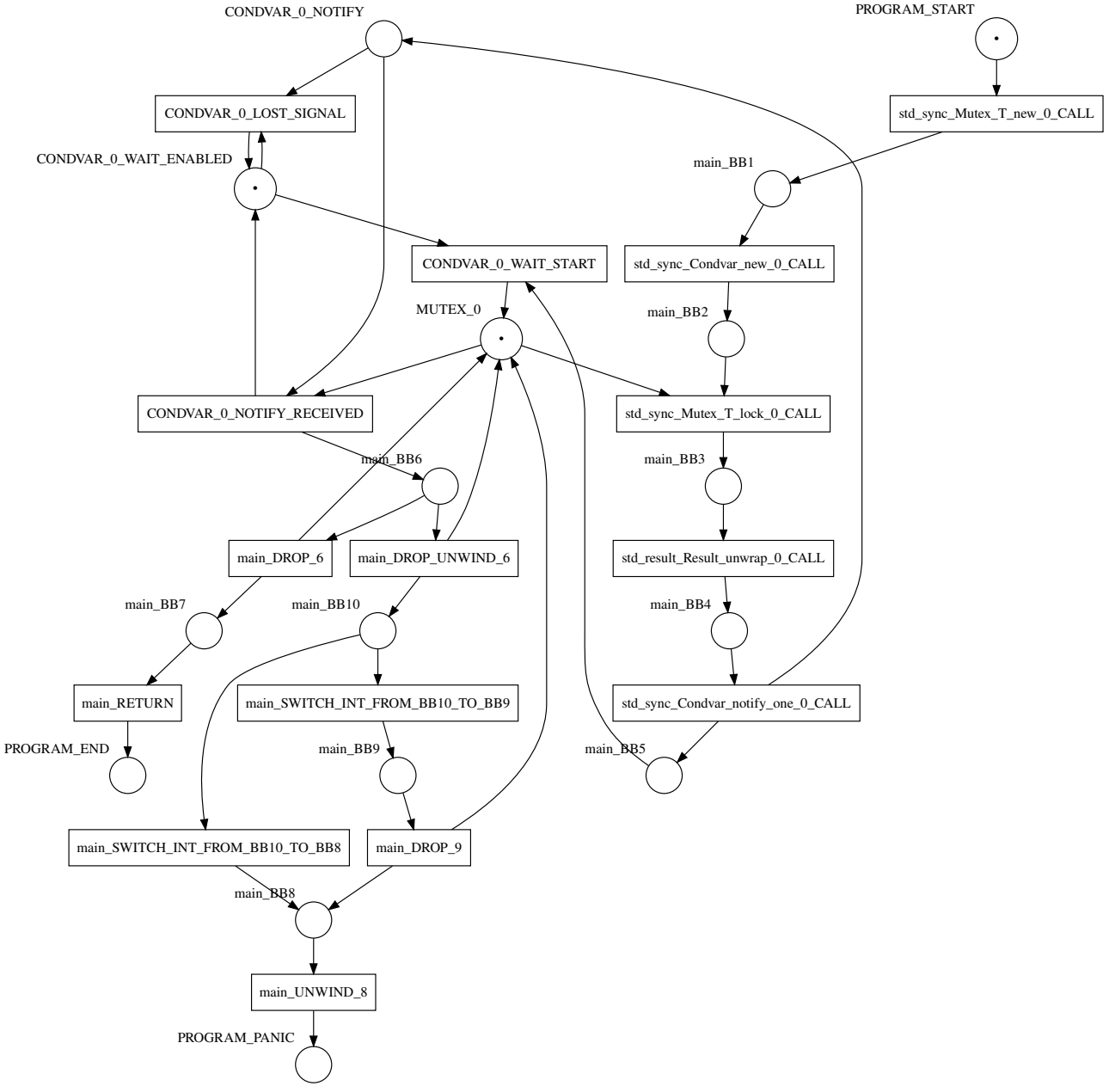

Figure 4.9: The Petri net model for the program in Listing 4.9.

```
fn main() {
      let mutex = std::sync::Mutex::new(false);
      let cvar = std::sync::Condvar::new();
      let mutex_guard = mutex.lock().unwrap();
      cvar.notify_one();
      let _result = cvar.wait(mutex_guard);
}
```

Listing 4.9: A basic program to showcase condition variable translation.

