UNIVERSIDAD DE BUENOS AIRES

TESIS DE GRADO DE INGENIERÍA EN INFORMÁTICA

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

Autor: Horacio Lisdero Scaffino
hlisdero@fi.uba.ar

Director: Ing. Pablo Andrés Deymonnaz
pdeymon@fi.uba.ar

Departamento de Computación
Facultad de Ingeniería

8 de junio de 2023

Índice general

Índice de figuras

List of Listings

Siglas

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

Capítulo 1

Introducción

Para comprender plenamente el alcance y el contexto de este trabajo, es beneficioso proporcionar algunos temas de fondo que sientan las bases de la investigación. Estos temas de fondo sirven como bloques teóricos sobre los que se construye la traducción.

En primer lugar, se presenta la teoría de las redes de Petri tanto gráficamente como en términos matemáticos. Para ilustrar el poder de modelado y la versatilidad de las redes de Petri, se proporcionan al lector varios modelos diferentes a modo de ejemplo. Estos modelos muestran la capacidad de las redes de Petri para capturar diversos aspectos de los sistemas concurrentes y representarlos de forma visual e intuitiva. Más adelante, se introducen algunas propiedades importantes y se explica el análisis de alcanzabilidad que realiza el verificador de modelos.

En segundo lugar, se analiza brevemente el lenguaje de programación Rust y sus principales características. Se incluye un puñado de ejemplos de aplicaciones notables de Rust en la industria. Se reúnen pruebas convincentes del uso de lenguajes con un manejo seguro de la memoria para argumentar que Rust proporciona una base excelente para ampliar la detección de clases de errores en tiempo de compilación.

En tercer lugar, se ofrece información general sobre el problema de los bloqueos mutuos y las señales perdidas cuando se utilizan *condition variables*, así como una descripción de las estrategias habituales utilizadas para resolver estos problemas.

Por último, se ofrece una visión general de la arquitectura de los compiladores y del concepto de verificación de modelos. Señalaremos el potencial aún sin explorar que subyace a la verificación formal para aumentar la seguridad y fiabilidad de los sistemas de software.

1.1. Redes de Petri

1.1.1. Visión general

Las redes de Petri (Petri nets (PN)) son una herramienta de modelado gráfico y matemático utilizada para describir y analizar el comportamiento de los sistemas concurrentes. Fueron introducidas por el investigador alemán Carl Adam Petri en su tesis doctoral [Petri, 1962] y desde entonces se han aplicado en diversos campos como la informática, la ingeniería y la biología. Puede encontrar un resumen conciso de la teoría de las redes de Petri, sus propiedades, análisis y aplicaciones en [Murata, 1989].

Una red de Petri es un grafo dirigido bipartito formado por un conjunto de lugares, transiciones y arcos. Hay dos tipos de nodos: lugares y transiciones. Los lugares representan el estado del sistema, mientras que las transiciones representan eventos o acciones que pueden ocurrir. Los arcos conectan lugares a transiciones o transiciones a lugares. No puede haber arcos entre dos lugares o entre dos transiciones, preservando así la propiedad bipartita.

Los lugares pueden contener cero o más marcas o fichas. Los tokens se utilizan para representar la presencia o ausencia de entidades en el sistema, como recursos, datos o procesos. En la clase más simple de redes de Petri, los tokens no llevan ninguna información y son indistinguibles unos de otros. El número de fichas en un lugar o la simple presencia de una ficha es lo que transmite significado en la red. Las fichas se consumen y se producen al dispararse las transiciones, lo que da la impresión de que se mueven a través de los arcos.

En la representación gráfica convencional, los lugares se representan mediante círculos, mientras que las transiciones se representan como rectángulos. Las fichas se representan como puntos negros dentro de los lugares, como se ve en la Fig. 1.1.

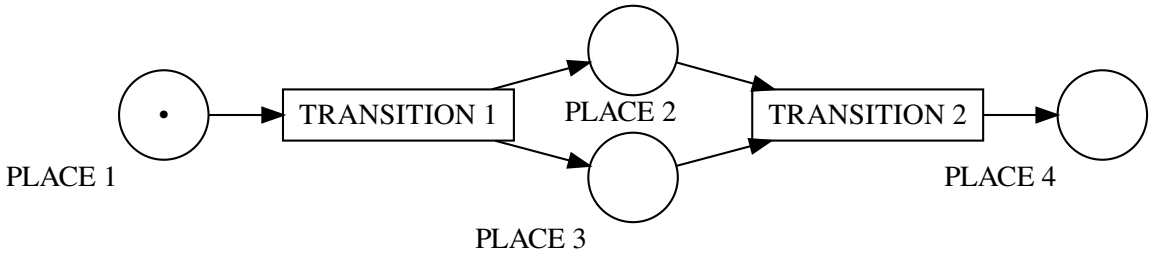

Figura 1.1: Ejemplo de una red de Petri. PLACE 1 contiene una marca.

Cuando una transición se dispara, consume fichas de sus lugares de entrada y produce fichas en sus lugares de salida, lo que refleja un cambio en el estado del sistema. El disparo de una transición se activa cuando hay suficientes fichas en sus lugares de entrada. En la Fig. 1.2, podemos ver cómo se producen los disparos uno detrás del otro.

El disparo de las transiciones habilitadas no es determinista, es decir, se disparan aleatoriamente

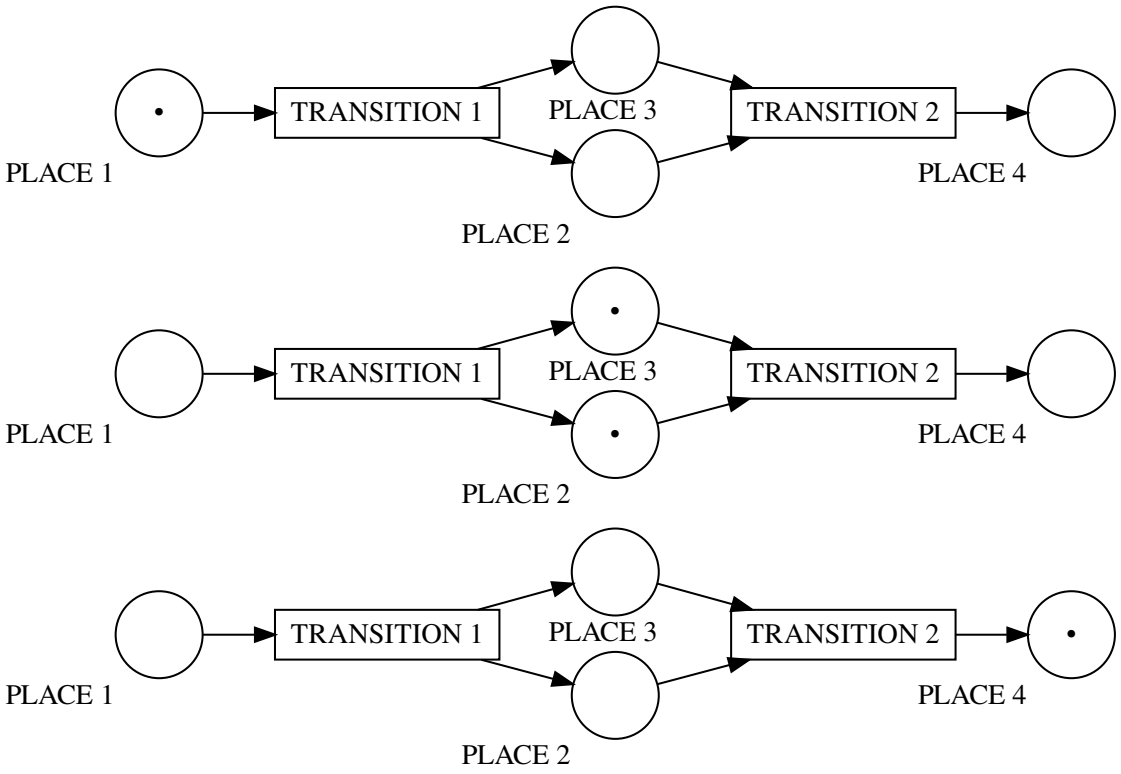

Figura 1.2: Ejemplo de disparo de una transición: La transición 1 se dispara primero, luego se dispara la transición 2.

mientras estén habilitadas. Una transición deshabilitada se considera *muerta* si no hay ningún estado alcanzable en el sistema que pueda llevar a que la transición se habilite. Si todas las transiciones de la red están muertas, entonces la red también se considera *muerta*. Este estado es análogo al bloqueo (*deadlock*) de un programa informático.

Las redes de Petri pueden utilizarse para modelar y analizar una amplia gama de sistemas, desde sistemas sencillos con unos pocos componentes hasta sistemas complejos con muchos componentes que interactúan entre sí. Pueden utilizarse para detectar problemas posibles en un sistema, optimizar su rendimiento y diseñar e implementar sistemas de forma más eficaz.

También pueden utilizarse para modelar procesos industriales [Van der Aalst, 1994], para validar requisitos de software expresados como casos de uso [Silva and Dos Santos, 2004] o para especificar y analizar sistemas en tiempo real [Kavi et al., 1996].

En concreto, las redes de Petri pueden utilizarse para detectar deadlocks en el código fuente modelando el programa de entrada como una red de Petri y analizando después la estructura de la red resultante. Se demostrará que este enfoque es formalmente sólido y practicable para el código fuente escrito en el lenguaje de programación Rust.

1.1.2. Modelo matemático formal

Una red de Petri es un tipo particular de grafo bipartito, con pesos y dirigido, dotado de un estado inicial denominado *marcado inicial*, M_0. Para este trabajo, se utilizará la siguiente definición general de una red de Petri tomada de [Murata, 1989].

Definition 1: Petri net

Una red de Petri es una 5-tupla, $PN = (P, T, F, W, M_0)$ *donde:*
$P = \{p_1, p_2, \ldots, p_m\}$ *es un conjunto finito de lugares,*
$T = \{t_1, t_2, \ldots, t_n\}$ *es un conjunto finito de transiciones,*
$F \subseteq (P \times T) \cup (T \times P)$ *es un conjunto de arcos (relación de flujo),*
$W : F \leftarrow \{1, 2, 3, ...\}$ *es una función de peso para los arcos,*
$M_0 : P \leftarrow \{0, 1, 2, 3, ...\}$ *es el marcado inicial,*
$P \cap T = \varnothing$ *y* $P \cup T \neq \varnothing$

En la representación gráfica, los arcos se etiquetan con su peso que es un número entero no negativo k. Normalmente el peso se omite si es igual a 1. Un arco con peso k puede interpretarse como un conjunto de k arcos paralelos distintos.

Un *marcado (estado)* asocia a cada lugar un número entero no negativo l. Si un marcado asigna al lugar p un número entero no negativo l, decimos que p está *marcado con* l *marcas o tokens*. Pictóricamente, denotamos esto colocando l puntos negros (fichas) en el lugar p. El *p-ésimo* componente de M, denotado por $M(p)$, es el número de fichas en el lugar p.

Una definición alternativa de las redes de Petri utiliza un multiconjunto (*bag*) en lugar de un conjunto para definir los arcos, permitiendo así la presencia de múltiples elementos. Puede encontrarse en la literatura, por ejemplo, [Peterson, 1981, Definition 2.3].

Como ejemplo, consideremos la red de Petri $PN_1 = (P, T, F, W, M)$ donde:

$P = \{p_1, p_2\}$,
$T = \{t_1, t_2\}$,
$F = \{(p_1, t_1), (p_2, t_2), (t_1, p_2), (t_2, p_2)\}$,
$W(a_i) = 1 \quad \forall a_i \in F$
$M(p_1) = 0, M(p_2) = 0$

Esta red no contiene fichas y todos los pesos de los arcos son iguales a 1. Se muestra en la Fig. 1.3.

La Fig. 1.3 contiene una estructura interesante que encontraremos más adelante. Esto motiva la siguiente definición.

Figura 1.3: Example of a small Petri net containing a self-loop.

Definition 2: Bucle

Un lugar p y una transición t definen un bucle si p es a la vez un lugar de entrada y un lugar de salida de t.

En la mayoría de los casos, nos interesan las redes de Petri que no contienen bucles las cuales se denominan *puras*.

Definition 3: Red de Petri pura

Se dice que una red de Petri es pura si no tiene bucles.

Además, si el peso de cada arco es igual a uno, llamamos a la red de Petri *ordinaria*.

Definition 4: Red de Petri ordinaria

Se dice que una red de Petri es ordinaria si todos los pesos de sus arcos son 1, es decir,

$$W(a) = 1 \quad \forall a \in F$$

1.1.3. Disparo de transiciones

La regla de disparo de transición es el concepto central de las redes de Petri. A pesar de ser aparentemente simple, sus implicaciones son de gran alcance y complejidad.

Definition 5: Regla de disparo de transiciones

Sea $PN = (P, T, F, W, M_0)$ una red de Petri.

(I) Se dice que una transición t está habilitada si cada lugar de entrada p de t marcado con al menos $W(p,t)$ marcas donde $W(p,t)$ es el peso del arco que va de p de t.

(II) Una transición activada puede dispararse o no, dependiendo de si el evento tiene lugar o no.

(III) El disparo de una transición activada t elimina $W(t,p)$ marcas de cada lugar de entrada p de t donde $W(t,p)$ es el peso del arco de t a p.

Siempre que se habiliten varias transiciones para un marcado M dado, puede dispararse cualquiera de ellas. La elección es no determinista. Se dice que dos transiciones habilitadas están en *conflicto* si el disparo de una de ellas inhabilita la otra transición. En este caso, las transiciones compiten por la ficha colocada en un lugar de entrada compartido.

Si dos transiciones t_1 y t_2 están habilitadas en algún marcado pero no están en conflicto, pueden dispararse en cualquier orden, es decir, t_1 luego t_2 o t_2 luego t_1. Tales transiciones representan eventos que pueden ocurrir concurrentemente o en paralelo. En este sentido, el modelo de red de Petri adopta un modelo de paralelismo basado en el intercalado (*interleaved model of parallelism*), es decir, el comportamiento del sistema es el resultado de un intercalado arbitrario de los eventos paralelos.

Las transiciones sin lugares de entrada ni lugares de salida reciben un nombre especial.

Definition 6: Transición fuente (*Source transition*)

Una transición sin ningún lugar de entrada se denomina transición fuente.

Definition 7: Transición sumidero (*Sink transition*)

Una transición sin lugar de salida se denomina transición de sumidero.

Cabe destacar que una transición fuente se activa incondicionalmente y produce fichas sin consumir ninguna, mientras que el disparo de una transición sumidero consume fichas sin producir ninguna.

1.1.4. Simuladores en línea

Para familiarizarse con la dinámica de las redes de Petri, resulta útil simular algunos ejemplos en línea, ya que ver una red de Petri en acción es más claro que cualquier explicación estática sobre el papel. Hemos reunido algunas herramientas con este fin para aliviar la carga del lector.

- Puede encontrar un sencillo simulador hecho por Igor Kim en `https://petri.hp102.ru/`. La herramienta incluye un vídeo tutorial en Youtube y redes de ejemplo.
- Como complemento, el profesor Wil van der Aalst de la Universidad de Hamburgo ha elaborado una serie de tutoriales interactivos. Estos tutoriales son archivos de Adobe Flash Player (con extensión `.swf`) que los navegadores web modernos no pueden ejecutar. Por suerte, se puede utilizar un emulador Flash en línea como el que se encuentra en `https://flashplayer.fullstacks.net/?kind=Flash_Emulator` para cargar los archivos y ejecutarlos.
- Otro editor y simulador de redes de Petri en línea es `http://www.biregal.com/`. El usuario puede dibujar la red, añadir los tokens y luego disparar manualmente las transiciones.

1.1.5. Ejemplos de modelado

En esta subsección, se presentan varios ejemplos sencillos para introducir algunos conceptos básicos de las redes de Petri que son útiles en el modelado. Esta subsección se ha adaptado de [Murata, 1989].

Para otros ejemplos de modelado, como el problema de exclusión mutua, los semáforos propuestos por Edsger W. Dijkstra, el problema del productor/consumidor y el problema de los filósofos cenando, se remite al lector a [Peterson, 1981, Chap. 3] y [Reisig, 2013].

Máquinas de estado finito

Las máquinas de estados finitos (Finite-state machine (FSM)) pueden representarse mediante una subclase de redes de Petri.

Como ejemplo de máquina de estado finito, consideremos una máquina expendedora de café. Acepta monedas de 1 € o 2 € euros y vende dos tipos de café, el primero cuesta 3 € euros y el segundo 4 € euros. Supongamos que la máquina puede contener hasta 4 € y no devuelve ningún cambio. Entonces, el diagrama de estados de la máquina puede representarse mediante la red de Petri que se muestra en la Fig. 1.4.

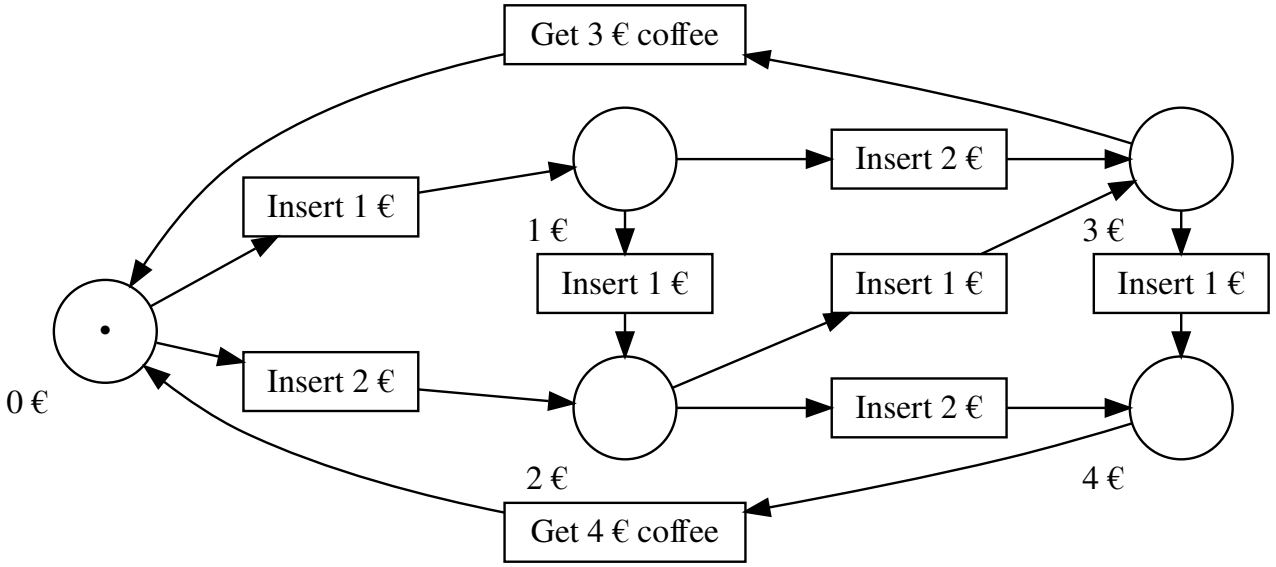

Figura 1.4: La red de Petri para una máquina expendedora de café, es equivalente a un diagrama de estados.

