Universidad de Buenos Aires

tesis de grado de ingeniería en informática

Compile-time Deadlock Detection in Rust using Petri Nets

Autor:

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

Director:

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

Departamento de Computación

Facultad de Ingeniería

10 de marzo de 2023

Contents

Chapter 1

Introduction

1.1 Petri nets

1.1.1 Overview

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

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

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

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

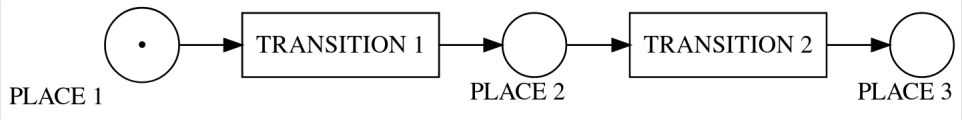

Figure 1.1: Example of a Petri net. PLACE 1 contains a token.

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

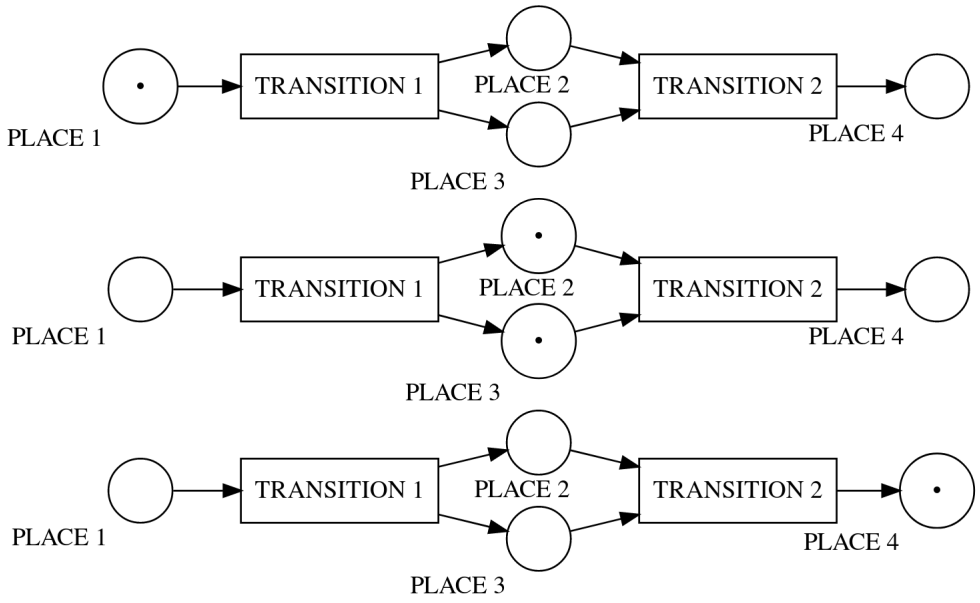

Figure 1.2: Example of transition firing: Transition 1 fires first, then transition 2 fires.

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

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

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. 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.

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

Definition 6 (Source transition)**.** A transition without any input place is called a *source transition*.

Definition 7 (Sink transition)**.** A transition without any output place is called a *sink transition*.