4.8.3 Algorithms for condition variable translation

To wrap up this section, we will present a concise summary of the algorithms utilized in the translation of condition variables. The additions required to the mutex algorithms are included afterward as well.

When a call to `std::sync::Condvar::new` is encountered:

1. Translate the function call using the model seen in Fig. 4.2.
2. Create a new `Condvar`[42] structure with an index to identify it unequivocally across the PN.
3. Link the return value of `std::sync::Condvar::new`, the new condition variable, to the `Condvar` structure.

When a call to `std::sync::Condvar::notify_one` is encountered:

1. Translate the function call using the model seen in Fig. 3.2. Ignore the cleanup place because, otherwise, any call may fail, which amounts to the notify operation not being present in the program, leading to a false lost signal. This is equivalent to assuming that the `notify_one` function never fails.
2. Retrieve the `self` reference to the condition variable on which the function is called.
3. Add an arc from the transition representing the function call to the `notify` place of the underlying condition variable.

When a call to `std::sync::Condvar::wait` or `std::sync::Condvar::wait_while` is encountered:

1. Ignore the cleanup place because, otherwise, any call may fail, which amounts to the wait operation not being present in the program, leading to an incorrect result. We must force the PN to "wait" for the notifying signal to be sent. This is equivalent to assuming that the `wait` or `wait_while` function never fails.

[42] https://github.com/hlisdero/cargo-check-deadlock/blob/main/src/translator/sync/condvar.rs

2. Retrieve the `self` reference to the condition variable on which the function is called.
3. Extract the mutex guard passed into the function.
4. If the condition variable was already connected to a function call, then the translation fails. This enforces Assumptions 1 and 2 seen at the beginning of the section.
5. Otherwise, connect the start and end places to the `wait_start` and `notify_received` transitions respectively.
6. Link the return value, the same mutex guard that was passed as an argument, to the `MutexGuard` structure.
7. Notify the translator that the mutex received must be linked to this `Condvar`. For this purpose, use the enum variant `PostprocessingTask::LinkMutexToCondvar`. This task will be processed after translating all the threads.

When all the threads finished translating, that is, when the queue of threads to process is empty, the `Translator` enters a loop to complete the postprocessing tasks by priority order:

1. Create at the beginning of the loop an empty vector of mutex references.
2. Pop from the `std::collections::BinaryHeap` the task with the lowest priority. This will be by design a `PostprocessingTask::NewMutex`. Add the mutex reference to the vector.
3. After processing all the lower priority tasks, the `Translator` has references to all the mutexes in the code. Continue popping tasks from the priority queue.
4. Eventually, a `PostprocessingTask::LinkMutexToCondvar` is extracted. Link each mutex to the condition variable, which creates the places `condition_set` and `condition_not_set` for the condition. It also connects the `deref_mut` transitions to these places to set the condition. Lastly, it connects the condition places to the condition variable transitions to disable the `wait`.

Modifications to the mutex algorithms

As stated before, the mutex algorithms require some additions to successfully perform missed signal detection.

Add the following to the handler of the `std::sync::Mutex::new` function:

1. Notify the translator that a new mutex has been created. For this purpose, use the enum variant `PostprocessingTask::NewMutex`. This task will be processed after translating all the threads.

When a call to `std::result::Result::<T, E>::unwrap` is encountered:

1. Check that the `self` reference is a mutex or a mutex guard.

2. Translate the function call using the model seen in Fig. 3.2. Ignore the cleanup place because, otherwise, any call may fail, as if the mutex lock operation were not present in the program, leading to a false lost signal. This is equivalent to assuming that the `unwrap` function never fails when applied to a variable linked to a mutex or a mutex guard.