Las transiciones representan la inserción de una moneda del valor etiquetado, por ejemplo, "Insert 1 € coin". Los lugares representan un posible estado de la máquina, es decir, la cantidad de dinero almacenada actualmente en su interior. El lugar situado más a la izquierda, etiquetado "0 €", está marcado con una ficha y corresponde al estado inicial del sistema.

Ahora podemos presentar la siguiente definición de esta subclase de redes de Petri.

Definition 8: Máquinas de estado

Una red de Petri en la que cada transición tiene exactamente un arco entrante y exactamente un arco saliente se conoce como máquina de estados.
Cualquier FSM (o su diagrama de estados) puede modelarse con una máquina de estados.

La estructura de un lugar p_1 que tiene dos (o más) transiciones de salida t_1 y t_2 se denomina conflicto, decisión o elección, según la aplicación en cuestión. Esto se ve en el lugar inicial de la Fig. 1.4, donde el usuario debe seleccionar qué moneda introducir al principio.

Actividades en paralelo

A diferencia de las máquinas de estados finitos, las redes de Petri también pueden modelar actividades paralelas o concurrentes. En la Fig. 1.5 se muestra un ejemplo de ello, en el que la red representa la división de una tarea mayor en dos subtareas que pueden ejecutarse en paralelo.

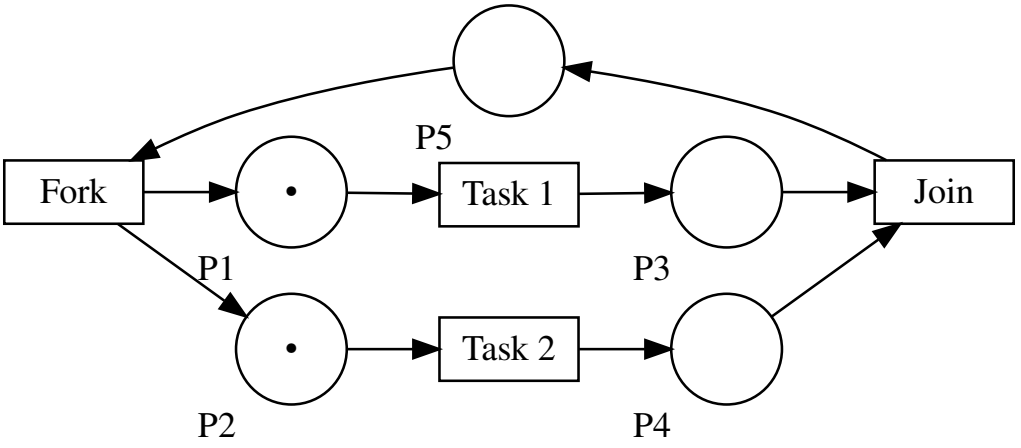

Figura 1.5: La red de Petri que representa dos actividades paralelas en forma de bifurcación.

La transición "Fork" se disparará antes que "Task 1" y "Task 2" y que "Join" sólo se disparará después de ambas tareas se completan. Pero tenga en cuenta que el orden en que se ejecutan la "Task 1" y la "Task 1" no es determinista. La tarea 1 podría dispararse antes, después o al mismo tiempo que la tarea 2. Es precisamente esta propiedad de la regla de disparo en las redes de Petri la que permite modelar sistemas concurrentes.

Definition 9: Concurrencia en redes de Petri

Se dice que dos transiciones son concurrentes si son causalmente independientes, es decir, el disparo de una transición no causa ni es provocado por el disparo de la otra.

Observe que cada lugar de la red de la Fig. 1.5 tiene exactamente un arco entrante y un arco saliente. Esta subclase de redes de Petri permite representar la concurrencia pero no las decisiones (conflictos).

Definition 10: Grafos marcados (*marked graphs*)

Una red de Petri en la que cada lugar tiene exactamente un arco entrante y exactamente un arco saliente se conoce como grafo marcado (marked graph).

Protocolos de comunicación

Los protocolos de comunicación también pueden representarse en redes de Petri. La fig. 1.6 ilustra un protocolo sencillo en el que el Proceso 1 envía mensajes al Proceso 2 y espera a recibir un acuse de recibo antes de continuar. Ambos procesos se comunican a través de un canal con búfer cuya capacidad máxima es de un mensaje. Por lo tanto, sólo un mensaje puede estar viajando entre los procesos en un momento dado. Para simplificar, no se ha incluido ningún mecanismo de *timeout*.

Se podría incorporar al modelo un tiempo de espera máximo para la operación de envío añadiendo una transición $t_{timeout}$ con aristas de "Wait for ACK" a "Ready to send". Esto mapea la decisión entre recibir el acuse de recibo y el tiempo de espera.

Control de sincronización

En un sistema multihilo, los recursos y la información se comparten entre varios hilos. Esta compartición debe controlarse o sincronizarse para garantizar el correcto funcionamiento del sistema global. Las redes de Petri se han utilizado para modelar diversos mecanismos de sincronización, incluidos los problemas de exclusión mutua, lectores-escritores y productores-consumidores [Murata, 1989].

En la Fig. 1.7 se muestra una red de Petri para un sistema de lectores-escritores con k procesos. Cada marca representa un proceso y la elección de T1 o T2 representa si el proceso realiza una operación de lectura o de escritura.

Utiliza aristas ponderadas para eliminar atómicamente $k-1$ tokens de P3 antes de realizar una escritura (transición T2), evitando así que los lectores entren en el bucle derecho de la red.

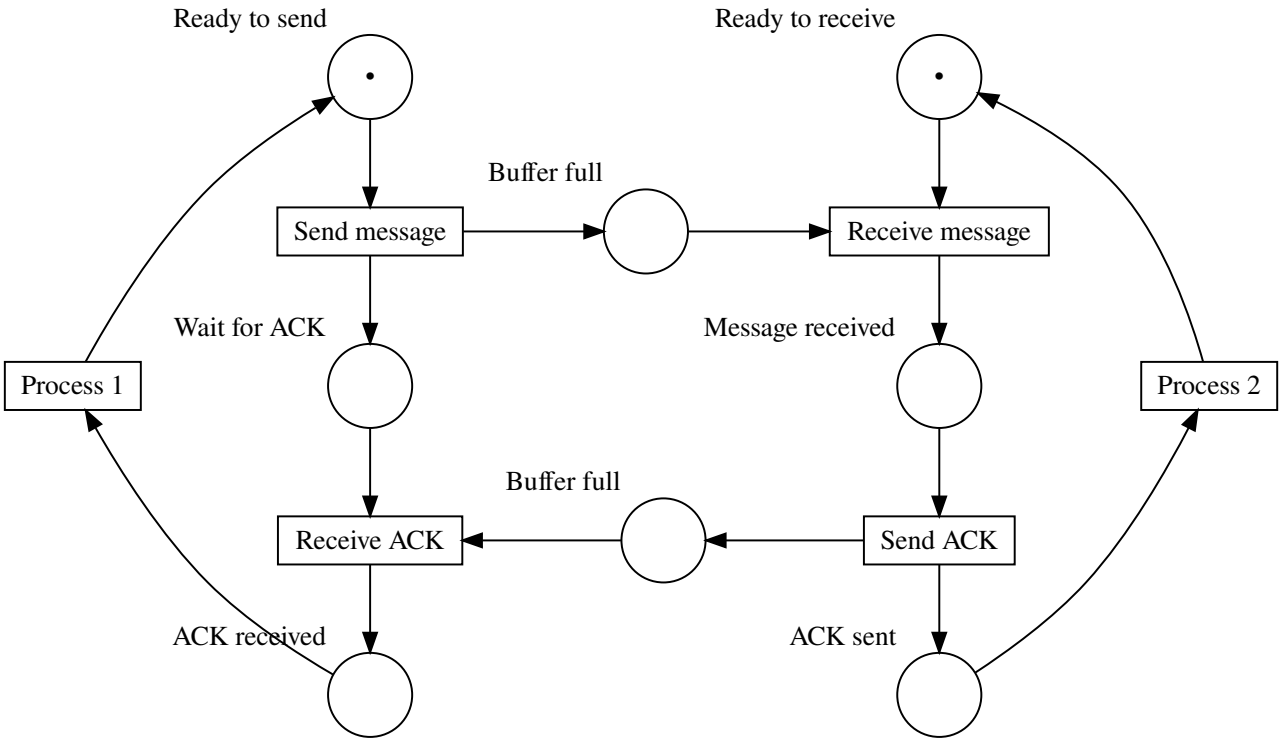

Figura 1.6: Modelo simplificado de red de Petri de un protocolo de comunicación.

Como máximo k procesos pueden estar leyendo al mismo tiempo, pero cuando un proceso esté leyendo, ningún proceso podrá leer, es decir, P2 estará vacío. Se puede comprobar fácilmente que la propiedad de exclusión mutua se satisface para el sistema.

Hay que señalar que este sistema no está libre de inanición (*starvation*), ya que no hay garantía de que una operación de escritura vaya a producirse en algún momento. Por otro lado, el sistema sí está libre de deadlocks.

1.1.6. Propiedades importantes

En esta subsección veremos conceptos fundamentales para el análisis de redes de Petri que facilitarán la comprensión de las redes que trataremos en el resto del trabajo.

Alcanzabilidad

La alcanzabilidad es una de las cuestiones más importantes cuando se estudian las propiedades dinámicas de un sistema. El disparo de transiciones habilitadas provoca cambios en la ubicación de las marcas. En otras palabras, cambia el marcado M. Una secuencia de disparos crea una

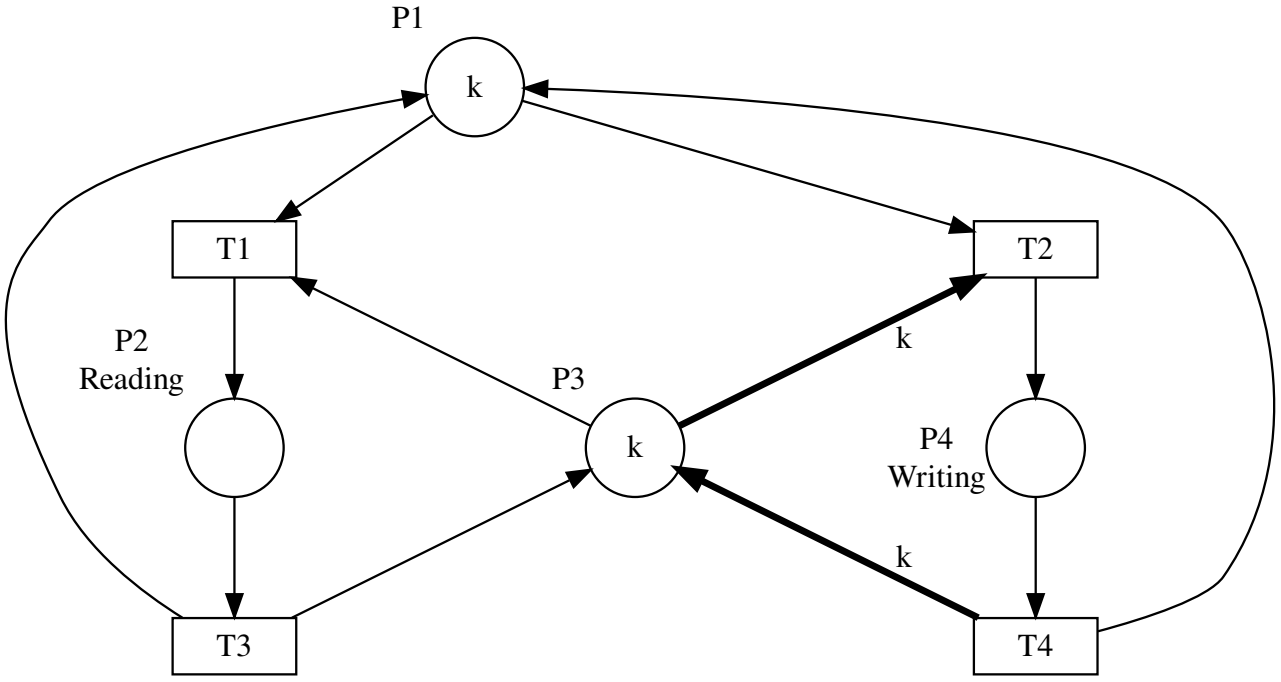

Figura 1.7: Un sistema de redes de Petri con k procesos que leen o escriben.

secuencia de marcados en la que cada marcado puede denotarse como un vector de longitud n, siendo n el número de lugares de la red de Petri.

Una *secuencia de disparos* u *ocurrencias* se denota por $\sigma = M_0\ t_1\ M_1\ t_2\ M_2\ \cdots\ t_l\ M_l$ o simplemente $\sigma = t_1\ t_2\ \cdots\ t_l$, ya que las marcas resultantes de cada disparo se derivan de la regla de disparo de transición descrita en la Sec. 1.1.3.

Definition 11: Alcanzabilidad (*Reachability*)

Decimos que una marca M es alcanzable desde M_0 si existe una secuencia de disparo σ tal que M está contenida en σ.

El conjunto de todas las marcas posibles alcanzables desde M_0 se denota por $R(N, M_0)$ o más sencillamente $R(M_0)$ cuando la red en cuestión es obvia. Este conjunto se denomina *conjunto de alcanzabilidad*.

Se puede presentar entonces un problema de suma importancia en la teoría de las redes de Petri, a saber, el *Problema de alcanzabilidad*: Encontrar si $M_n \in R(M_0, N)$ para una red y un marcado inicial dados.

En algunas aplicaciones, sólo nos interesan los marcados de un subconjunto de lugares y podemos ignorar los restantes. Esto da lugar a una variación del problema conocida como *problema de alcanzabilidad del submarcado* (*submarking reachability problem*).

Se ha demostrado que el problema de la alcanzabilidad es decidible [Mayr, 1981]. Sin embargo, también se ha demostrado que ocupa un espacio exponencial (formalmente, es EXPSPACE-hard) [Lipton, 1976]. Se han propuesto nuevos métodos para que los algoritmos sean más eficientes [Küngas, 2005]. Recientemente, [Czerwiński et al., 2020] mejoraron el límite inferior y demostraron que el problema no es ELEMENTARY. Estos resultados ponen de relieve que el problema de la alcanzabilidad sigue siendo un área activa de investigación en teória de la computación.

Para éste y otros problemas clave, los resultados teóricos más importantes obtenidos hasta 1998 se detallan en [Esparza and Nielsen, 1994].

Acotamiento y seguridad

Durante la ejecución de una red de Petri, los tokens pueden acumularse en algunos lugares. Diversas aplicaciones sulen necesitar garantizar que el número de fichas en un lugar determinado no supere una cierta tolerancia. Por ejemplo, si un lugar representa un búfer, nos interesa que el búfer nunca se desborde.

Definition 12: Acotamiento (*Boundedness*)

Un lugar de una red de Petri es k-acotada o es k-seguro si el número de fichas de ese lugar no puede superar un número entero finito k para cualquier marcado alcanzable desde M_0.
Una red de Petri es k-acotada o simplemente acotada si todos los lugares están acotados.

La seguridad es un caso especial de la acotación. Aplica cuando el lugar contiene 1 ó 0 fichas durante la ejecución.

Definition 13: Seguridad (*Safeness*)

Un lugar de una red de Petri es seguro si el número de fichas de ese lugar nunca es superior a uno.
Una red de Petri es segura si cada lugar de esa red es seguro.

Las redes de las Fig. 1.4, 1.5 y 1.6 son todas seguras.

La red de la Fig. 1.7 es k-acotada porque todos sus lugares son k-acotados.

Liveness

El concepto de liveness es análogo a la ausencia total de deadlocks en los programas informáticos.

Definition 14: Liveness

Se dice que una red Patri (N, M_0) está viva (o equivalentemente se dice que M_0 es una marca viva (live) para N) si, para cada marca alcanzable desde M_0, es posible disparar cualquier transición de la red progresando a través de alguna secuencia de disparo.

Cuando una red está viva, siempre puede seguir ejecutándose, sin importar las transiciones que se dispararon antes. Eventualmente, cada transición puede dispararse de nuevo. Si una transición sólo puede dispararse una vez y no hay forma de volver a activarla, entonces la red no es viva (*live*).

Esto equivale a decir que la red de Petri está *libre de bloqueo* (*deadlock-free*). Definamos ahora lo que constituye un bloqueo/deadlock y mostremos ejemplos de ello.

Definition 15: Bloqueo en redes de Petri

Un bloqueo o deadlock en una red Patri es una transición (o un conjunto de transiciones) que no puede dispararse para ninguna marca alcanzable desde M_0. La transición (o un conjunto de transiciones) no puede volver a activarse después de un cierto punto de la ejecución.

Una transición está *viva* si no está bloqueada. Si una transición está viva, siempre es posible elegir una serie de disparos de transiciones adecuada para pasar del marcado actual a un marcado que habilite la transición.

Las redes de las Fig. 1.4, 1.5 and 1.6 están todas vivas. En todos estos casos, después de algunos disparos, la red vuelve al estado inicial y puede reiniciar el ciclo.

La red de la Fig. 1.1 no está viva. Después de dos disparos termina de ejecutarse y no puede ocurrir nada más. La red de la Fig. 1.3 tampoco está viva, porque T1 sólo se ejecutará una vez y a partir de ese momento sólo se podrá activar T2.

1.1.7. Análisis de alcanzabilidad

Tras haber introducido el conjunto de alcanzabilidad $R(N, M_0)$ en la Sec. 1.1.6, ahora podemos presentar una técnica de análisis importante para las redes de Petri: *el árbol de alcanzabilidad* (*reachability tree*).

Ejecutaremos paso a paso el algoritmo para construir el árbol de alcanzabilidad y, a continuación, presentaremos sus ventajas e inconvenientes. En términos generales, el árbol de alcanzabilidad tiene la siguiente estructura: Los nodos representan las marcas generadas a partir de M_0, la raíz del árbol y sus sucesores. Cada arco representa un disparo de transición que transforma un marcado en otro.

