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 digraph 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 1.1.

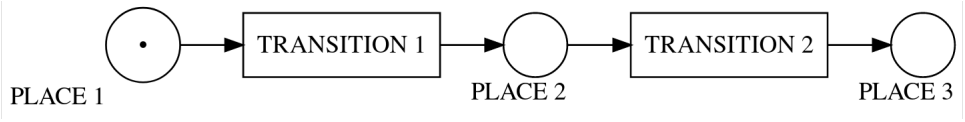

Figure 1.1: Example of a Petri net. PLACE 1 contains a token.

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 1.2, we can see how successive firings happen.

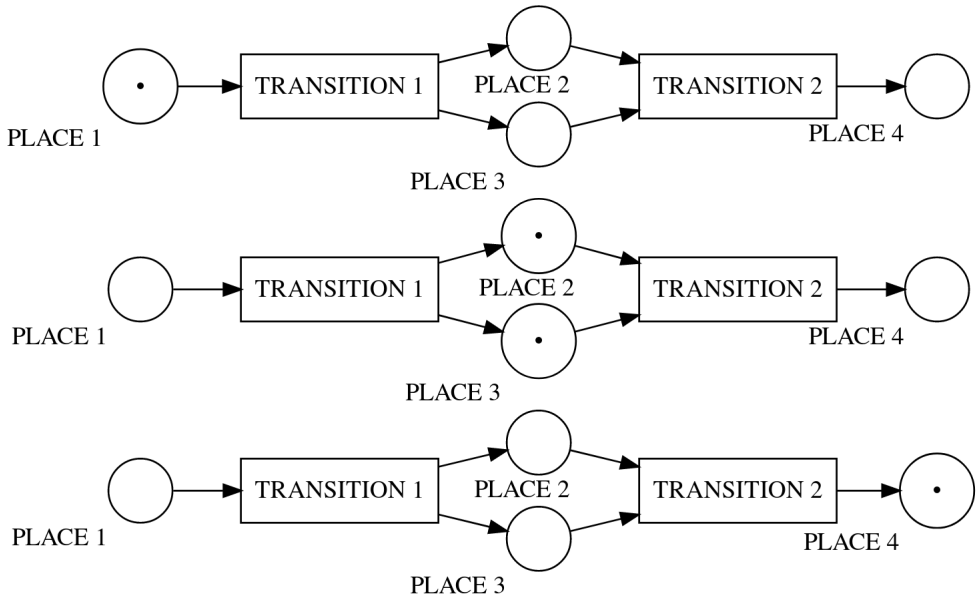

Figure 1.2: Example of transition firing. First, transition 1 fires, then transition 2 fires.

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

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

2.1 Rust compiler: *rustc*

2.2 Mid-level Intermediate Representation (MIR)

2.3 Entry point for the translation

2.4 Function calls

2.5 Function memory

2.6 MIR function

2.6.1 Basic blocks

2.6.2 Statements

2.6.3 Terminators

2.7 Panic handling

2.8 Multithreading

2.9 Emulation of Rust synchronization primitives

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

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

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

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

Chapter 3

Testing the implementation

3.1 Unit tests

3.2 Integration tests

3.3 Generating the MIR representation

3.4 Visualizing the result

Chapter 4

Conclusions

Chapter 5

Future work

Chapter 6

Related work

Bibliography

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

[Petri, 1962] Petri, C. A. (1962). Kommunikation mit Automaten. *Institut für Instrumentelle Mathematik*, 3.