When a call to `std::ops::Deref::deref` or `std::ops::DerefMut::deref_mut` is encountered:

1. Check that the `self` reference is a mutex or a mutex guard.
2. Translate the function call using the model seen in Fig. 3.2. Ignore the cleanup place because, otherwise, any call may fail, as if the mutex lock operation were not present in the program, leading to a false lost signal. This is equivalent to assuming that the `deref` and `deref_mut` functions never fail when dereferencing a variable linked to a mutex or a mutex guard.
3. If the value is being dereferenced mutably (`deref_mut`), extract the first argument passed to the function: The mutex or mutex guard. Add the `deref_mut` transition to the mutex to set the condition for a condition variable in the postprocessing step.
4. Otherwise do nothing. The immutable case does not need to be added to the mutex.

It should now be clear to the reader that the algorithms for missed signal detection are fundamentally of higher complexity and ought to handle more border cases than those for detecting simple deadlocks caused by incorrect usage of mutexes or calling `join` on threads that never terminate.

It is worth mentioning that some border cases arise due to the inclusion of the cleanup logic from the MIR in the PN model. If the implementation instead skipped this, under the hypothesis that Rust standard library functions may never panic, then the algorithms would become simpler. Sec. 7.2 is concerned with the question of not modeling the cleanup paths.

Chapter 5

Testing the implementation

5.1 Generating the MIR

5.2 Visualizing the result

5.3 Unit tests

5.4 Integration tests

Chapter 6

Conclusions

Chapter 7

Future work

7.1 Reducing the size of the Petri net in postprocessing

[Murata, 1989] describes in a Section titled "Simple Reduction Rules for Analysis" six operations that preserve the properties of safeness, liveness, and boundedness of PN. See Definitions 13, 14 and 12 respectively for a refresher of what these properties mean.

The six operations involve simplifications that reduce the number of places or transitions in the Petri net. Next, we reproduce the names used for the reduction rules in the paper and Fig. 7.1 depicts the transformation that takes place in each case.

a) Fusion of Series Places.

b) Fusion of Series Transitions.

c) Fusion of Parallel Places.

d) Fusion of Parallel Transitions.

e) Elimination of Self-Loop Places.

f) Elimination of Self-Loop Transitions.

As these operations do not impact the liveness property, the outcome of the deadlock detection remains unchanged. Consequently, it might be advantageous to reduce the size of the PN after the translation process using specific methods available in the `netcrab` library. This step should be performed after translating all threads but before invoking the model checker.

Incorporating this functionality into the PN library itself would be more suitable, as it would allow other applications to benefit from this feature. It would be interesting to investigate whether this approach proves helpful when translating larger programs that contain hundreds or thousands of places and transitions.