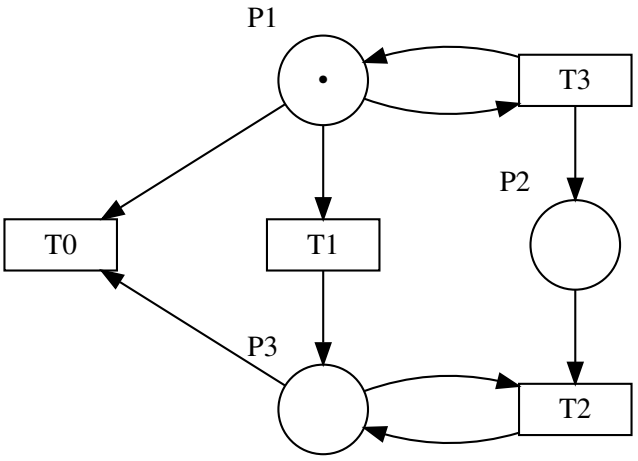

Figura 1.8: Una red de Petri marcada para ilustrar la construcción de un árbol de alcanzabilidad.

Considere la red de Petri mostrada en la Fig. 1.8. La marca inicial es $(1, 0, 0)$. En este marcado inicial se habilitan dos transiciones: T1 y T3. Dado que queremos obtener todo el conjunto de alcanzabilidad, definimos un nuevo nodo en el árbol de alcanzabilidad para cada marcado alcanzable, que resulta de disparar cada transición. Un arco, etiquetado por la transición disparada, conduce desde la marca inicial (la raíz del árbol) hasta cada una de las nuevas marcas. Tras este primer paso (Fig. 1.9), el árbol contiene todas las marcas que son inmediatamente alcanzables desde la marca inicial.

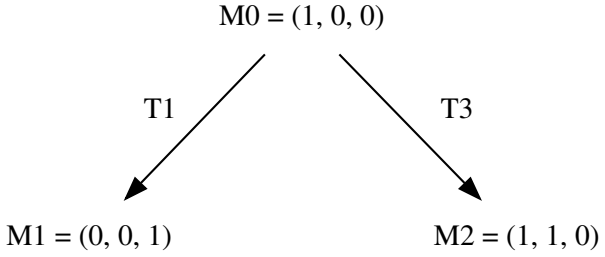

Figura 1.9: Primer paso para construir el árbol de alcanzabilidad de la red de Petri de la Fig. 1.8.

Ahora debemos considerar todas las marcas alcanzables desde las hojas del árbol.

A partir del marcado $(0, 0, 1)$ no podemos disparar ninguna transición. Esto se conoce como un *marcado muerto* (*dead marking*). En otras palabras se trata de un nodo "sin salida". Esta clase de estados finales es especialmente relevante para el análisis de bloqueos.

A partir de la marca de la derecha del árbol, denotada $(1, 1, 0)$, podemos disparar T1 o T3. Si disparamos T1, obtenemos $(0, 1, 1)$ y si dispara T3, la marca resultante es $(1, 2, 0)$. Esto produce el árbol de la Fig. 1.10.

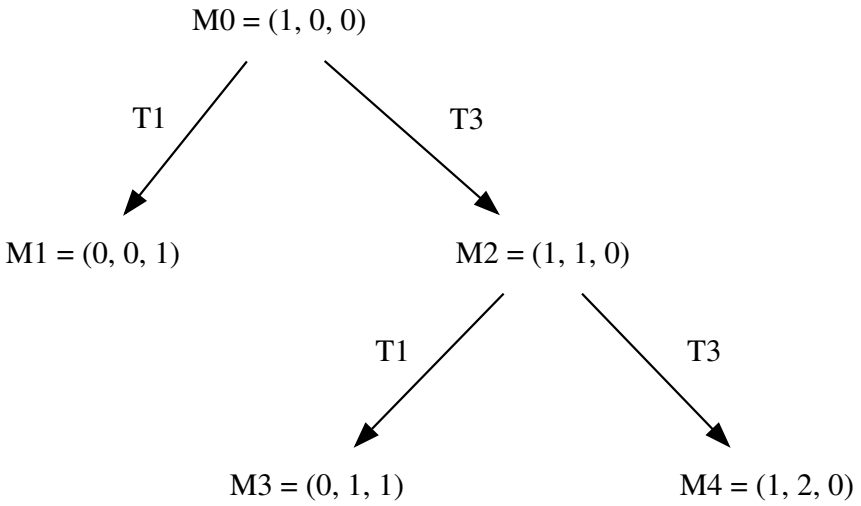

Figura 1.10: Segundo paso en la construcción del árbol de alcanzabilidad de la red de Petri de la Fig. 1.8.

Observe que partiendo de la marca $(0, 1, 1)$, sólo se habilita la transición T2 que conducirá a una marca $(0, 0, 1)$ ya vista anteriormente. Si en cambio tomamos $(1, 2, 0)$ tenemos de nuevo las mismas posibilidades que partiendo de $(1, 1, 0)$. Es fácil ver que el árbol seguirá creciendo por ese camino. Por tanto, el árbol es infinito y esto se debe a que la red de la Fig. 1.8 no está acotada. Veáse en la Fig. 1.11 el resultado final abreviado.

El método presentado anteriormente enumera los elementos del conjunto de alcanzabilidad. Se producirá cada marca del conjunto de alcanzabilidad y, por tanto, para cualquier red de Petri con un conjunto de alcanzabilidad infinito, es decir, un número infinito de estados posibles, el árbol correspondiente también sería infinito. Sin embargo, lo contrario no es cierto. Una red de Petri con un conjunto de alcanzabilidad finito puede tener un árbol infinito (véase la Fig. 1.12). Esta red es incluso *segura*. En conclusión, tratar con una red acotada o segura no es garantía de que el número total de estados alcanzables sea finito.

Para que el árbol de alcanzabilidad sea una herramienta de análisis útil, es necesario idear un método que lo limite a un tamaño finito. Esto implica en general una cierta pérdida de información, ya que el método tendrá que mapear (o mejor dicho reducir) un número infinito de marcados alcanzables a un solo elemento. La reducción a una representación finita puede lograrse por los siguientes medios.

Observe por un lado que podemos encontrarnos con nodos duplicados en nuestro árbol y que siempre los tratamos ingenuamente como nuevos. Esto se ilustra más claramente en la Fig. 1.12. Por tanto, es posible detener la exploración de los sucesores de un nodo duplicado.

Nótese, por otro lado, que algunas marcados son estrictamente diferentes de las marcas vistas anteriormente pero permiten el mismo conjunto de transiciones. Decimos en este caso que la

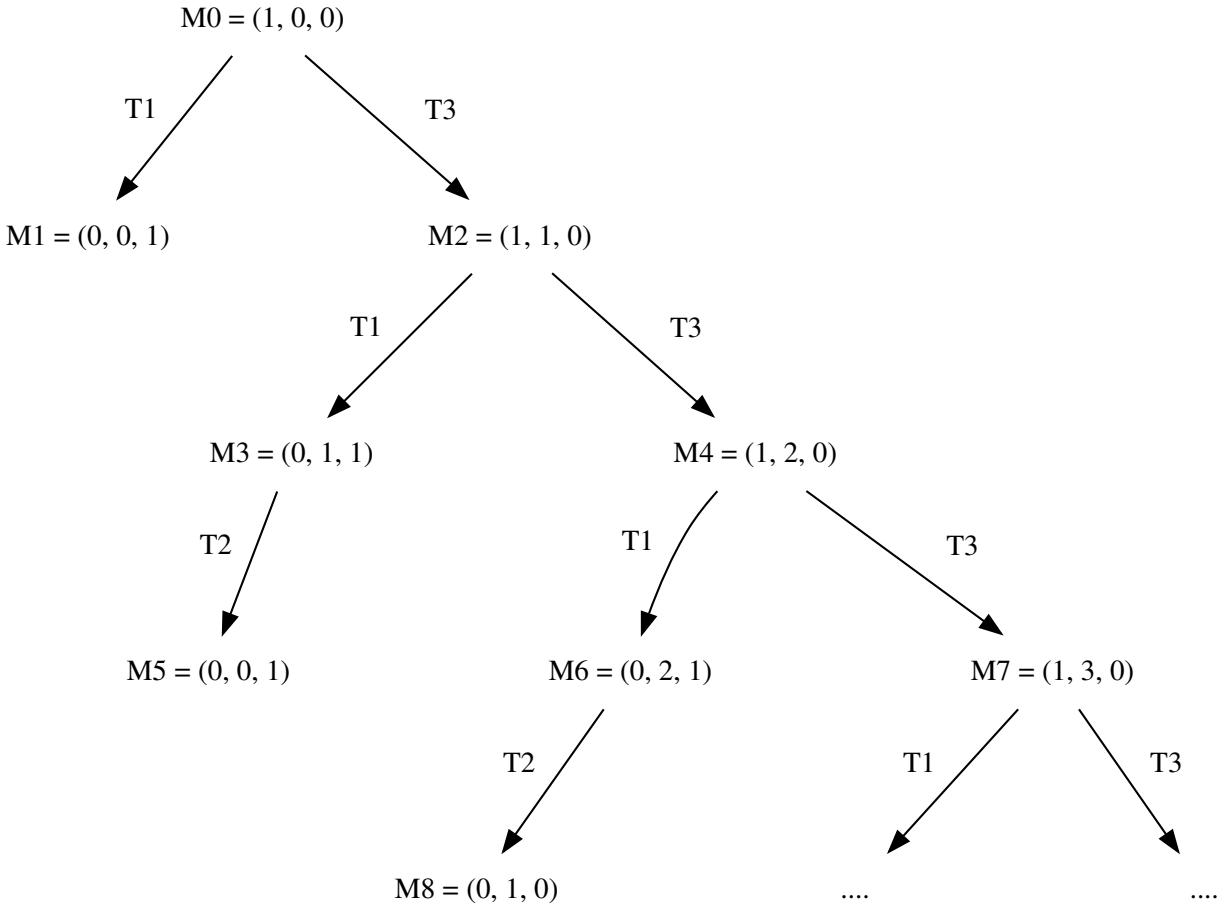

Figura 1.11: El árbol de alcanzabilidad infinita para la red de Petri de la Fig. 1.8.

marca con fichas adicionales *cubre* (*covers*) la que tiene el número mínimo de fichas necesarias para permitir el conjunto de transiciones en cuestión. Disparar algunas transiciones puede permitirnos acumular un número arbitrario de fichas en un lugar. Por ejemplo, disparar T3 en la red de Petri que se ve en la Fig. 1.8 muestra exactamente este comportamiento. En conclusión, bastaría con marcar el lugar de acumulación con una etiqueta especial ω, que significa infinito, ya que podríamos obtener tantas marcas como quisiéramos en ese lugar.

Por ejemplo, el resultado de convertir el árbol de la Fig. 1.11 en un árbol finito se muestra en la Fig. 1.13.

Para más detalles sobre

1. la técnica de representación de árboles de alcanzabilidad infinita mediante ω,
2. una definición del algoritmo y los pasos precisos para construir el árbol de alcanzabilidad,
3. la demostración matemática de que el árbol de alcanzabilidad generado por el algoritmo

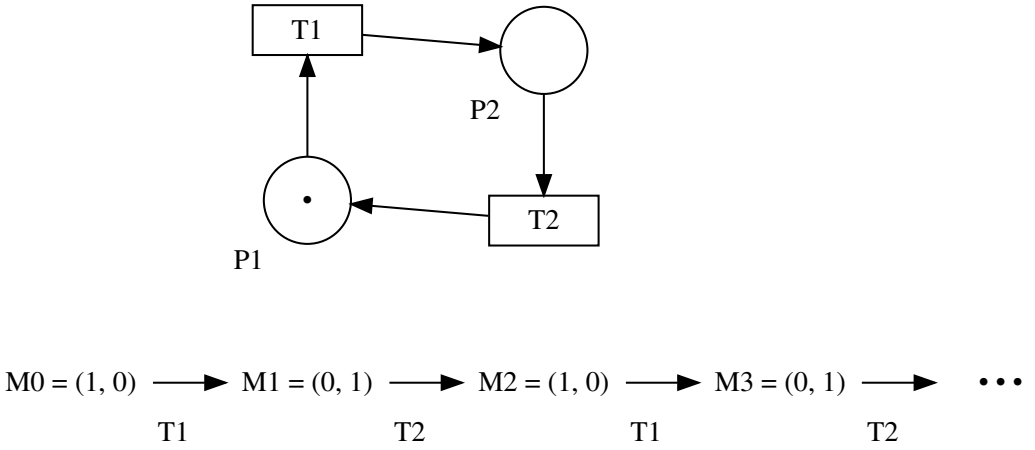

Figura 1.12: Una red de Petri simple con un árbol de alcanzabilidad infinito.

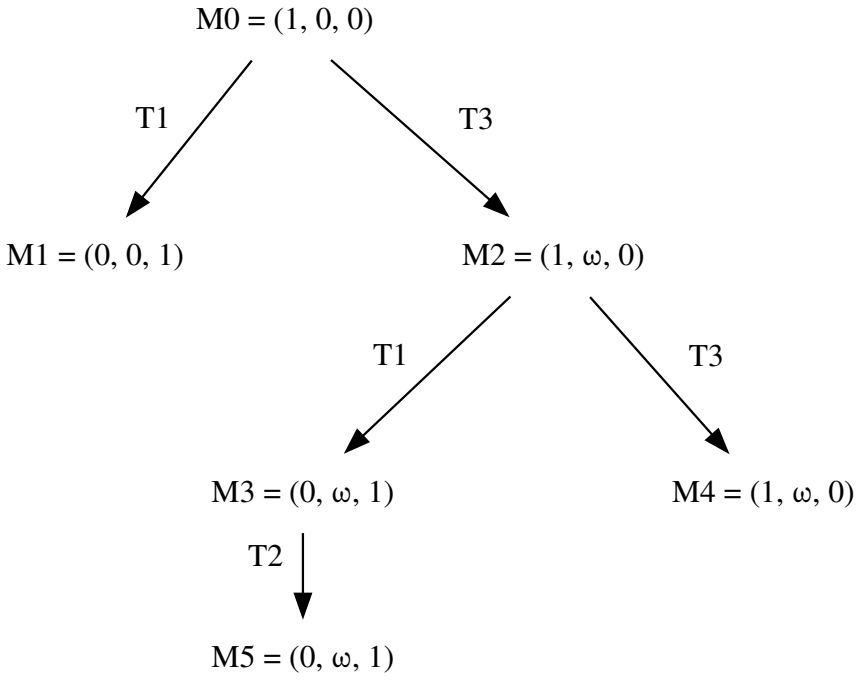

Figura 1.13: El árbol de alcanzabilidad finito para la red de Petri de la Fig. 1.8.

es finito,

4. y la distinción entre el árbol de alcanzabilidad y el *grafo de alcanzabilidad* (*reachability graph*)

se remite al lector a [Murata, 1989] y [Peterson, 1981]. Estos conceptos están fuera del alcance

de este trabajo y no son necesarios en los capítulos siguientes.

1.2. El lenguaje de programación Rust

Uno de los lenguajes de programación modernos más prometedores para la programación concurrente y segura para la memoria es Rust[1]. Rust es un lenguaje de programación multiparadigma y de propósito general cuyo objetivo es proporcionar a los desarrolladores una forma segura, concurrente y eficiente de escribir código de bajo nivel. Comenzó como un proyecto en Mozilla Research en 2009. La primera versión estable, Rust 1.0, se anunció el 15 de mayo de 2015. Para una breve historia de Rust hasta 2023, véase [Thompson, 2023].

El modelo de memoria de Rust basado en el concepto de propiedad (*ownership*) y su expresivo sistema de tipos evitan una gran variedad de clases de errores relacionados con la gestión de memoria y la programación concurrente en tiempo de compilación:

- 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] (con algunas advertencias importantes expuestas en [Rust Project, 2023c, Chap. 8.1])
- Pasar variables no seguras entre hilos [Klabnik and Nichols, 2023, Chap. 16.4]

El compilador oficial *rustc*[2] se encarga de controlar cómo se utiliza la memoria y de asignar y desasignar objetos. Si se encuentra una violación de sus reglas estrictas, el programa simplemente no será compilado.

En esta sección, justificaremos la elección de Rust para estudiar la detección de bloqueos y señales perdidas. Mostraremos cómo estos problemas pueden estudiarse por separado, sabiendo que otros errores ya se detectan en tiempo de compilación. En otras palabras, argumentaremos que la estabilidad y la seguridad del lenguaje proporcionan una base firme sobre la que construir una herramienta que detecte errores adicionales durante la compilación.

1.2.1. Características principales

Algunas de las principales características de Rust son:

- Sistema de tipos: Rust cuenta con un potente sistema de tipos que proporciona comprobaciones de seguridad en tiempo de compilación y evita muchos errores comunes de

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

programación. Incluye características como la inferencia de tipos, los tipos genéricos, los enums y el *pattern matching*. Cada variable tiene un tipo pero éste suele ser inferido por el compilador.

- Performance: La performance de Rust es comparable a la de C y C++ y a menudo es más rápido que muchos otros lenguajes de programación populares como Java, Go, Python o Javascript. La performance de Rust se debe a una combinación de características como las abstracciones de cero coste, un *runtime* mínimo y una gestión eficiente de la memoria.
- Concurrencia: Rust tiene un buen soporte por defecto para la concurrencia. Soporta varios paradigmas de concurrencia como el estado compartido, el pasaje de mensajes y la programación asíncrona. No obliga al desarrollador a implementar la concurrencia de una manera específica.
- *Ownership* y *borrowing*: Rust utiliza un modelo de *ownership* único para gestionar la memoria, lo que permite una asignación y desasignación de memoria eficientes sin riesgo de perder memoria o condiciones de carrera en el acceso a los datos. Además, no depende de un recolector de basura (*garbage collector*), ahorrando recursos. El verificador de préstamos (*borrow checker*) garantiza que sólo haya un propietario (*owner*) de un recurso en un momento dado.
- Impulsado por la comunidad: Rust cuenta con una vibrante y creciente comunidad de desarrolladores que contribuyen al desarrollo y al ecosistema del lenguaje. Cualquiera puede contribuir al lenguaje y sugerir mejoras. La documentación también es de código abierto y las decisiones importantes se documentan en forma de Requests for Comments (RFCs)[3].

El ciclo de publicación del compilador oficial de Rust, *rustc*, es notablemente rápido. Cada 6 semanas se publica una nueva versión estable del compilador [Klabnik and Nichols, 2023, Appendix G]. Esto es posible gracias a un complejo sistema de pruebas automatizado que compila incluso todos los paquetes disponibles en `crates.io`[4] utilizando un programa llamado *crater*[5] para verificar que la compilación y ejecución de las pruebas con la nueva versión del compilador no rompe los paquetes existentes [Albini, 2019].