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

1.1.4 Modeling examples

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

Finite-state machines

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

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

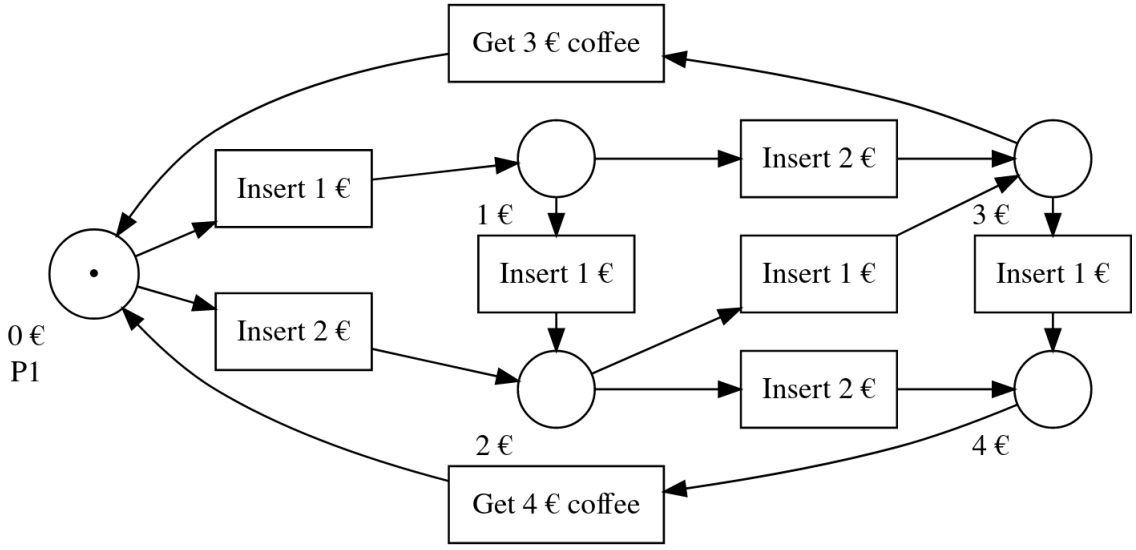

Figure 1.4: The Petri net for a coffee vending machine. It is equivalent to a state diagram.

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

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

Definition 8 (State machines)**.** A Petri net in which each transition has exactly one incoming arc and exactly one outgoing arc is known as a *state machine*.

Any finite-state machine (or its state diagram) can be modeled with a state machine.

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

Parallel activities

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

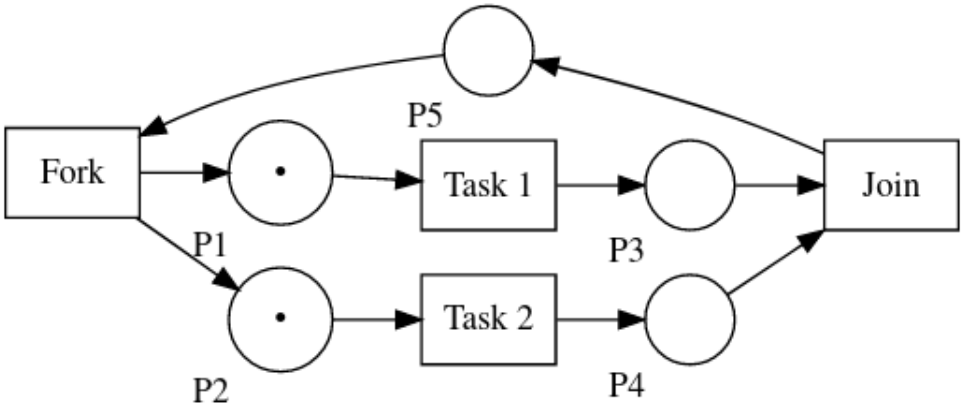

Figure 1.5: The Petri net depicting two parallel activities in a fork-join fashion.

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

Definition 9 (Concurrency in Petri nets)**.** Two transitions are said to be *concurrent* if they are causally independent, i.e. the firing of one transition does not cause and is not triggered by the firing of the other.

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

Definition 10 (Marked graphs)**.** A Petri net in which each place has exactly one incoming arc and exactly one outgoing arc is known as a *marked graph*.