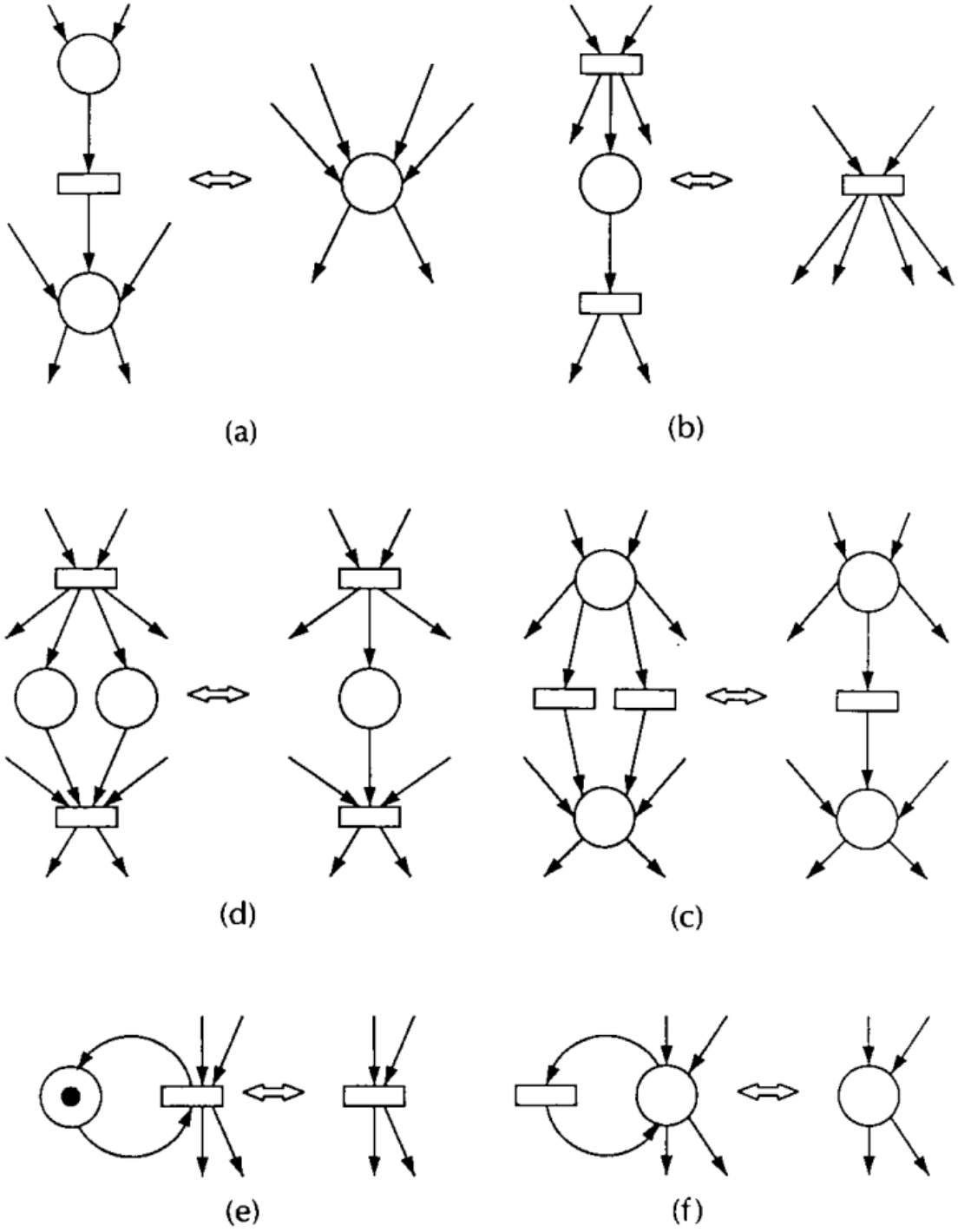

Figure 7.1: The reduction rules presented in Murata's paper.

One notable drawback of applying these operations is that it could obscure the source of the deadlock. It is valuable for the user to have precise information about the line in the source code

where the deadlock occurs. If the corresponding transitions or places representing this line are merged, this information is lost. However, this disadvantage may be deemed acceptable when dealing with extensive models, and the feature could be enabled or disabled at the discretion of the user.

7.2 Eliminating the cleanup paths from the translation

The error handling mechanism in the MIR must account for every possible scenario of failure during runtime. The aim of the *rustc* compiler is to ensure that compiled code fails gracefully, even in the most extreme circumstances, e.g., when the program is running out of memory or system calls fail unexpectedly due to hard limits on the resources available or other causes. However, the majority of this safeguarding cleanup code is never executed in practice. OOM errors and OS failures are uncommon and if they indeed emerge, a deadlock in user code is the least of our problems.

[Meyer, 2020] argues that the program will always terminate in a panic end-state once a single function call or assertion fails. Instead of translating the alternative path that the execution follows in the MIR, he proposes to set a token in the place `PROGRAM_PANIC` directly. This is equivalent to ignoring the specific cleanup target BB during the translation process and connecting the BB to the `PROGRAM_PANIC` place as if it were an `Unwind` terminator (Sec. 4.4.3).

This reduces the size of the Petri net model substantially. It comes with the disadvantage that cleanup BB are visited but never connected to other BB. These must be removed in a postprocessing step to not clutter the final model. Meyer's implementation does not seem to have performed this crucial step. It is unclear whether the implementation matches what the thesis proposed because the source code cannot be compiled anymore and no output examples are present in the repository[1].