El verificador de préstamos (*borrow checker*)

El verificador de préstamos (*borrow checker*) de Rust es un componente esencial de su modelo de *ownership*, diseñado para garantizar la seguridad de la memoria y evitar las carreras de datos (*data races*) en el código concurrente. El borrow checker analiza el código Rust en tiempo de compilación y aplica un conjunto de reglas para garantizar que se accede a la memoria de un programa de forma segura y eficiente.

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

La idea central detrás del borrow checker es que cada porción de memoria en un programa Rust tiene un propietario. El propietario puede cambiar durante la ejecución, pero sólo puede haber un propietario en un momento dado. Los valores de memoria también pueden tomarse *prestados* (*borrowed*), es decir, utilizarse sin cambiar el propietario, de forma similar al acceso al valor a través de un puntero o una referencia en otros lenguajes de programación. Cuando se toma prestado un valor, el prestatario recibe una referencia al valor, pero el propietario original conserva la propiedad. El verificador de préstamos aplica reglas para garantizar que un valor prestado no se modifica mientras está prestado y que el prestatario libera la referencia antes de que el propietario salga de *scope*.

Para mayor claridad, presentaremos a continuación algunas de las reglas centrales aplicadas por el borrow checker:

- No pueden existir simultáneamente dos referencias mutables a la misma posición de memoria. Esto evita las carreras de datos en las que dos hilos intentan modificar la misma posición de memoria al mismo tiempo.
- Las referencias mutables no pueden existir al mismo tiempo que las referencias inmutables a la misma ubicación de memoria. Esto garantiza que las referencias mutables e inmutables no puedan utilizarse simultáneamente, evitando lecturas y escrituras incoherentes.
- Las referencias no pueden sobrevivir al valor al que hacen referencia. Esto garantiza que las referencias no apunten a ubicaciones de memoria no válidas, evitando desferencias de punteros nulos y otros errores de memoria.
- Las referencias no pueden utilizarse después de que su propietario haya sido desplazado o destruido. Esto asegura que las referencias no apunten a memoria que ha sido desasignada, evitando errores de uso después de liberar.

Puede requerir cierto esfuerzo escribir código Rust que satisfaga estas reglas. El borrow checker suele señalarse como un aspecto del lenguaje que resulta confuso para los nuevos usuarios. Sin embargo, esta disciplina adquirida paga sus frutos en términos de mayor seguridad de memoria y performance durante la ejecución. Al garantizar que los programas Rust siguen estas reglas, el verificador de préstamos elimina muchos errores comunes de programación que pueden dar lugar a fugas de memoria, *data races* y otros bugs, al tiempo que enseña buenas prácticas y patrones de programación.

Manejo de errores aplicado por el compilador

La gestión de errores es un aspecto esencial de la programación y suele abordarse en el diseño de los lenguajes de programación. La miríada de enfoques puede resumirse en dos grupos distintos.

Un grupo formado por lenguajes como C++, Java o Python emplea excepciones, utilizando bloques `try` y `catch` para manejar condiciones excepcionales. Cuando se lanzan excepciones y no se capturan, el programa termina abruptamente.

El otro grupo lo forman lenguajes como C o Go, entre otros, en los que la convención es comunicar un error a través del valor de retorno de las funciones o mediante un parámetro de función específicamente dedicado a este fin. La desventaja es que el compilador no impone al programador la comprobación de errores, lo que puede hacer que no se tengan en cuenta los casos de error al añadir nuevas funciones.

Rust adopta un enfoque diferente promoviendo la noción de que las funciones idealmente no deberían fallar y que la firma de la función debería reflejar si la función puede devolver un error. En lugar de excepciones o códigos de error con números enteros, las funciones Rust que pueden terminar con errores devuelven un tipo `std::result::Result`[6] que puede contener el resultado del cálculo o un tipo de error personalizado acompañado de una descripción del error. *rustc* impone que el programador escriba código para ambos casos y el lenguaje proporciona mecanismos para facilitar la gestión de errores [Klabnik and Nichols, 2023, Chap. 9.2].

En Rust, el foco está puesto en la gestión coherente del caso de error. Los errores pueden propagarse a las llamadas a funciones de nivel superior hasta que pueda restablecerse un estado coherente del programa. Sin embargo, puede haber situaciones en las que la recuperación de un estado de error no sea factible. En tales casos, se puede ordenar al programa que entre en pánico, lo que resulta en un cierre abrupto y sin gracia (*ungraceful*), similar a una excepción no capturada en otros lenguajes de programación. Durante un pánico, la ejecución del programa se aborta y la pila se despliega (*stack unwinding*) [Klabnik and Nichols, 2023, Chap. 9.1]. Se genera un mensaje de error que contiene detalles del pánico, por ejemplo, el propio mensaje de error y su ubicación. Aunque los *panic* pueden ser capturados por hilos padre (*parent threads*) y en casos específicos cuando el programador así lo desea[7], normalmente conducen a la terminación del programa actual. Este mecanismo de pánico estructurado hace que el compilador sea consciente de los posibles errores irrecuperables, lo que permite la generación del código adecuado para manejar estos casos.

Rust también proporciona un tipo `std::option::Option`[8] que representa tanto la presencia de un valor como su ausencia. De nuevo, el compilador impone disciplina al programador para manejar siempre el caso `None`. De este modo, Rust elimina casi por completo la necesidad de un puntero NULL como se encuentra en otros lenguajes como C, C++, Java, Python o Go.

1.2.2. Adopción

En esta subsección, describiremos brevemente la tendencia en la adopción del lenguaje de programación Rust. Esto resalta la relevancia de este trabajo como contribución a una comunidad creciente de programadores que hacen hincapié en la importancia de una programación de sistemas segura y eficaz para los próximos años en la industria del software.

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

En los últimos años, varios proyectos importantes de la comunidad de código abierto y de empresas privadas han decidido incorporar Rust para reducir el número de bugs relacionados con la gestión de la memoria sin sacrificar performance. Entre ellos, podemos citar algunos ejemplos representativos:

- El Android Open Source Project fomenta el uso de Rust para los componentes del SO por debajo del Android Runtime (ART) [Stoep and Hines, 2021].
- El kernel Linux introduce en la versión 6.1 (publicada en diciembre de 2022) soporte oficial de herramientas para la programación de componentes en Rust [Corbet, 2022, Simone, 2022].
- En Mozilla, el proyecto Oxidation se creó en 2015 para aumentar el uso de Rust en Firefox y proyectos relacionados. En marzo de 2023, las líneas de código en Rust representan más del 10 % del total en Firefox Nightly [Mozilla Wiki, 2015].
- En Meta, el uso de Rust como lenguaje de desarrollo del lado del servidor está aprobado y es alentado desde julio de 2022 [Garcia, 2022].
- En Cloudflare, se construyó desde cero un nuevo proxy HTTP en Rust para superar las limitaciones architectónicas de NGINX, reduciendo el uso de CPU en un 70 % y el de memoria en un 67 % [Wu and Hauck, 2022].
- En Discord, la reimplementación en Rust de un servicio crucial escrito en Go proporcionó grandes beneficios en el rendimiento y resolvió una pérdida de rendimiento debida al *garbage collection* en Go [Howarth, 2020].
- En npm Inc, la empresa detrás del npm registry, Rust permitió escalar los servicios limitados por la cantidad de CPU disponible a más de 1.300 millones de descargas al día [The Rust Project Developers, 2019].

En otros casos, Rust ha demostrado ser una gran elección en proyectos C/C++ existentes para reescribir módulos que procesan entradas de usuario no fiables, por ejemplo, analizadores sintácticos, y reducir el número de vulnerabilidades de seguridad debidas a problemas de memoria [Chifflier and Couprie, 2017].

Además, el interés de la comunidad de desarrolladores por Rust es innegable, ya que ha sido calificado durante 7 años consecutivos como el lenguaje de programación más "querido"(*loved*) por los programadores en la encuesta Stack Overflow Developer Survey [Stack Overflow, 2022].

1.2.3. Importancia del uso seguro de la memoria

En esta subsección, se presentan pruebas convincentes que apoyan el uso de un lenguaje de programación que soporte un uso seguro de la memoria. El objetivo es resaltar la importancia de avanzar en la investigación sobre la detección de errores en tiempo de compilación para evitar fallos que posteriormente son difíciles de corregir en los sistemas en producción.

Varias investigaciones empíricas han concluido que alrededor del 70 % de las vulnerabilidades encontradas en grandes proyectos C/C++ se deben a errores en el manejo de la memoria. Esta cifra elevada puede observarse en proyectos como:

- Android Open Source Project [Stepanov, 2020],
- los componentes Bluetooth y multimedia de Android [Stoep and Zhang, 2020],
- el Proyecto Chromium detrás del navegador web Chrome [The Chromium Projects, 2015],
- el componente CSS de Firefox [Hosfelt, 2019],
- iOS y macOS [Kehrer, 2019],
- productos de Microsoft [Miller, 2019, Fernandez, 2019],
- Ubuntu [Gaynor, 2020]

Numerosas herramientas se han fijado el objetivo de abordar estas vulnerabilidades causadas por una asignación inadecuada de memoria en bases de código ya establecidas. Sin embargo, su uso conlleva una pérdida notable de rendimiento y no todas las vulnerabilidades pueden evitarse [Szekeres et al., 2013]. Un ejemplo de herramienta representativa en este ámbito, más concretamente un detector dinámico de data races para programas multihilo en C, puede encontrarse en [Savage et al., 1997], cuyo algoritmo se mejoró posteriormente en [Jannesari et al., 2009] y se integró en la herramienta Helgrind, parte del conocido framework de instrumentación Valgrind[9].

En [Jaeger and Levillain, 2014], los autores ofrecen un estudio detallado de las características de los lenguajes de programación que comprometen la seguridad de los programas resultantes. Hablan de las características de seguridad intrínsecas de los lenguajes de programación y enumeran recomendaciones para la formación de desarrolladores o evaluadores de software seguro. La seguridad de tipos se menciona como uno de los elementos clave para eliminar clases completas de errores desde el principio. Otra consideración digna de mención es utilizar un lenguaje en el que las especificaciones sean lo más completas, explícitas y formalmente definidas posible. El concepto de Undefined Behavior (UB) debe incluirse con precaución y sólo con moderación. Algunos ejemplos de la especificación C/C++ ilustran la confusión que se deriva de no seguir estos principios. Los autores concluyen que la seguridad de la memoria conseguida mediante la recogida de basura (*garbage collection*) supone una amenaza para la seguridad y que en su lugar deberían considerarse otros mecanismos.

Debemos tener en cuenta que el propio Rust, como cualquier otra producto de software, no está exento de vulnerabilidades de seguridad. En el pasado se han descubierto bugs serios en la biblioteca estándar [Davidoff, 2018]. Además, la generación de código en Rust también incluye mitigaciones a exploits de diversa índole [Rust Project, 2023b, Chap. 11]. Sin embargo, esto dista mucho de los problemas ampliamente conocidos en C y C++.

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

1.3. Correctitud de programas concurrentes

En el área de la computación concurrente, uno de los principales retos es demostrar la correctitud de un programa concurrente. A diferencia de un programa secuencial en el que para cada entrada se obtiene siempre la misma salida, en un programa concurrente la salida puede depender de cómo se hayan intercalado las instrucciones de los distintos procesos o hilos durante la ejecución.

La correctitud de un programa concurrente se define entonces en términos de las propiedades de la computación realizada y no sólo en términos del resultado obtenido. En la literatura [Ben-Ari, 2006, Coulouris et al., 2012, van Steen and Tanenbaum, 2017], se definen dos tipos de propiedades de correctitud:

- **Propiedades de seguridad (*Safety properties*):**: La propiedad debe ser *siempre* verdadera.
- **Propiedades de liveness**: La propiedad debe volverse *eventualmente* verdadera.

Dos propiedades de seguridad deseables en un programa concurrente son:

- **Exclusión mutua**: Dos procesos no deben acceder a los recursos compartidos al mismo tiempo.
- **Ausencia de bloqueo (*deadlock*)**: Un sistema en funcionamiento debe poder seguir realizando su tarea, es decir, progresando y produciendo trabajo útil.

Las primitivas de sincronización como los mutexes, los monitores (propuestos por [Hansen, 1972, Hansen, 1973]), los semáforos (propuestos por [Dijkstra, 2002]) y las variables de condición (propuestas por [Hoare, 1974]) suelen utilizarse para implementar el acceso coordinado de hilos o procesos a recursos compartidos. Sin embargo, el uso correcto de estas primitivas es difícil de conseguir en la práctica y puede introducir errores difíciles de detectar y corregir. Actualmente, la mayoría de los lenguajes de propósito general, ya sean compilados o interpretados, no permiten detectar estos errores en todos los casos.

Dada la creciente importancia de la programación concurrente debido a la proliferación de sistemas de hardware multihilo y multihilo, minimizar la aparición de errores asociados a la sincronización de hilos o procesos tiene una importancia innegable para la industria. El funcionamiento libre de deadlocks es un requisito fundamental para muchos proyectos, como los sistemas operativos [Arpaci-Dusseau and Arpaci-Dusseau, 2018], las aeronaves [Carreño and Muñoz, 2005, Monzon and Fernandez-Sanchez, 2009] y los vehículos autónomos [Perronnet et al., 2019].

En la próxima sección examinaremos más detenidamente las condiciones que provocan un deadlock y las estrategias utilizadas para hacerles frente.

1.4. Bloqueo mutuo (*deadlocks*)

Los bloqueos o *deadlocks* son un problema común en sistemas concurrentes, es decir, sistemas en los que varios hilos o procesos se ejecutan simultáneamente y potencialmente comparten recursos. Se han estudiado al menos desde [Dijkstra, 1964] quien acuñó el término "abrazo mortal" en holandés el cual cayó eventualmente en desuso.

Un deadlock se produce cuando dos o más hilos o procesos están bloqueados y no pueden seguir ejecutándose porque cada uno está esperando a que el otro libere un recurso que necesita. Esto da lugar a una situación en la que ninguno de los hilos o procesos puede progresar y el sistema queda efectivamente atascado. En [Holt, 1972] puede encontrarse una definición alternativa equivalente de los bloqueos en términos de los estados del programa.

Los deadlocks pueden ser un problema grave en los sistemas concurrentes, ya que pueden provocar que el sistema deje de responder o incluso que se interrumpa la ejecución abruptamente. Por lo tanto, sería ventajoso poder detectar y prevenir los dealocks. Pueden producirse en cualquier sistema concurrente en el que varios hilos o procesos compiten por recursos compartidos. Algunos ejemplos de recursos compartidos que pueden provocar deadlocks son la memoria del sistema, los dispositivos de entrada/salida, los *locks* y otros tipos de primitivas de sincronización.

Los bloqueos pueden ser difíciles de detectar y prevenir porque dependen de la sincronización precisa de los eventos en el sistema. Incluso en los casos en los que pueden detectarse los deadlocks, resolverlos puede ser difícil, ya que puede requerir liberar recursos que ya han sido adquiridos o deshacer transacciones completadas. Para evitar los bloqueos, es importante gestionar cuidadosamente los recursos compartidos en un sistema concurrente. Esto puede implicar el uso de técnicas como algoritmos de asignación de recursos, algoritmos de detección de bloqueos y otros tipos de primitivas de sincronización. Gestionando cuidadosamente los recursos compartidos es posible evitar que se produzcan bloqueos y garantizar el buen funcionamiento de los sistemas concurrentes.

Para entender el concepto con más detalle, considérese un ejemplo sencillo en el que dos procesos, A y B, compiten por dos recursos, X e Y. Inicialmente, el proceso A ha adquirido el recurso X y está esperando adquirir el recurso Y, mientras que el proceso B ha adquirido el recurso Y y está esperando adquirir el recurso X. En esta situación, ninguno de los dos procesos puede seguir ejecutándose porque está esperando a que el otro proceso libere un recurso que necesita. Esto da lugar a un deadlock, ya que ninguno de los dos procesos puede progresar. La Fig. 1.14 ilustra esta situación. El ciclo que aparece en ella indica un deadlock como se explicará en la siguiente sección.

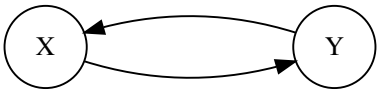

Figura 1.14: Ejemplo de un grafo de estados con un ciclo que indica un bloqueo mutuo.

1.4.1. Condiciones necesarias

Según el paper clásico sobre el tema [Coffman et al., 1971], deben darse las siguientes condiciones para que se produzca un deadlock. A veces se las denominan "condiciones de Coffman".

1. **Exclusión mutua (*Mutual Exclusion*)**: Al menos un recurso del sistema debe mantenerse en un modo no compartible, lo que significa que sólo un hilo o proceso puede utilizarlo a la vez, por ejemplo, una variable detrás de un mutex.
2. **Retener y esperar (*Hold and Wait*)**: Al menos un hilo o proceso del sistema debe estar reteniendo un recurso y esperando para adquirir recursos adicionales que en ese momento están siendo retenidos por otros hilos o procesos.
3. **Sin apropiación (*No Preemption*)**: Los recursos no pueden apropiarse, lo que significa que un hilo o proceso que posea un recurso no puede ser obligado a liberarlo hasta que haya completado su tarea.
4. **Espera circular (*Circular Wait*)**: Debe haber una cadena circular de dos o más hilos o procesos, en la que cada hilo o proceso esté esperando un recurso en poder del siguiente de la cadena. Esto suele visualizarse en un gráfico que representa el orden en que se adquieren los recursos.

Usualmente, las tres primeras condiciones son características del sistema estudiado, es decir, los protocolos utilizados para adquirir y liberar recursos, mientras que la cuarta puede materializarse o no en función del intercalado de instrucciones durante la ejecución.

Cabe señalar que las condiciones de Coffman son en general necesarias pero no suficientes para que se manifieste un bloqueo. En efecto, las condiciones son suficientes en el caso de sistemas de recursos de una sola instancia (una unidad de cada recurso). Pero sólo indican la posibilidad de un deadlock en los sistemas en los que hay múltiples instancias indistinguibles del mismo recurso.

En el caso general, si no se cumple alguna de las condiciones, no puede producirse un bloqueo, pero la presencia de las cuatro condiciones no garantiza necesariamente un bloqueo. No obstante, las condiciones de Coffman son un marco útil para comprender y analizar las causas de los bloqueos en los sistemas concurrentes y pueden ayudar a orientar el desarrollo de estrategias para prevenir y resolver los bloqueos.