Communication protocols

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

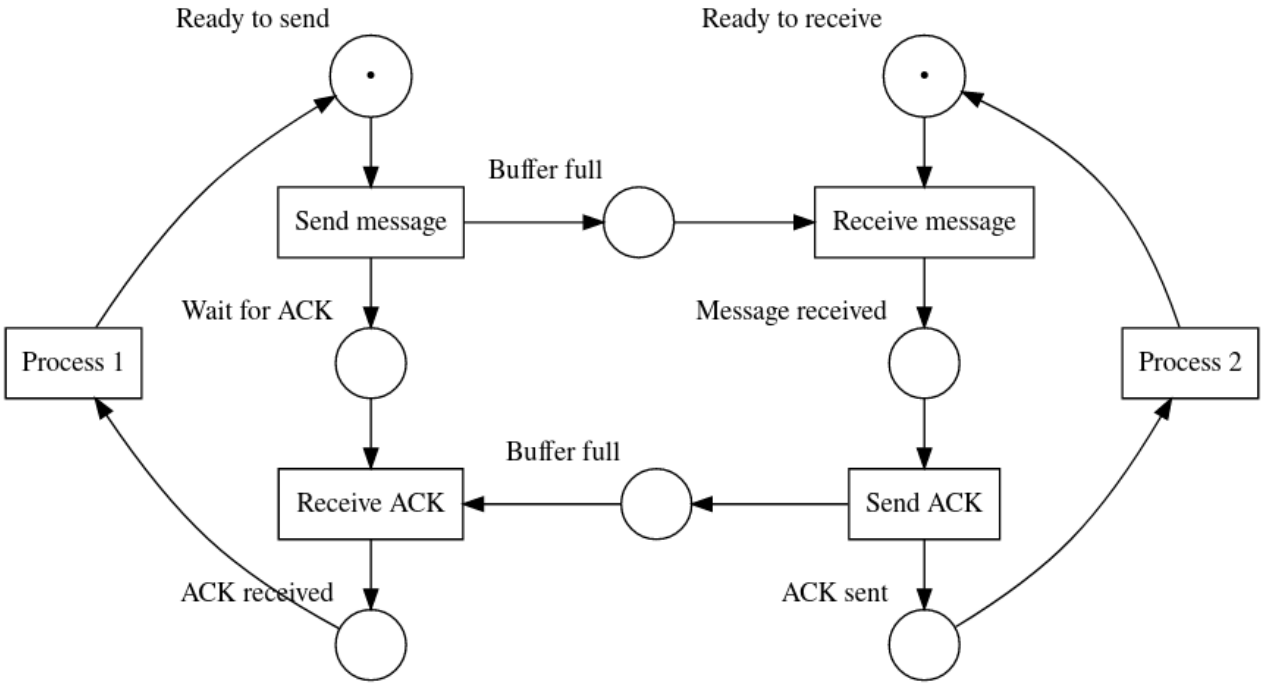

Figure 1.6: A simplified Petri net model of a communication protocol.

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

Synchronization control

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

A Petri net for a readers-writers system with k processes is shown in Fig. 1.7. Each token represents a process and the choice of t_1 or t_2 represents whether the process performs a read or a write operation.

It makes use of weighted edges to remove atomically $k-1$ tokens from p_3 before performing a write (transition t_2), 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 p_2 will be empty. It can be easily verified that the mutual exclusion property is satisfied for the system.

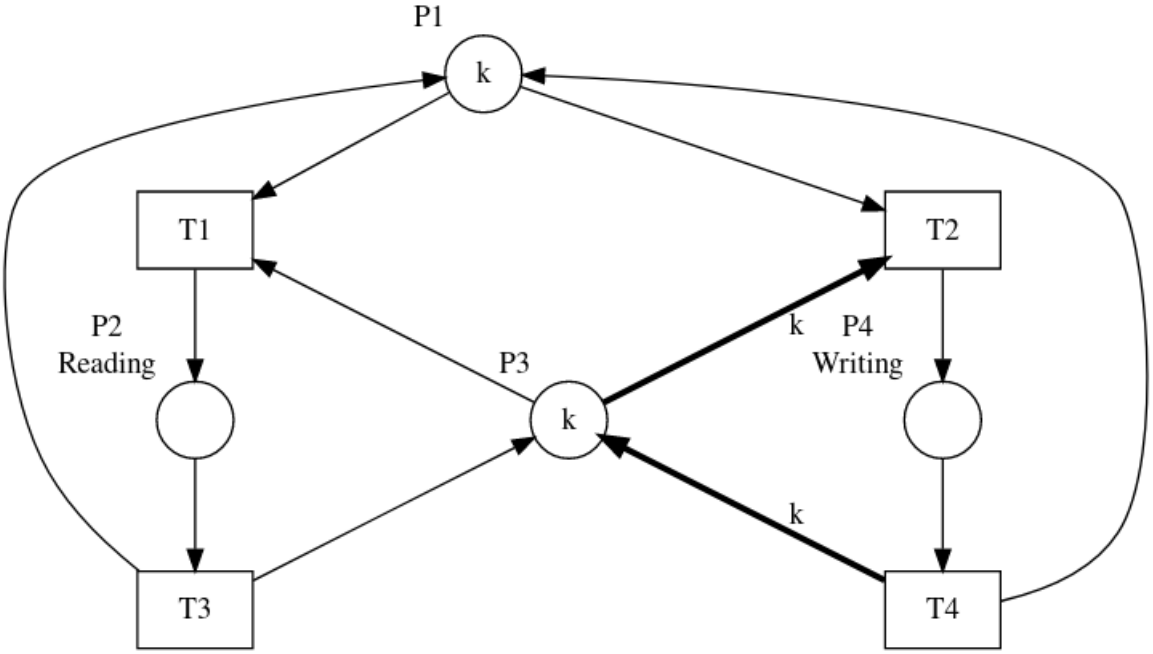

Figure 1.7: A Petri net system with k processes that either read or write.

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

1.2 The Rust programming language

1.3 Deadlocks

1.4 Lost signals

1.5 Compiler arquitecture

1.6 Model checking

Chapter 2

Design of the proposed solution

Chapter 3

Testing the implementation

3.1 Unit tests

3.2 Integration tests

3.3 Generating the MIR representation

3.4 Visualizing the result

Chapter 4

Conclusions

Chapter 5

Future work

Chapter 6

Related work

Bibliography

[Murata, 1989] Murata, T. (1989). Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4).

[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.