The claim that the panic state is unrecoverable necessitates thorough examination, as we have previously observed in the introduction to Rust that the programmer has the option to utilize `std::panic::catch_unwind`. Furthermore, this intuitive reasoning might overlook situations in which a deadlock arises following a panic. This need not be a catastrophic failure. Consider for instance a thread that deadlocks while waiting on a message from another thread that panicked due to incorrect user input.

In conclusion, this modification of the translation logic looks promising to significantly reduce the number of places and transitions in the PN, especially in larger models, but more research is needed.

[1] https://github.com/Skasselbard/Granite

7.3 Translated function cache

A cache that stores functions after translating them is an interesting optimization to explore. The goal is to avoid redundant translations of the same function when it is called multiple times within the program. This idea was already briefly mentioned (but not implemented) in [Meyer, 2020]. The current implementation does not incorporate such caching mechanisms.

This cache would have to store a separate PN for each function. It could be realized as a `HashMap<rustc_hir::def_id::DefId, PetriNet>`, analogous to the function counter already present in the implementation. Furthermore, the translation process would need to merge/connect the Petri nets resulting from each translated function. This merging step requires support from the PN library `netcrab` to combine the multiple subnets into a cohesive whole.

However, connecting the individual Petri nets is not a trivial task, as a function may call an arbitrary number of other functions. Consequently, determining the appropriate "contact points" where the subnet should be connected becomes a challenging endeavor. The potential existence of numerous contact points, arising from the varying function call patterns, thus complicates the merging process.

Additionally, the Petri nets for each function should have labels that are unequivocal across the whole program or at least when exporting them to the format for the model checker. This requires generating slightly different versions of the same function for every call, which partly neglects the benefits of having a cache in the first place.

Lastly, some functions may not be cached at all in the case that special side effects exist. This happens for instance for all synchronization primitives currently supported. Their translation must be handled individually.

7.4 Recursion

Recursion in function calls poses a challenge in PN when defined as in Definition 1 due to the inability to properly map the data values to the model. PN lack the necessary expressive power to represent this compactly.

The number of times a recursive function is called ultimately depends on the data it is called with and cannot be determined at compile time. In normal program execution, a recursive function is pushed onto a new stack frame repeatedly until the base case is reached or the stack overflows. However, in PN, the function call where the base case is reached cannot be distinguished from the others, unless somehow the tokens representing recursion levels are distinct.

[Meyer, 2020, Sec. 3.4.2] discusses this problem and proposes using high-level Petri nets, i.e., Colored Petri nets (CPN) to solve it. High-level Petri nets provide a possible solution by allowing the distinction between tokens and the annotation of tokens with corresponding recursion levels. Nevertheless, this necessitates a serious reconsideration of the entire translation logic

owing to the different formalism. When using CPN each transition becomes a generalized function of input tokens of a specific type that generates tokens of the same or a different type. The resulting Petri net is substantially more complex and not all model checkers support CPN.

Mitigation strategies provide no comfort in this case either. On one hand, one could try to detect recursion and stop the translation, but recursion may exhibit unusual patterns that are not trivial to detect. For instance, consider a function A that calls a function B that calls a function C which finally calls A again. This recursion cycle may be arbitrarily long and adding this capability to the translator does not add much value compared to simply ignoring the problem and reaching a stack overflow.

On the other hand, [Meyer, 2020] suggests modeling each recursion level up to a maximum fixed depth, but this would impact verification results, as the properties of programs could vary with different maximum recursion depths. For every maximum recursion depth N, a counterexample program can be constructed that exhibits a different behavior, e.g., a deadlock, at recursion depth $N + 1$, hence avoiding detection.

7.5 Improvements to the memory model

Despite the seemingly straightforward implementation, devising a memory model that works in all cases is a challenging task. That being said, the current model is primarily a good first approximation and the solution has its drawbacks too.