1.4.2. Estrategias

Existen varias estrategias para gestionar los bloqueos, cada una de las cuales tiene sus puntos fuertes y débiles. En la práctica, la estrategia más eficaz dependerá de los requisitos y limitaciones específicos del sistema que se esté desarrollando. Los diseñadores y desarrolladores deben considerar cuidadosamente las compensaciones entre las distintas estrategias y

elegir el enfoque que mejor se adapte a sus necesidades. Se remite a los lectores interesados a [Coffman et al., 1971, Singhal, 1989].

Prevención

Una forma de hacer frente a los bloqueos es evitar que se produzcan en primer lugar. La idea es que los bloqueos se excluyan a priori. Con este objetivo en mente, debemos asegurarnos de que en cada momento no se cumple al menos una de las condiciones necesarias desarrolladas en la Sec. 1.4.1. Esto restringe los posibles protocolos en los que se pueden realizar solicitudes de recursos. A continuación examinaremos cada condición por separado y desarrollaremos los enfoques más comunes.

Si la primera condición debe ser falsa, entonces el programa debe permitir el acceso compartido a todos los recursos. Los algoritmos de sincronización sin bloqueo pueden utilizarse para este fin, ya que no implementan la exclusión mutua. Esto es difícil de conseguir en la práctica para todos los tipos de recursos, ya que, por ejemplo, un archivo no puede ser compartido por más de un hilo o proceso durante una actualización del contenido del archivo.

En cuanto a la segunda condición, un enfoque viable sería imponer que cada hilo o proceso adquiera todos los recursos necesarios a la vez y que el hilo o proceso no pueda continuar hasta que se le haya concedido acceso a todos ellos. Esta política de "todo o nada" provoca una penalización significativa del rendimiento, dado que los recursos pueden asignarse a un hilo o proceso específico pero pueden permanecer sin utilizar durante largos periodos. En términos más sencillos, disminuye la concurrencia.

Si se deniega la condición de no apropiación, los recursos pueden recuperarse en determinadas circunstancias, por ejemplo, utilizando algoritmos de asignación de recursos que garanticen que los recursos nunca se retienen indefinidamente. Tras un tiempo de espera o cuando se cumple una condición, el hilo o proceso libera el recurso o un proceso supervisor lo recupera a la fuerza. Normalmente, esto funciona bien cuando el estado del recurso puede guardarse fácilmente y restaurarse más tarde. Un ejemplo de ello es la asignación de núcleos de CPU en un sistema operativo (OS) moderno. El *scheduler* asigna un núcleo de procesador a una tarea y puede cambiar a una tarea diferente o puede mover la tarea a un nuevo núcleo de procesador en cualquier momento simplemente guardando el contenido de los registros [Arpaci-Dusseau and Arpaci-Dusseau, 2018, Chap. 6]. Sin embargo, si no es posible preservar el estado de los recursos, el adelantamiento puede implicar una pérdida del progreso realizado hasta el momento, lo que no es aceptable en muchos escenarios.

Por último, si el grafo de estados de los recursos nunca forma un ciclo, entonces la cuarta condición necesaria es falsa y se evitan los bloqueos. Para lograrlo, se podría introducir una ordenación lineal de los tipos de recursos. En otras palabras, si a un proceso o hilo se le han asignado recursos del tipo r_i, podrá requerir posteriormente sólo aquellos recursos de tipos que sigan a r_i en el ordenamiento. Esto implica utilizar primitivas de sincronización especiales que permitan compartir recursos de forma controlada y aplicar reglas estrictas para la adquisición

y liberación de recursos. En estas condiciones, el grafo de estado será estrictamente un bosque (un grafo acíclico), por lo que no es posible que se produzcan bloqueos.

En aplicaciones prácticas, una combinación de las estrategias anteriores puede resultar útil cuando ninguna de ellas sea totalmente aplicable.

Evasión (*avoidance*)

La evitación es otra estrategia para hacer frente a los bloqueos, que consiste en detectar y evitar dinámicamente los bloqueos potenciales *antes* de que se produzcan. Para ello, el sistema requiere un conocimiento global por adelantado sobre qué recursos solicitará un hilo o proceso durante su vida. Tenga en cuenta que, en términos lingüísticos, "evitar un bloqueo" y "prevenir un bloqueo" pueden parecer similares, pero en el contexto de la gestión de bloqueos, son conceptos distintos.

Uno de los algoritmos clásicos para evitar el bloqueo es el algoritmo de Banker [Dijkstra, 1964]. Otro algoritmo relevante es el propuesto por [Habermann, 1969].

Lamentablemente, estas técnicas sólo son efectivas en escenarios muy específicos, como en un sistema embebido en el que se conoce a priori el conjunto completo de tareas a ejecutar y el *locks* necesarios. En consecuencia, la evitación de bloqueos no es una solución de uso común aplicable a una amplia gama de situaciones.

Detección y recuperación

Otra estrategia para gestionar los bloqueos es detectarlos *después* de que se produzcan y recuperarse de ellos. Para un estudio de los algoritmos de detección de bloqueos en sistemas distribuidos, véase [Singhal, 1989]. Presentaremos brevemente la idea general que subyace a uno de ellos con fines ilustrativos.

El gráfico de asignación de recursos (Resource Allocation Graph (RAG)) es un método comúnmente utilizado para detectar bloqueos en sistemas concurrentes. Representa la relación entre hilos/procesos y recursos en el sistema como un grafo dirigido. Cada proceso y recurso está representado por un nodo en el grafo y se traza una arista dirigida desde un proceso a un recurso si el proceso está actualmente ocupando ese recurso. Esto es análogo al grafo de estado mostrado en la Fig. 1.14 pero con los hilos/procesos representados en el diagrama. El grafo de estado también puede aplicarse a la detección de bloqueos [Coffman et al., 1971].

Para detectar bloqueos mediante el RAG tenemos que buscar ciclos en el gráfico. Si hay un ciclo en el gráfico, indica que un conjunto de procesos está esperando recursos que en ese momento están en manos de otros procesos del ciclo. Por lo tanto, ningún proceso del ciclo puede avanzar.

La parte de recuperación del proceso consiste en terminar uno de los hilos o procesos del ciclo. Esto hace que se liberen los recursos y que los demás hilos o procesos puedan continuar.

Los sistemas de gestión de bases de datos (Database management systems (DBMS)) incorporan subsistemas para detectar y resolver los bloqueos. Un detector de bloqueos se ejecuta a intervalos, generando un gráfico de asignación regular, también llamado gráfico de transacción-espera (transaction-wait-for (TWF)), y examinándolo en busca de cualquier ciclo. Si se identifica un ciclo (deadlock), el sistema debe reiniciarse. Una excelente visión general de la detección de bloqueos en sistemas de bases de datos distribuidos es [Knapp, 1987]. El tema del control de la concurrencia y la recuperación de los bloqueos en los DBMS se trata ampliamente en [Bernstein et al., 1987].

Aceptar o ignorar por completo los deadlocks

En algunos casos, puede ser admisible aceptar simplemente el riesgo de que se produzcan deadlocks y gestionarlos a medida que vayan apareciendo. Este enfoque puede ser adecuado en sistemas en los que el coste de prevenir o detectar los deadlocks sea demasiado elevado, o en los que la frecuencia de los deadlocks sea lo suficientemente baja como para que el impacto en el rendimiento del sistema sea mínimo, o en los que la pérdida de datos que se produzca cada vez sea tolerable.

UNIX es un ejemplo de sistema operativo que sigue este principio [Shibu, 2016, p. 477]. Otros sistemas operativos populares también muestran este comportamiento. Por otro lado, un sistema de misión crítica no puede permitirse fingir que su funcionamiento estará libre de bloqueos por ningún motivo.

1.5. Condition variables

Las variables de condición (*condition variables*) son una primitiva de sincronización en la programación concurrente que permite a los hilos esperar eficientemente a que se cumpla una condición específica antes de continuar. Fueron introducidas por primera vez por [Hoare, 1974] como parte de un bloque de construcción para el concepto de monitor desarrollado originalmente por [Hansen, 1973].

Siguiendo la definición clásica, se pueden llamar dos operaciones principales sobre una variable de condición:

- `esperar (wait)`: Bloquea el hilo o proceso actual. En algunas implementaciones, el mutex asociado se libera como parte de la operación.
- `señal (signal)`: Despierta un hilo o proceso que espera en la condition variable. En algunas implementaciones, el mutex asociado es adquirido inmediatamente por el hilo o proceso sobre el que se aplica la operación.

Las condition variables suelen estar asociadas a un predicado booleano (una condición) y a un mutex. El predicado booleano es la condición que esperan los hilos o procesos. Cuando

se establece en un valor determinado (verdadero o falso), el hilo o proceso puede continuar ejecutándose. El mutex garantiza que sólo un hilo o proceso pueda acceder a la condition variable a la vez.

Las variables de condición no contienen un valor real accesible para el programador en su interior. En su lugar, se implementan utilizando una estructura de datos de cola donde los hilos o procesos se añaden a la cola cuando entran en el estado de espera. Cuando otro hilo o proceso señala el estado, se selecciona un elemento de la cola para reanudar la ejecución. La política de *scheduling* específica puede variar en función de la implementación.

A lo largo de los años, se han desarrollado diversas implementaciones y optimizaciones para las condition variables con el fin de mejorar el rendimiento y reducir la sobrecarga. Por ejemplo, algunas implementaciones permiten despertar varios hilos a la vez (una operación denominada *broadcast*), mientras que otras utilizan una cola de prioridad para lograr que los hilos de mayor prioridad se despierten primero.

Las condition variables forman parte de la biblioteca estándar POSIX para hilos (`pthreads`) [Nichols et al., 1996] y en la actualidad se utilizan ampliamente en lenguajes y sistemas de programación concurrentes. Se encuentran entre otros en:

- UNIX[10],
- Rust[11]
- Python[12]
- Go[13]
- Java[14]

A pesar de su uso generalizado, las variables de condición pueden ser difíciles de utilizar correctamente, y un uso incorrecto puede dar lugar a errores sutiles y difíciles de depurar, como señales perdidas o despertares espurios. A continuación veremos estos errores en detalle.

1.5.1. Señales perdidas

Una señal perdida ocurre cuando un hilo o proceso que espera en una condition variable no recibe una señal aunque haya sido emitida. Esto puede ocurrir debido a una condición de carrera en la que la señal se emite antes de que el hilo entre en estado de espera, provocando que se pierda la señal.

[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

Para ilustrar el concepto de señal perdida, veremos un ejemplo. Supongamos que tenemos dos hilos, T1 y T2, y una variable entera compartida llamada `flag`. T1 establece `flag` en `true` y envía una señal a una variable de condición `cv` para despertar a T2 que está esperando en cv para saber cuándo se ha marcado `flag`. T2 espera en `cv` hasta que recibe una señal de T1. El Listado 1.1 muestra el pseudocódigo correspondiente.

```
1  // T1
2  lock.acquire()
3  flag = true
4  cv.signal()   // Signal T2 to wake up
5  lock.release()
6
7  // T2
8  lock.acquire()
9  while (flag == false)   // Wait until flag has changed
10     cv.wait(lock)
11 lock.release()
```

Listing 1.1: Pseudocódigo para un ejemplo de señal perdida.

Supongamos ahora que T1 activa `flag` y emite una señal a `cv` pero T2 aún no ha entrado en estado de espera en `cv` debido a algún retraso en *scheduling*. En este caso, la señal emitida por T1 podría ser pasada por alto por T2, como se muestra en la siguiente secuencia de acontecimientos:

1. T1 adquiere el bloqueo y marca `flag` como `true`.
2. T1 indica a `cv` que despierte a T2.
3. T1 libera el lock.
4. T2 adquiere el bloqueo y comprueba si `flag` ha cambiado. Como `flag` sigue siendo `flag`, T2 entra en estado de espera en `cv`.
5. Debido a retrasos en el *scheduling* o a otros factores, T2 no recibe la señal emitida por T1 y permanece atascado en el estado de espera para siempre.

Este escenario ilustra el concepto de señal perdida, en el que un hilo que espera en una variable de condición no recibe una señal aunque haya sido emitida. Para evitar que se pierdan señales, es esencial asegurarse de que los hilos que esperan en variables de condición estén correctamente sincronizados con los hilos que emiten señales y de que no existan condiciones de carrera o problemas de sincronización que puedan hacer que se pierdan señales.

1.5.2. Despertares espurios (*spurious wakeups*)

Un despertar espurio (*spurious wakeup*) ocurre cuando un hilo que espera en una variable de condición se despierta sin recibir una señal o notificación de otro hilo. Las razones son múltiples: interrupciones del hardware o del sistema operativo, detalles internos de implementación de la condition variable u otros factores impredecibles.

Reutilizando la situación descrita en la sección anterior y el pseudocódigo mostrado en el Listado 1.1, supongamos ahora que T1 pone el `flag` em `true` y emite una señal a `cv`, pero T2 se despierta sin recibir la señal emitida por T1.

Este es precisamente el despertar espurio. La siguiente secuencia de acontecimientos conduce a este desafortunado resultado:

1. T1 adquiere el bloqueo y marca `flag` como `true`.
2. T1 indica a `cv` que despierte a T2.
3. T1 libera el lock.
4. T2 adquiere el lock y comprueba si `flag` es verdadero. Como `flag` sigue siendo `false`, T2 entra en estado de espera en cv.
5. Debido a algún detalle de implementación interna de la condition variable o a otros factores impredecibles, T2 se despierta sin recibir la señal emitida por T1 y continúa ejecutando la siguiente sentencia de su código.

Este ejemplo demuestra la idea de un despertar espurio en el que un hilo que espera en una condition variable se despierta sin recibir una señal o notificación de otro hilo. Para evitar los despertares espurios es inevitable utilizar un bucle para volver a comprobar la condición después de despertarse de un estado de espera, como se muestra en el pseudocódigo para T2 (línea 9). Esto garantiza que el hilo no prosiga hasta que la condición que está esperando se haya producido realmente. Si no existiera el bucle while, un despertar espurio haría que T2 continuara ejecutándose después de la llamada a `wait`, independientemente de si T1 emitió una señal o no.

1.6. Arquitectura del compilador

Los compiladores son programas que transforman el código fuente escrito en un lenguaje en otro lenguaje, normalmente código máquina. Un compilador toma un programa en un lenguaje, el *lenguaje fuente*, y lo traduce a un programa equivalente en otro lenguaje, el *lenguaje de destino*.

Para lograrlo, los compiladores suelen tener una serie de fases o pases que se ejecutan en secuencia. El objetivo de estos pases es traducir el código de alto nivel en código de bajo nivel que la máquina pueda ejecutar. Tras cada pasada, el código se acerca cada vez más a la

representación final. Estas fases están hoy en día bien definidas y diferentes compiladores las implementan en alguna forma u otra [Aho et al., 2014, Chap. 1.2].

La primera pasada de un compilador típico es la fase de **análisis léxico (*lexical analysis*)**. En esta fase, el código fuente se descompone en un flujo de tokens, cada uno de los cuales representa una única pieza del código. El analizador léxico (*lexer*) identifica palabras clave, identificadores, literales y otros tokens que forman los bloques de construcción del código fuente.

La siguiente pasada es la fase de **análisis sintáctico (*syntax analysis*)**, también conocida como fase del *parser*. En esta fase, los tokens producidos por el *lexer* se analizan según las reglas de la gramática del lenguaje de programación. El analizador sintáctico construye un *parse tree* o un árbol sintáctico abstracto (abstract syntax tree (AST)) que representa la estructura del código.

La tercera pasada es la fase de **análisis semántico (*semantic analysis*)**, en la que el compilador comprueba la correctitud semántica del código, como la comprobación de errores de tipo, variables no definidas y operaciones no válidas. El analizador semántico (*semantic analyzer*) construye una tabla de símbolos que contiene información sobre las variables, funciones y otras entidades definidas en el código.

La cuarta pasada es la fase de **generación de código (*code generation*)**. El compilador toma el AST y la tabla de símbolos producidos por las fases anteriores y genera código de bajo nivel que puede ser ejecutado por la máquina. El generador de código suele generar código en lenguaje ensamblador o código máquina. En otros casos, genera bytecode, como en Java o cuando se utiliza el compilador just-in-time (JIT) de Python.

Finalmente, puede haber cero o más fases de **optimización del código (*code optimization*)**. Estas son, desde un punto de vista teórico, opcionales, pero suelen incluirse por defecto en los compiladores modernos. En esta fase, el compilador analiza el código generado e intenta mejorar su eficiencia aplicando diversas técnicas de optimización. Algunos ejemplos de optimizaciones son:

- plegado de constantes (*constant folding*) [Aho et al., 2014, Chap. 8.5.4],
- desenrollado de bucles (*loop unrolling*) [Aho et al., 2014, Chap. 10.5],
- asignación de registros (*register allocation*) [Aho et al., 2014, Chap. 8.1.4],
- propagación de constantes (*constant propagation*) [Aho et al., 2014, Chap. 9],
- análisis de vida útil (*livenes analysis*) [Aho et al., 2014, Chap. 9],
- y muchos más...

Las optimizaciones *locales* de código se refieren a mejoras dentro de un bloque básico, mientras que la optimización *global* de código es cuando las mejoras tienen en cuenta lo que ocurre más allá de un bloque básico. En Rust, un ejemplo de optimización global es la optimización del tiempo de enlace (link time optimization (LTO)) [Huss, 2020].

La Fig. 1.15 tomada de [Aho et al., 2014] visualiza intuitivamente las fases del compilador descritas en esta sección.

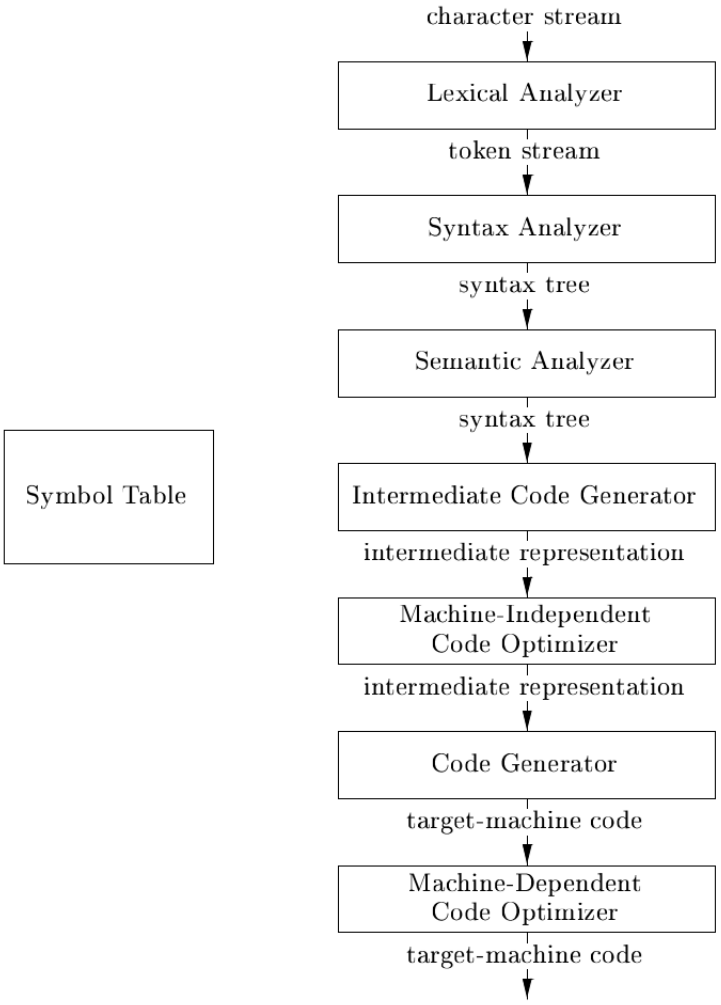

Figura 1.15: Fases de un compilador.

En la práctica, las fases pueden tener límites poco claros. Pueden solaparse y algunas pueden saltarse por completo. En secciones posteriores estudiaremos la arquitectura del compilador de Rust *rustc* y explicaremos su arquitectura en términos generales.

1.7. Verificación de modelos

La comprobación de modelos es una técnica utilizada en el desarrollo de software para verificar formalmente la correctitud del comportamiento de un sistema con respecto a sus especificaciones

o requisitos. Consiste en construir un modelo matemático del sistema y analizarlo para garantizar que cumple ciertas propiedades, como la exclusión mutua al acceder a recursos compartidos, la ausencia de carreras de datos (*data races*) y la ausencia de deadlocks.

El proceso de comprobación de modelos comienza construyendo un modelo de estado finito del sistema, típicamente utilizando un lenguaje formal, en el caso de este trabajo el lenguaje de las redes de Petri. El modelo captura el comportamiento del sistema y las propiedades que deben verificarse. El siguiente paso es realizar una búsqueda exhaustiva del espacio de estados del modelo para asegurarse de que se han considerado todos los comportamientos posibles. Esta búsqueda puede realizarse automáticamente utilizando herramientas de software especializadas.

Durante la búsqueda, el verificador de modelos busca contraejemplos, es decir secuencias de eventos que violen las especificaciones del sistema. Si se encuentra un contraejemplo, el verificador de modelos proporciona información sobre el estado del sistema en el momento de la violación, lo que ayuda a los desarrolladores a identificar y solucionar el problema.

La comprobación de modelos se ha convertido en una técnica ampliamente aplicada en el desarrollo de sistemas de software críticos, como los dispositivos médicos, los sistemas financieros y de control aeroespacial [Carreño and Muñoz, 2005, Monzon and Fernandez-Sanchez, 2009] y de automóviles [Perronnet et al., 2019]. Al verificar la correctitud del software antes de su despliegue, los desarrolladores pueden garantizar que el sistema cumple sus requisitos y es seguro de usar.

Una de las principales ventajas de la comprobación de modelos es que proporciona un enfoque formal y riguroso para verificar la correctitud del software. A diferencia de los métodos de prueba (test) tradicionales, que sólo pueden demostrar la presencia de errores, la comprobación de modelos puede demostrar la ausencia de errores. Esto es especialmente relevante para sistemas de seguridad crítica como los mencionados anteriormente en los que un solo error puede tener consecuencias catastróficas para vidas humanas. La comprobación de modelos también puede automatizarse, lo que permite a los desarrolladores verificar de forma rápida y eficaz la correctitud de sistemas de software complejos. Esto reduce el tiempo y el coste del desarrollo de software y aumenta la confianza en el sistema.

Se sabe que las herramientas formales de verificación de software se aplican actualmente en unos pocos campos muy específicos en los que se requiere una demostración formal de la correctitud del sistema. [Reid et al., 2020] habla de la importancia de acercar las herramientas de verificación a los desarrolladores mediante un enfoque que busque maximizar la relación coste- beneficio de su uso. Se presentan mejoras en la usabilidad de las herramientas existentes y enfoques para incorporar su uso a la rutina del desarrollador. El paper parte de la premisa de que, desde el punto de vista del desarrollador, la verificación puede verse como un tipo diferente de prueba unitaria o de integración. Por lo tanto, es de suma importancia que la ejecución de la verificación sea lo más sencilla posible y que se proporcione retroalimentación (feedback) al desarrollador con rapidez durante el proceso de desarrollo para aumentar su adopción.

La principal conclusión de esta sección es que la comprobación de modelos podría aportar

mejoras sustanciales en términos de mayor seguridad y fiabilidad de los sistemas de software. Estos objetivos se alinean con las metas del lenguaje de programación Rust y los objetivos de este trabajo. Detectar los bloqueos y las señales perdidas en el código fuente en tiempo de compilación podría ayudar a los desarrolladores a evitar errores difíciles de encontrar y a obtener feedback rápidamente sobre el uso correcto de las primitivas de sincronización, ahorrando tiempo y, además, dinero en el proceso de desarrollo. Un objetivo concreto de este trabajo es hacer que la herramienta sea fácil de usar y de empezar a utilizar, de modo que su adopción beneficie a la comunidad entera de desarrolladores de Rust.

Capítulo 2

Estado del arte

En este capítulo, se revisa brevemente la literatura sobre la verificación formal del código Rust y el modelado de redes de Petri para la detección de deadlocks. Algunas de estas publicaciones anteriores contienen enfoques que han guiado este trabajo.

En las dos secciones sucesivas examinaremos las herramientas existentes, su alcance y sus objetivos en comparación con la herramienta desarrollada en esta tesis.

A continuación, se ofrece un estudio de las bibliotecas de redes de Petri existentes en el ecosistema Rust a principios de 2023 para justificar la necesidad de implementar una biblioteca desde cero.

Posteriormente, exploramos la comunidad de investigadores detrás del Concurso de Verificación de Modelos (Model Checking Contest (MCC)) y los verificadores de modelos que participan en él para confirmar el potencial de estas herramientas para analizar modelos de redes de Petri de tamaño significativo. Esto es relevante ya que el verificador de modelos actúa como backend de la herramienta desarrollada en este trabajo.

Por último, se presentan tres de los formatos de archivo existentes para el intercambio de redes de Petri y se explica su finalidad en el contexto de este trabajo.

2.1. Verificación formal de código Rust

Existen numerosas herramientas de verificación automática disponibles para el código Rust. Una primera aproximación recomendable al tema es la encuesta elaborada por Alastair Reid, investigador de Intel. En ella se enumera explícitamente que la mayoría de las herramientas de verificación formal no soportan la concurrencia [Reid, 2021].

El intérprete *Miri*[1] desarrollado por el proyecto Rust en GitHub es un intérprete experimental

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

para la representación intermedia del lenguaje Rust ("Mid-level Intermediate Representation", comúnmente conocida como "MIR") que permite ejecutar binarios estándar de proyectos de *cargo* en una forma granularizada, instrucción por instrucción, para comprobar la ausencia de Comportamientos Indefinidos (Undefined Behavior (UB)) y otros errores en el manejo de la memoria. Detecta fugas de memoria, accesos a memoria no alineados, carreras de datos y violaciones de precondiciones o invariantes en el código marcado como inseguro (`unsafe`).

[Toman et al., 2015] presenta un verificador formal para Rust que no requiere modificaciones en el código fuente. Se probó en versiones anteriores de módulos de la biblioteca estándar de Rust. Como resultado se detectaron errores en el uso de la memoria en código Rust `unsafe` que, en circunstancias reales, el equipo de desarrollo tardó meses en descubrir manualmente. Esto ejemplifica la importancia de utilizar herramientas de verificación automática para complementar las revisiones manuales del código (*code reviews*).

[Kani Project, 2023] es otra herramienta popular para la verificación formal de código Rust destinada a comprobar los bloques inseguros (`unsafe`) a nivel de bits. Ofrece un framework de pruebas análogo al framework de pruebas proporcionado por Rust. Además, dispone de un plugin para *cargo* y VS Code.

Como explica la documentación del repositorio[2], Kani verifica (entre otros):

- Uso seguro de la memoria, por ejemplo, desreferencias de punteros nulos (*Memory safety, e.g., null pointer dereferences*)
- Aserciones especificadas por el usuario (*User-specified assertions, i.e.,* `assert!(...)`)
- La ausencia de panics, por ejemplo, `unwrap()` en valores `None`
- La ausencia de algunos tipos de comportamiento inesperado, por ejemplo, desbordamientos aritméticos (*arithmetic overflows*)

Sin embargo, los programas concurrentes están actualmente fuera de alcance[3]. La conclusión es que Kani ofrece una CLI fácil de usar y un framework de pruebas que se integran perfectamente en el proceso de desarrollo. Sirve como ilustración de las capacidades de la comprobación de modelos en el desarrollo de software moderno.

2.2. Detección de deadlocks mediante redes de Petri

La prevención de los bloqueos es una de las estrategias clásicas para abordar este problema fundamental en la programación concurrente, como se explica en la Sec. 1.4.2. El principal problema del enfoque de detectar los bloqueos antes de que se produzcan es probar que se detecta el tipo de bloqueo deseado en todos los casos y que no se producen falsos negativos en el proceso.

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

El enfoque basado en redes de Petri, al ser un método formal, satisface estas condiciones. Sin embargo, la dificultad de su adopción radica principalmente en la practicabilidad de la solución debido al gran número de estados posibles en un proyecto de software real.

En [Karatkevich and Grobelna, 2014], se propone un método para reducir el número de estados explorados durante la detección de bloqueos mediante el análisis de alcanzabilidad. Esta heurística ayuda a mejorar el rendimiento del enfoque basado en redes de Petri. Otra optimización se presenta en [Küngas, 2005]. El autor propone un método de orden polinómico muy prometedor para evitar el problema de la explosión de estados que subyace al algoritmo ingenuo de detección de deadlocks. A través de un algoritmo que abstrae una red de Petri dada a una representación más simple, se obtiene una jerarquía de redes de tamaño creciente para las que la verificación de la ausencia de bloqueos es sustancialmente más rápida. Se trata, dicho crudamente, de una estrategia de "divide y vencerás" que comprueba la ausencia de deadlocks en partes de la red para construir después la verificación del conjunto final añadiendo partes a la pequeña red inicial.

A pesar de las advertencias mencionadas anteriormente, el uso de las redes de Petri como método formal de verificación de software se ha establecido desde finales de la década de 1980. Las redes de Petri permiten un modelado intuitivo de las primitivas de sincronización, como el envío de un mensaje o la espera de la recepción de un mensaje. En [Heiner, 1992] encontrará ejemplos de estas sencillas redes con un comportamiento correspondientemente simple. Estas redes son bloques de construcción que pueden combinarse para formar un sistema más complejo.

Para poner en práctica estos modelos, existen dos posibilidades:

- Una es diseñar el sistema en términos de redes de Petri y luego traducir las redes de Petri al código fuente.
- La otra consiste en traducir el código fuente existente a una representación de red de Petri y, a continuación, verificar que el modelo de red de Petri satisface las propiedades deseadas.

A efectos de este trabajo, nos interesa esta última. Este enfoque no es novedoso. Ya se ha implementado para otros lenguajes de programación como C y Rust, como se ve en la bibliografía existente.

En [Kavi et al., 2002] y [Moshtaghi, 2001], se describe una traducción de algunas primitivas de sincronización disponibles como parte de la biblioteca POSIX de hilos (`pthread`) en C a redes de Petri. En concreto, la traducción admite:

- La creación de hilos con la función `pthread_create` y el manejo de la variable de tipo `pthread_t`.
- La operación de unión de hilos con la función `pthread_join`.
- La operación de adquirir un mutex con `pthread_mutex_lock` y su eventual liberación manual con `pthread_mutex_unlock`.

- Las funciones `pthread_cond_wait` y `pthread_cond_signal` para trabajar con condition variables.

Lamentablemente, el código fuente de esta biblioteca llamada "C2Petri" no se encuentra en línea, ya que las publicaciones son bastante antiguas.

En una tesis de máster más reciente, [Meyer, 2020] establece las bases de una semántica de redes de Petri para el lenguaje de programación Rust. Sin embargo, centra sus esfuerzos en el código de un solo hilo, limitándose a la detección de los deadlocks causados por la ejecución de la operación de `lock` dos veces sobre el mismo mutex en el hilo principal. Desafortunadamente, el código disponible en GitHub[4] como parte de la tesis ya no es válido para la nueva versión de *rustc*, puesto que las partes internas del compilador han cambiado significativamente en los últimos tres años.

En un *pre-print* de finales de 2022, [Zhang and Liua, 2022] implementan una traducción del código fuente de Rust a redes de Petri para encontrar deadlocks. La traducción se centra en los bloqueos muertos causados por dos tipos de bloqueos de la biblioteca estándar: `std::sync::Mutex` y `std::sync::RwLock`. La red de Petri resultante se expresa en el lenguaje de marcado de redes de Petri (Petri Net Markup Language (PNML)) y se introduce en el verificador de modelos Platform Independent Petri net Editor 2 (PIPE2)[5] para realizar el análisis de alcanzabilidad. Las llamadas a funciones se manejan de una forma muy diferente a la de este trabajo y las señales perdidas no se modelan en absoluto. El código fuente de su herramienta, denominada TRustPN, no está disponible públicamente en el momento de escribir este artículo. A pesar de estas limitaciones, los autores ofrecen un estudio muy detallado y actualizado de las herramientas de análisis estático para la verificación de código Rust que podría resultar atractivo para el lector interesado en ahondar en esta temática. Además, enumeran varios trabajos dedicados a formalizar la semántica del lenguaje de programación Rust que quedan fuera del alcance de este trabajo.

2.3. Bibliotecas de redes de Petri en Rust

Como parte del desarrollo de la traducción del código fuente a una red de Petri, es necesario utilizar una biblioteca de redes de Petri para el lenguaje de programación Rust. Una búsqueda rápida de los paquetes disponibles en *crates.io*[6], GitHub y GitLab reveló que, por desgracia, no existe ninguna biblioteca bien mantenida.

Se encontraron algunos simuladores de redes de Petri como:

- pns[7]: Programado en C. No ofrece la opción de exportar la red resultante a un formato

[4] https://github.com/Skasselbard/Granite
[5] https://pipe2.sourceforge.net/
[6] https://crates.io/
[7] https://gitlab.com/porky11/pns

estándar.

- PetriSim[8]: Un antiguo simulador DOS/PC programado en Bordland Pascal.
- WOLFGANG[9]: Un editor de redes de Petri en Java, mantenido por el Departamento de Informática de la Universidad de Friburgo, Alemania.

Desafortunadamente ninguno de ellos cumple los requisitos de la tarea.

Considerando que una red de Petri es un grafo, se evaluó la posibilidad de utilizar una biblioteca de grafos y modificarla para adaptarla a los objetivos de este trabajo. Se encontraron dos bibliotecas de grafos en Rust:

- petgraph[10]: La biblioteca más utilizada para gráficos en *crates.io*. Ofrece una opción para exportar al formato DOT.
- gamma[11]: Inestable y sin cambios desde 2021. No ofrece la posibilidad de exportar el gráfico.

Ninguna de las posibilidades satisface el requisito de exportar la red resultante al formato PNML. Además, si se utiliza una biblioteca de grafos, las operaciones de una red de Petri deben implementarse como un *wrapper* alrededor de un grafo, lo que reduce la posibilidad de optimizaciones para nuestro caso de uso y dificulta la extensibilidad a largo plazo del proyecto.

En conclusión, es imperativo implementar una biblioteca de redes de Petri en Rust desde cero como un proyecto independiente. Esto aporta una herramienta más a la comunidad que podría reutilizarse en el futuro.

2.4. Verificadores de modelos

La elección de un verificador de modelos adecuado es una parte vital de este trabajo porque es el responsable de verificar la ausencia de bloqueos. Afortunadamente se han desarrollado varios comprobadores de modelos para analizar redes de Petri.

El Model Checking Contest (MCC) [Kordon et al., 2021] organizado en la Universidad de la Sorbona de París es una gran fuente de verificadores de modelos de última generación. Se trata de un concurso anual en el que los verificadores de modelos presentados se ejecutan sobre una serie de modelos de redes de Petri procedentes del mundo académico y de la industria[12]. Estos modelos han sido aportados por muchas personas a lo largo de un periodo de más de una década

[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/
[11] https://github.com/metamolecular/gamma
[12] https://mcc.lip6.fr/2023/models.php

y el número total de puntos de referencia ha crecido paulatinamente a medida que se han ido añadiendo nuevos modelos.

Cada año, los puntos de referencia incluyen redes de lugares/transiciones (place/transition nets (P/T nets)), es decir, redes de Petri, y redes de Petri coloreadas (Colored Petri nets (CPN)). El número de lugares en las redes puede oscilar entre una docena y más de 70000 y las transiciones entre menos de un centenar y más de un millón. Esto pone de manifiesto la aplicabilidad práctica de los verificadores de modelos que participan en el concurso.

Los resultados se publican en la página web oficial (véase por ejemplo [Kordon et al., 2022]) y consisten en:

1. una lista de las herramientas cualificadas que participaron,
2. las técnicas aplicadas en cada una de las herramientas,
3. una sección dedicada a detallar las condiciones experimentales en las que se desarrolló el concurso (el hardware utilizado y el tiempo necesario para completar las ejecuciones),
4. los resultados en forma de tablas, gráficos e incluso los registros de ejecución de cada programa,
5. la lista de ganadores de cada categoría,
6. un análisis de la fiabilidad de las herramientas basado en la comparación de los resultados.

Un breve vistazo a las diapositivas de la edición de 2022[13] reproducidas en la Fig. 2.1 ilustra que varios verificadores de modelos han demostrado una participación ininterrumpida, con ejemplos notables que incluyen:

- Tool for Verification of Timed-Arc Petri Nets (TAPAAL) mantenida por la Universidad de Aalborg en Dinamarca[14], ganadora de una medalla de oro en la edición de 2023.
- Low-Level Petri Net Analyzer (LoLA) mantenido por la Universidad de Rostock en Alemania[15], ganador en ediciones anteriores y fue utilizado base para otros verificadores de modelos.
- ITS-tools [Thierry Mieg, 2015], que también se combinó con LoLA y obtuvo medallas en 2020[16].