Passing variables between MIR functions is not supported yet. This is a major drawback since it needs to be solved to support calling methods in `impl` blocks that receive synchronization variables. For this thesis, it was sufficient to write the programs in a simplified way to avoid this limitation but in a real case, this is not feasible.

There is significant coupling between the functions that handle the calls to functions in the `std::sync` module of the standard library and the `Memory`. A more generalized interface could be useful to add support for external libraries.

The idea of "linking" works well but does not match the semantics of Rust programs. In the long run, it would be preferable to delete the mapping if the variable gets moved to a different function. Taking references should also be treated as a distinct case from simply copying or using the variable.

The initial size of the `std::collections::HashMap` could be optimized for the average number of local variables in a typical MIR function. This could be a configuration parameter for the tool.

7.6 Higher-level models

The field of higher-level Petri net models is vast and encompasses numerous branches and potential methodologies. Exploring this domain offers a wide range of possibilities for advancing the modeling capabilities.

One notable advancement lies in the utilization of Colored Petri nets (CPN). Data values could then be modeled as tokens of different types, thereby enhancing the expressiveness and accuracy of the Petri net representation. A related paper in this regard is presented in the next chapter. [Meyer, 2020] also mentioned higher-level models when discussing improvements to his Petri net semantics for Rust. For an introduction to higher-level Petri nets, see [Murata, 1989]

Another intriguing addition to the current Petri net model involves the incorporation of inhibitor arcs. These arcs provide a means to model conditions in the source code where the presence of a zero value is checked. By introducing inhibitor arcs, Petri nets can effectively capture situations where the absence of a specific token is required for a transition to occur. For example, when checking a boolean flag used as a condition for a condition variable. Inhibitor arcs raise the expressive power of Petri nets to the level of Turing machines [Peterson, 1981].

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 and 16th 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.

[Kani Project, 2023] Kani Project (2023). The Kani Rust Verifier. https://model-checking.github.io/kani/. Accessed on 2023-05-30.

[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.

[Kordon et al., 2022] Kordon, F., Bouvier, P., Garavel, H., Hulin-Hubard, F., Amat., N., Amparore, E., Berthomieu, B., Donatelli, D., Dal Zilio, S., Jensen, P., Jezequel, L., He, C., Li, S., Paviot-Adet, E., Srba, J., and Thierry-Mieg, Y. (2022). Complete Results for the 2022 Edition of the Model Checking Contest. http://mcc.lip6.fr/2022/results.php.

[Kordon et al., 2021] Kordon, F., Hillah, L. M., Hulin-Hubard, F., Jezequel, L., and Paviot-Adet, E. (2021). Study of the efficiency of model checking techniques using results of the mcc from 2015 to 2019. *International Journal on Software Tools for Technology Transfer*.

[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.

[Levick, 2022] Levick, R. (2022). Rust Before Main - Rust Linz. https://www.youtube.com/watch?v=q8irLfXwaFM. Accessed on 2023-04-30.

[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 on Embedded Devices Working Group, 2023] Rust on Embedded Devices Working Group (2023). The Embedded Rust Book. https://docs.rust-embedded.org/book/. Accessed on 2023-06-02.

[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 rustup Book. https://rust-lang.github.io/rustup/index.html. Accessed on 2023-05-02.

[Rust Project, 2023d] Rust Project (2023d). 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.

[Schmidt, 2000] Schmidt, K. (2000). Lola a low level analyser. In *Application and Theory of Petri Nets 2000: 21st International Conference, ICATPN 2000 Aarhus, Denmark, June 26–30, 2000 Proceedings 21*, pages 465–474. Springer.

[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.

[Thierry Mieg, 2015] Thierry Mieg, Y. (2015). Symbolic Model-Checking using ITS-tools. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035 of *Lecture Notes in Computer Science*, pages 231–237, London, United Kingdom. Springer Berlin Heidelberg.

[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.

[Weber and Kindler, 2003] Weber, M. and Kindler, E. (2003). The Petri Net Markup Language. *Petri Net Technology for Communication-Based Systems: Advances in Petri Nets*, pages 124–144.

[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*.