Estas observaciones indican colectivamente la madurez y vitalidad de la comunidad de verificadores de modelos. El establecimiento de un panorama de herramientas bien desarrollado, fomentado por la colaboración y la difusión de código abierto de resultados, *benchmarks* y técnicas, presenta una valiosa oportunidad para aprovechar estas herramientas en el ámbito del

[13] https://mcc.lip6.fr/2022/pdf/MCC-PN2022.pdf
[14] https://www.tapaal.net/
[15] https://theo.informatik.uni-rostock.de/theo-forschung/tools/lola/
[16] https://github.com/yanntm/its-lola

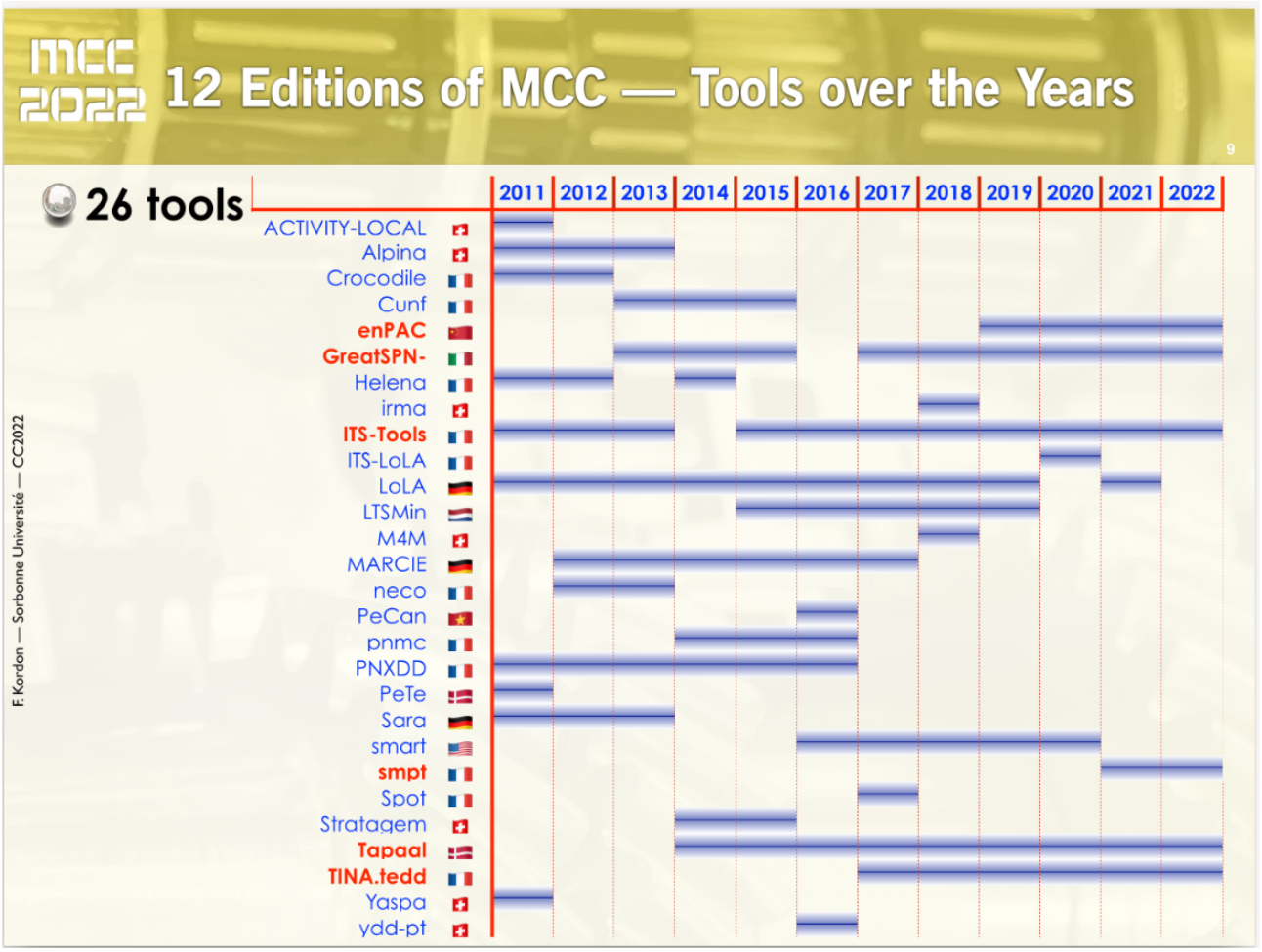

Figura 2.1: Participación de los verificadores de modelos en el MCC a lo largo de los años.

desarrollo de software. Concretamente, en el contexto de integrarlas como backends para un traductor para un lenguaje de programación específico que se encargue de automatizar el proceso de creación de modelos de redes de Petri. Capitalizando los esfuerzos académicos invertidos en los verificadores de modelos, se puede lograr una mayor seguridad y fiabilidad en los proyectos de software.

2.5. Formatos de archivo para intercambio de redes de Petri

Como se ha observado en el capítulo anterior, las redes de Petri son una herramienta muy utilizada para modelar sistemas de software. Sin embargo, debido a las diferentes clases de redes de Petri (redes de Petri simples, redes de Petri de alto nivel, redes de Petri con tiempo,

redes de Petri estocásticas, redes de Petri coloreadas, por nombrar algunas), diseñar un formato de archivo de intercambio estandarizado compatible con todas las aplicaciones ha resultado todo un reto. Una de las razones es que las redes de Petri pueden implementarse y representarse de múltiples formas, en función de los objetivos específicos, visto que son un tipo de grafo.

Para garantizar un cierto grado de interoperabilidad entre la herramienta desarrollada en el marco de esta tesis y otras herramientas existentes y futuras, es primordial investigar qué formatos de archivo sería más conveniente soportar. El objetivo es admitir formatos de archivo que sean adecuados tanto para el análisis como para la visualización, permitiendo la posibilidad de ampliación a formatos adicionales en el futuro, a través de una API bien definida en la biblioteca de redes de Petri. Una revisión de la literatura condujo a tres formatos de archivo relevantes que se presentan a continuación.

2.5.1. Petri Net Markup Language

El Petri Net Markup Language (PNML)[17] es un formato de archivo estándar diseñado para el intercambio de redes de Petri entre distintas herramientas y aplicaciones de software. Su desarrollo se inició en el "Meeting on XML/SGML based Interchange Formats for Petri Nets" celebrada en Aarhus en junio de 2000 [Jüngel et al., 2000, Weber and Kindler, 2003] con el objetivo de proporcionar un formato estandarizado y ampliamente aceptado para redes de Petri. PNML es una norma ISO que consta, a partir de 2023, de tres partes:

- ISO/IEC 15909-1:2004[18] (y su última revisión ISO/IEC 15909-1:2019[19]) para conceptos, definiciones y notación gráfica.
- ISO/IEC 15909-2:2011[20] para la definición de un formato de transferencia basado en XML.
- ISO/IEC 15909-3:2021[21] para las extensiones y los mecanismos de estructuración.

Se ha convertido en un estándar *de facto* para intercambiar modelos en redes de Petri entre diferentes herramientas y sistemas. Es el resultado de muchos años de duro trabajo para unificar la notación, tal y como se expone en [Hillah and Petrucci, 2010].

PNML ha sido diseñado para ser un formato flexible y extensible que pueda representar diferentes clases de redes de Petri, incluidas las redes de Petri simples y las redes de Petri de alto nivel. Se basa en Extensible Markup Language (XML), lo que facilita su lectura y análisis tanto por humanos como por máquinas. Además, PNML admite el uso de metadatos para proporcionar información adicional sobre los modelos de redes de Petri, como la autoría, la fecha de creación e información sobre licencias.

[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

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 which makes it easy to read and parse by humans and machines alike. Additionally, supports the use of metadata to provide additional information about the Petri net models, such as authorship, date of creation, and licensing information.

El desarrollo de PNML ha mejorado significativamente la interoperabilidad y el intercambio de modelos de redes de Petri entre diferentes herramientas y sistemas. Antes de la adopción de PNML el intercambio de modelos de redes de Petri era una tarea ardua puesto que las distintas herramientas utilizaban formatos propietarios que a menudo eran incompatibles entre sí. El PNML ha simplificado enormemente este proceso, permitiendo a investigadores y profesionales compartir y colaborar en modelos de redes de Petri con facilidad. Su uso también ha facilitado el desarrollo de nuevas herramientas y aplicaciones de software para redes de Petri porque proporciona un formato estándar que puede ser analizado y procesado fácilmente por distintos sistemas. En particular es el formato utilizado en [Zhang and Liua, 2022] y está soportado en [Meyer, 2020].

2.5.2. Formato GraphViz DOT

El formato DOT es un lenguaje de descripción de grafos utilizado para crear representaciones visuales de grafos y redes, que forma parte de la suite de código abierto GraphViz[22]. Fue creado a principios de la década de 1990 en AT&T Labs Research como un lenguaje sencillo, conciso y legible por humanos para la descripción de grafos. La suite GraphViz proporciona varias herramientas para trabajar con archivos DOT, incluida la capacidad de generar automáticamente diseños para gráficos complejos y de exportar visualizaciones en diversos formatos, como PNG, PDF y SVG.

DOT puede utilizarse para representar redes de Petri en un formato gráfico, lo que facilita la visualización de la estructura y el comportamiento del sistema que se está modelando. Resulta especialmente útil para visualizar redes de Petri de gran tamaño, ya que el usuario puede navegar por la imagen para comprender cómo fluyen las marcas por la red.

El formato DOT está basado en texto plano y es fácil de usar, convirtiéndolo en una opción popular para generar representaciones visuales de gráficos. Esta facilidad también significa que los archivos DOT pueden ser generados fácilmente por programas y pueden ser leídos por una amplia gama de herramientas de software, un aspecto esencial para la interoperabilidad. Además, DOT permite especificar diversas propiedades de los grafos, como formas de nodos, colores y estilos [Gansner et al., 2015] que pueden utilizarse para representar diferentes aspectos de una red de Petri, como lugares, transiciones y arcos. Esta flexibilidad a la hora de especificar las propiedades visuales también permite a los usuarios personalizar la visualización según sus necesidades y resaltar características particulares de la red de Petri que sean relevantes para su análisis.

[22] https://graphviz.org/

2.5.3. LoLA - Low-Level Petri Net Analyzer

Low-Level Petri Net Analyzer (LoLA) [Schmidt, 2000] es un verificador de modelos de última generación cuyo desarrollo comenzó en 1998 en la Universidad Humboldt de Berlín. Actualmente lo mantiene la Universidad de Rostock y se publica bajo la Licencia Pública General Affero de GNU. LoLA es una herramienta que puede comprobar si un sistema satisface una propiedad dada expresada en Computational Tree Logic* (CTL*). Su punto fuerte es la evaluación de propiedades sencillas como la libertad de bloqueo (*deadlock freedom*) o la alcanzabilidad, tal y como se indica en la página web. Este es el verificador de modelos utilizado en [Meyer, 2020] y en este trabajo. En consecuencia, es necesario implementar el formato de archivo requerido por la herramienta. En la Sec. 5.2 se presentan ejemplos.

Capítulo 3

Diseño de la solución propuesta

Una vez cubiertos los temas de fondo pertinentes, podemos proceder a profundizar en los aspectos específicos del diseño del proceso de traducción. El diseño está marcado por tres opciones arquitecturales cruciales sobre las que se profundizará en este capítulo:

1. La decisión de utilizar el compilador Rust como backend para la traducción.
2. Basar la traducción en el Mid-level Intermediate Representation (MIR).
3. Hacer un *inlining* de las llamadas a funciones en la red de Petri.

A lo largo de este capítulo, analizaremos en profundidad los mecanismos internos del compilador de Rust y sus etapas de compilación relevantes para este trabajo.

3.1. En busca de un backend

Para ponerlo de forma sucinta, existen dos enfoques para traducir código Rust a redes de Petri. La primera opción es crear un traductor desde cero, mientras que la segunda consiste en basarse en una herramienta ya existente.

La primera opción puede parecer atractiva al principio, teniendo en cuenta que da al desarrollador la libertad de moldear la herramienta según sus deseos. Se pueden añadir funciones según las necesidades y adaptar las estructuras de datos al propósito específico. Sin embargo, esta flexibilidad tiene un alto precio. Para dar soporte a un subconjunto razonable del lenguaje de programación Rust, es necesario invertir grandes cantidades de esfuerzo en la tarea. Las construcciones complejas del lenguaje, como las macros, los *generics* o mismo el rico sistema de tipos, deben ser comprendidas en sus detalles más intrincados para poder ser traducidas con eficacia. El resultado es, esencialmente, un nuevo compilador para código Rust. Teniendo en cuenta que el compilador de Rust se desarrolló a lo largo de muchos años y con el apoyo de una gran comunidad de colaboradores, queda claro que este camino no es más que una duplicación

de trabajo. De hecho, es una labor hercúlea que requeriría la dedicación a tiempo completo de un equipo completo para mantener al día de los cambios más recientes en el lenguaje Rust y en el compilador.

Por otro lado, existe la posibilidad de integrarse con el compilador de Rust existente, que está disponible bajo una licencia de código abierto y su documentación es extensa y se actualiza con regularidad. Esto libera en parte a la implementación de tener que ocuparse de los cambios en el lenguaje, lo que da más tiempo para centrarse en las características que añaden valor a los usuarios. De ahí que el compilador desempeñe el papel de *backend* en el que se apoya el análisis estático. Por supuesto, esto requiere aprender las interioridades del compilador, pero no es la primera vez que una herramienta se propone ello. A modo de ejemplo, el linter oficial de Rust, *clippy*[1], analiza el código Rust en busca de construcciones incorrectas, ineficaces o no idiomáticas. Se trata de una herramienta muy valiosa para los desarrolladores que va más allá de las comprobaciones estándar realizadas durante la compilación.

Dar soporte a todas las características del lenguaje desde el principio y colaborar con la comunidad es clave para el éxito de la solución propuesta. Por lo tanto, es aconsejable integrarse con el ecosistema existente y reutilizar todo el trabajo posible. Por todo ello, este proyecto se basa en *rustc*. A continuación estudiaremos con más detalle los componentes centrales del compilador de Rust.

3.2. El compilador de Rust: *rustc*

El compilador de Rust, *rustc*, se encarga de traducir el código Rust en código ejecutable. Sin embargo, *rustc* no es un compilador tradicional en el sentido de que realiza múltiples pasadas sobre el código, como se describe en la Sec. 1.6. En su lugar, *rustc* está construido sobre un sistema basado en consultas que soporta la compilación incremental.

En el sistema de consulta de *rustc*, el compilador calcula un grafo de dependencias entre los artefactos de código, incluidos los archivos fuente, los crates y los artefactos intermedios, como los archivos objeto. A continuación, el sistema de consulta utiliza este grafo para recompilar eficientemente sólo aquellos artefactos que hayan cambiado desde la última compilación[2]. Esta compilación incremental puede reducir significativamente el tiempo de compilación de grandes proyectos, facilitando el desarrollo y la iteración del código Rust.

El sistema de consulta también permite al compilador de Rust realizar otras optimizaciones, como la memoización y el almacenamiento en caché de resultados intermedios. Por ejemplo, si el valor de retorno de una función se ha calculado antes, el sistema de consulta puede devolver el resultado almacenado en caché en lugar de volver a calcularlo, lo que reduce aún más el tiempo de compilación.

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

Otra elección de diseño importante en *rustc* es el *interning*. *Interning* es una técnica para almacenar cadenas de texto y otras estructuras de datos de forma eficiente en memoria. En lugar de almacenar varias copias de la misma cadena o estructura de datos, el compilador de Rust almacena sólo una copia en un *allocator* especial llamado *arena*. Las referencias a los valores almacenados en la arena se pasan de una parte a otra del compilador y pueden compararse de forma barata comparando punteros. Esto permite reducir el uso de memoria y acelerar las operaciones que comparan o manipulan cadenas de texto y estructuras de datos.

rustc utiliza la infraestructura del compilador LLVM[3] para realizar la generación de código de bajo nivel y la optimización. LLVM proporciona un marco flexible para compilar código a una variedad de *targets*, incluyendo código máquina nativo y WebAssembly (WASM). El compilador de Rust utiliza LLVM para optimizar el código en términos de rendimiento y generar código de alta calidad para una gran variedad de plataformas. En lugar de generar código máquina, sólo necesita generar la intermediate representation (IR) de LLVM del código fuente y luego ordenar a LLVM que lo transforme al objetivo de compilación (*compilation target*), aplicando las optimizaciones deseadas.

rustc está programado en Rust. Para compilar la versión más reciente del compilador y la versión más reciente de la biblioteca estándar que lo acompaña, se utiliza una versión ligeramente más antigua de *rustc* y de la biblioteca estándar. Este proceso se denomina *bootstrapping* e implica que uno de los principales usuarios de Rust es el propio compilador de Rust. Teniendo en cuenta que cada seis semanas se publica una nueva versión estable, el bootstrapping implica una gran complejidad y se describe detalladamente en la documentación[4] y en conferencias [Nelson, 2022] y tutoriales [Klock, 2022] hechos por miembros del Rust team.

3.2.1. Etapas de compilación

La existencia del sistema de consulta no implica que *rustc* no tenga fases de compilación en absoluto. Al contrario, se requieren varias etapas de compilación para transformar el código fuente de Rust en código máquina que pueda ejecutarse en una computadora. Estas etapas implican múltiples representaciones intermedias del programa, cada una optimizada para un propósito específico. A continuación describiremos brevemente estas etapas. Encontrará una descripción más completa en la documentación[5].

Lexado y análisis sintáctico

En primer lugar, el texto fuente en bruto de Rust es analizado por un *lexer* de bajo nivel. En esta etapa, el texto fuente se convierte en un flujo (*stream*) de unidades atómicas de código fuente conocidas como tokens.

[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

A continuación se realiza el análisis sintáctico (*parsing*). El flujo de tokens se convierte en un AST. Aquí se produce el interning de los valores de cadena. La expansión de macros, la validación del AST, la resolución de nombres y el linting temprano también tienen lugar durante esta e t a p a . La representación intermedia resultante de esta etapa es, a fin de cuentas, el AST.

HIR lowering

Posteriormente, el AST se convierte en High-Level Intermediate Representation (HIR). Este proceso se conoce como "rebajar" (*lowering*). Esta representación se parece al código Rust pero con construcciones complejas convertidas en versiones más simples. Por ejemplo, todos los bucles `while` y `for` se convierten en versiones más simples con bucles `loop`.

La HIR se utiliza para realizar algunos pasos importantes:

1. *Inferencia de tipo (type inference)*: La detección automática del tipo de una expresión, por ejemplo, al declarar variables con `let`.
2. *Resolución de traits (trait solving)*: Garantizar que cada bloque de implementación (`impl`) hace referencia a un trait válido y existente.
3. *Comprobación de tipos (type checking)*: Este proceso convierte los tipos escritos por el usuario en la representación interna utilizada por el compilador. Es, en otras palabras, donde se internan los tipos. Después, utilizando esta información, se verifica la seguridad de tipos, la correctitud y la coherencia.

MIR lowering

En esta etapa, la HIR se rebaja a Mid-level Intermediate Representation (MIR), que se utiliza para el *borrow checking*. Como parte del proceso, se construye la Typed High-Level Intermediate Representation (THIR), que es una representación más fácil de convertir a MIR que la HIR.

La THIR es una versión aún más "desugarizada" (*desugared*) del HIR. Se utiliza para el *pattern matching* y el *exhaustiveness matching*. Es similar a la HIR pero con todos los tipos y llamadas a métodos explícitos. Además se incluyen desreferencias implícitas cuando es necesario.

Muchas optimizaciones se realizan sobre la MIR puesto que sigue siendo una representación muy genérica. Las optimizaciones son en algunos casos más fáciles de realizar sobre la MIR que sobre la posterior IR de LLVM.

Generación de código

Esta es la última etapa en la producción de un binario. Incluye la llamada a LLVM para la generación de código y las optimizaciones correspondientes. Para ello, la MIR se convierte en LLVM IR.

LLVM IR es la forma estándar de entrada para el compilador LLVM que utilizan todos los compiladores que utilizan LLVM, como el compilador de C *clang*. Es un tipo de lenguaje ensamblador bien anotado y diseñado para que otros compiladores puedan producirlo fácilmente. Además, está diseñado para ser lo suficientemente rico como para permitir a LLVM realizar varias optimizaciones sobre él.

LLVM transforma el LLVM IR a código máquina y aplica muchas más optimizaciones. Por último, los archivos objeto que contienen código ensamblador pueden enlazarse (*linking*) entre sí para formar el binario.

3.2.2. Rust nightly

Comprender el modelo de lanzamiento de versiones de Rust es indispensable para implementar con éxito la herramienta propuesta en este trabajo. La razón es que para utilizar las crates de *rustc* como dependencia en nuestro proyecto, debe compilarse con la versión *nightly*.

El compilador nightly de Rust se refiere a una compilación específica de *rustc* que se actualiza cada noche con los últimos cambios y mejoras pero que también incluye características experimentales o inestables que aún no forman parte de la versión estable. En Rust, el lenguaje y su biblioteca estándar se versionan utilizando un modelo de "tren de versiones" (*release train*), en el que existen tres canales de versiones principales: estable, beta y nightly[6].

La versión estable del compilador de Rust es la más utilizada y recomendada para su uso en producción. Pasa por un riguroso proceso de pruebas y estabilización para garantizar que proporciona una experiencia estable y fiable a los desarrolladores. La versión estable sólo incluye características y mejoras que han sido revisadas a fondo, testeadas y consideradas lo suficientemente estables para su uso en producción.

Por otro lado, el compilador nightly de Rust es la versión más *bleeding-edge*, en la que se introducen a diario nuevas características, correcciones de errores y cambios experimentales. Es utilizado por los desarrolladores y colaboradores del lenguaje Rust con fines de prueba y desarrollo, pero no se recomienda su uso en producción debido a la inestabilidad potencial y a la falta de soporte a largo plazo.

Cada característica exclusiva de la versión nightly está detrás de una *feature flag*. Sólo pueden utilizarse al compilar con la *toolchain* nightly. Las feature flags pueden habilitar

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

- construcciones sintácticas que no están disponibles en la versión estable,
- funciones de biblioteca exclusivas de la versión nocturna,
- soporte para instrucciones de hardware específicas de un ISA o plataforma determinados,
- flags adicionales del compilador.

La lista completa de banderas de características se encuentra en [Rust Project, 2023e] y contiene más de 500 entradas en total. De forma más concisa, el lenguaje Rust utilizado dentro de *rustc* es un superconjunto del lenguaje Rust estable utilizado fuera de él. Estas diferencias deben tenerse en cuenta cuando se trabaje en el compilador o se construya software que dependa directamente del compilador.

3.3. Selección de un punto de partida adecuado para la traducción

En esta sección, se elucidan los motivos para seleccionar la Mid-level Intermediate Representation (MIR) como punto de partida para la traducción a una red de Petri. Esta elección de diseño arquitectural se justifica por varias razones.

3.3.1. Beneficios

En primer lugar, la MIR es la IR más baja en rustc que aún es independiente de la arquitectura de la computadora. Captura la semántica del código Rust después de que haya sido sometido a una serie de pases de optimización sin depender de los detalles de cualquier máquina en particular. Al interceptar la traducción en esta fase, la herramienta de análisis estático aprovecha las ventajas de estas optimizaciones, como el plegado de constantes (*constant folding*), la eliminación de código muerto y el *inlining*, lo que da como resultado una representación de la red de Petri más eficiente y, en general, más pequeña.

En segundo lugar, interceptar la compilación una vez completadas las etapas anteriores ofrece una ventaja en términos de eficiencia y reutilización del código. En esta etapa, el compilador de Rust ya ha realizado pasos cruciales como el *borrow checking*, la comprobación de tipos, la monomorfización del código genérico y la expansión de macros, entre otros. Estos pasos consumen muchos recursos e implican un análisis complejo del código Rust para garantizar su correctitud y seguridad. Reimplementar estos pasos en nuestra herramienta desde cero sería redundante y llevaría mucho tiempo. Requeriría duplicar los esfuerzos del compilador de Rust y podría introducir posibles incoherencias o errores. Al construir sobre el MIR existente, aprovechamos al máximo el trabajo ya realizado por *rustc*. Esto no sólo ahorra esfuerzo, sino que también alinea nuestra herramienta de análisis estático con el mismo nivel de correctitud y seguridad que el compilador de Rust.

En tercer lugar, simplifica la tarea de mantenimiento de mantenerse al día con las continuas incorporaciones al lenguaje Rust y a su compilador. Rust es un lenguaje en rápida evolución y su compilador se actualiza constantemente con nuevas características, correcciones de errores y mejores de performance. Reutilizar la MIR significa que nuestra herramienta puede beneficiarse de estas actualizaciones sin tener que implementar y mantener esos cambios de forma independiente. Esto proporciona en general una solución de análisis estático más robusta y fiable.

Adicionalmente, como se explicará en la siguiente sección, la MIR se basa en el concepto de grafo de flujo de control (control flow graph (CFG)), o en otras palabras, un tipo de grafo que se encuentra en los compiladores. Esto significa que tanto la MIR como las redes de Petri son representaciones gráficas, lo que hace que la MIR sea especialmente adecuada para una traducción. Tanto la MIR como las redes de Petri pueden considerarse modelos gráficos que capturan las relaciones e interacciones entre diferentes entidades. El grafo MIR representa el flujo de ejecución subyacente dentro de un programa Rust, mientras que una red Petri captura las transiciones de estado y las ocurrencias de eventos en un sistema. Consecuentemente, resulta más fácil convertir la MIR en una red de Petri, dado que la estructura del grafo y las relaciones ya están presentes. Esto permite un proceso de traducción más directo y eficiente sin tener que crear una estructura de grafos de la nada, lo que resulta en una mejor integración entre el MIR y el modelo de red de Petri para la detección de deadlocks.

Para concluir, trabajar con la MIR crea sinergias con la compilación incremental y el análisis modular. De hecho, una de las razones por las que se introdujo MIR en primer lugar fue la compilación incremental [Matsakis, 2016]. Aunque no es obligatorio en la implementación inicial, la herramienta podría beneficiarse de la compilación incremental y realizar análisis por crate/por módulo, lo que permitiría un análisis más rápido y eficiente de grandes bases de código Rust.

3.3.2. Limitaciones

A pesar de los numerosos beneficios, el enfoque de basar la traducción en la Mid-level Intermediate Representation (MIR) tiene algunas limitaciones.

La más importante es que la MIR está sujeta a cambios. No se ofrecen garantías de estabilidad en cuanto a cómo se traducirá el código Rust a MIR o cuáles son sus elementos constitutivos. Se trata de detalles internos que los desarrolladores del compilador se reservan para sí mismos. En resumen, la MIR como interfaz no es estable. A medida que se sigue trabajando en el compilador, la MIR sufre modificaciones para incorporar nuevas características del lenguaje, optimizaciones o correcciones de errores, lo que puede requerir frecuentes actualizaciones y ajustes en el proceso de traducción, aumentando el coste de mantenimiento.

En el transcurso de este proyecto esta situación se produjo en varias ocasiones. A modo de ejemplo, en el periodo comprendido entre mediados de febrero de 2023 y mediados de abril de

2023, el código se modificó 7 veces para dar cabida a estos cambios. Siempre fueron de unas pocas líneas de código y se detectaron mediante pruebas. Hablaremos de cómo las pruebas desempeñan un papel importante a la hora de lidiar con estos cambios en la Sec. 5.2.

En la misma línea, [Meyer, 2020] también se basó en la MIR pero no incorporó pruebas para hacer frente a las versiones nightly más recientes. Como resultado, la cadena de herramientas se fijó a una versión nightly exacta[7] para evitar que la implementación se rompiera antes de la publicación de la tesis.

Otro inconveniente digno de mención es que, en algunos casos, el código genérico podría adoptar la forma de una función cuyo comportamiento puede ser modelado por la misma red de Petri en todos los casos. En estas circunstancias, el MIR podría "condensarse" aún más antes de traducirlo a una red de Petri. Del mismo modo, algunas partes de la MIR pueden ser superfluas para el análisis de detección de bloqueo y su traducción puede agrandar la salida, lo que ralentiza el análisis de alcanzabilidad realizado por el verificador de modelos. Esto puede contrarrestarse con optimizaciones cuidadosas que se propondrán en las Sec. 6.1 y 6.2.

3.3.3. Síntesis

En conclusión, a pesar de los inconvenientes mencionados anteriormente, interceptar la traducción en el nivel MIR ofrece ventajas significativas, entre las que se incluyen la maximización de la utilización del código del compilador existente, la reducción del esfuerzo de implementación y un mapeo más natural a las redes de Petri. Estas ventajas superan a los contras y hacen de la MIR un punto de partida convincente para la traducción en el contexto de la construcción de una herramienta de análisis estático para detectar deadlocks y señales perdidas en el código Rust.

Tanto [Meyer, 2020] como [Zhang and Liua, 2022] basan también sus traducciones en la MIR y, hasta donde sabe este autor, no existe ninguna herramienta análoga que realice una traducción a redes de Petri partiendo de una representación intermedia de nivel superior.

3.4. Mid-level Intermediate Representation (MIR)

En esta sección se ofrece una visión general de la Mid-level Intermediate Representation (MIR). La MIR se introdujo en la RFC 1211[8] en agosto de 2015. Exploraremos sus diferentes partes, cómo se mapean en ellas diferentes fragmentos de código y la estructura de grafos subyacente.

```
1 // WARNING: This output format is intended for human consumers only
2 // and is subject to change without notice. Knock yourself out.
```

[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: Programa Rust sencillo para explicar los componentes de la MIR.

```
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
    }

    bb3: {
        return;                          // scope 0 at src/main.rs:7:2: 7:2
    }
```

```
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 del Listado 3.1 compilado utilizando rustc 1.71.0-nightly en modo debug.

Considere el código de ejemplo que aparece en el Listado 3.1, el MIR[9] correspondiente se muestra en el Listado 3.2. Observe la advertencia explícita en la parte superior de la salida generada. Se omitirá en los listados posteriores por simplicidad. Además, la salida depende de los siguientes factores:

- La versión de *rustc* en uso, alternativamente el canal de lanzamiento (estable, beta o nightly).
- El tipo de compilación: *debug* o *release*. Por defecto, el comando cargo build genera un *debug build*, mientras que `cargo build -release` produce un *release build*.

Para ilustrar esta variabilidad, el Listado 3.3 muestra la salida al compilar el mismo programa en modo *release*. La característica distintiva que se encuentra en las compilaciones *release* es la presencia de las sentencias `StorageLive` y `StorageDead`. Por otro lado, las compilaciones de depuración (*debug*) generan MIR más cortos y claros que se acercan más a lo que escribió el usuario. Por esta razón, a menos que se indique lo contrario, los Listados de este trabajo contienen MIR generados en *debug builds*.

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

[9]Los comentarios de la MIR se han modificado ligeramente para mejorar el resultado

```
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 del Listado 3.1 compilado usando rustc 1.71.0-nightly en modo release.

El formato específico al convertir MIR a una cadena de texto sólo ha cambiado ligeramente con el tiempo. Consulte [Meyer, 2020, Section 3.3] para ver un ejemplo de salida más antiguo de mediados de 2019.

Como se indica en la Sec. 3.3, la MIR se deriva de un grafo de flujo de control (control flow graph (CFG)) previamente existente en el compilador Rust. Fundamentalmente, un CFG es una representación gráfica de un programa que expone el flujo de control subyacente.

3.4.1. Componentes de la MIR

La MIR está formada por funciones. Cada función se representa como una serie de bloques básicos (basic blocks (BB)) conectados por aristas dirigidas. Cada BB contiene cero o más sentencias o *statements* (normalmente abreviadas como "STMT") y por último una sentencia terminadora (*terminator statement*), para abreviar *terminator*. El terminator es la única sentencia en la que el programa puede emitir una instrucción que dirija el flujo de control a otro bloque básico dentro de la misma función o para llamar a otra función. Las bifurcaciones como en las sentencias `match` o `if` de Rust sólo pueden producirse en los terminators. Los terminators desempeñan el papel de mapear las construcciones de alto nivel para la ejecución condicional y los bucles a la representación de bajo nivel en código máquina como simples instrucciones `branch` de bifurcación condicional o incondicional.

En la Fig. 3.1 se presenta la representación gráfica de la MIR que aparece en el Listado 3.2. Los statements están coloreados en azul claro y los terminators en rojo claro. Para que el tipo de declaración terminadora sea más claro, se han añadido anotaciones adicionales como en `CALL:` o `DROP:`.

Debe tenerse en cuenta que la llamada a la función `std::env::args().len()` en la línea 2 del Listado 3.1 puede retornar con éxito o fallar. Un fallo desencadena un desenrollado de la pila (*stack unwinding*), finalizando el programa e informando de un error. Esto está representado por la bifurcación al final de BB1, donde la ejecución del código puede tomar el camino de la izquierda o el de la derecha en el gráfico. La bifurcación izquierda (BB4 y BB5) corresponde a la ejecución correcta del programa, mientras que la bifurcación derecha se refiere a la terminación anormal del programa.

Existen diferentes tipos de terminadores y éstos son específicos de la semántica de Rust. Presentaremos algunos de ellos para aclarar el significado del ejemplo presentado.

- Como era de esperar, un terminador de tipo `CALL:` llama a una función, que devuelve un valor, y continúa la ejecución hasta el siguiente BB.
- Un terminador de tipo `DROP:` libera la memoria de la variable pasada. Ejecuta los destructores[10] y realiza todas las tareas de limpieza necesarias. A partir de ese momento, la variable ya no puede utilizarse en el programa.
- `RETURN:` retorna de la función. El valor de retorno se almacena siempre en la variable local `_0`, como veremos en breve.

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

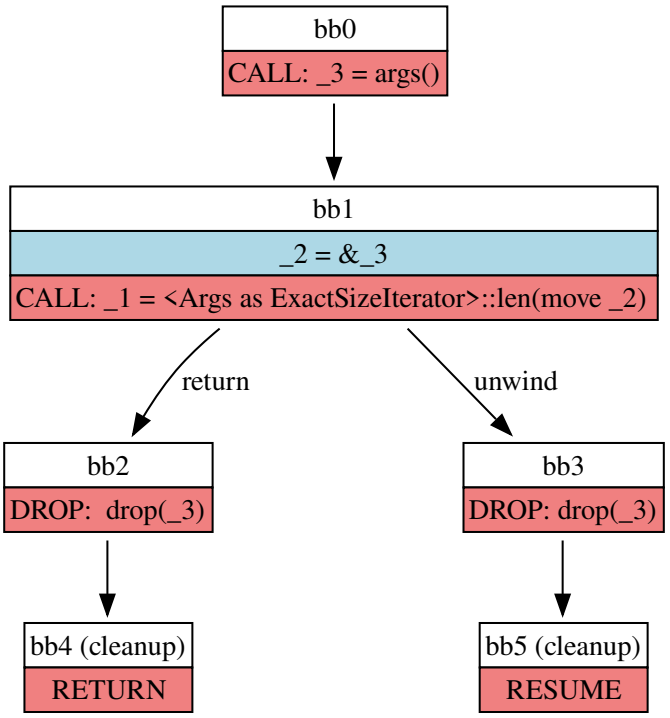

Figura 3.1: Representación gráfica del flujo de control de la MIR mostrada en el Listado 3.2.

- `RESUME`: indica que el proceso debe continuar desenrollándose (*unwinding*). De forma análoga a un retorno, esto marca el final de esta invocación de la función. Sólo se permite en bloques de limpieza.

La lista completa de tipos de terminadores puede consultarse en la documentación nightly[11]. Otros tipos de terminadores se tratarán en detalle en la Sec. 4.4.3.

En cuanto a las variables, los datos en MIR pueden dividirse en dos categorías: *locals* y *places*. Es sumamente importante observar que estos "places" no están relacionados con los lugares o *places* de las redes de Petri. Los places se utilizan para representar todo tipo de ubicaciones de memoria (incluidos los alias), mientras que las locals se limitan a las ubicaciones de memoria basadas en la pila (*stack*), es decir, las variables locales de una función. En otras palabras, los *places* son más generales y las *locals* son un caso especial de un lugar, por lo que los places no siempre son equivalentes a las locals. Convenientemente, todos los lugares son también locales en la Fig. 3.1.

Los locales se identifican mediante un índice no negativo creciente y son emitidos por el compilador como una cadena de la forma "`_<index>`". En particular, el valor de retorno de la función siempre se almacena en el primer local `_0`. Esto coincide estrechamente con la representación de bajo nivel en la pila.

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

3.4.2. Ejemplo paso a paso

En esta subsección, daremos una breve explicación de lo que ocurre en cada bloque básico de la Fig. 3.1 para cubrir toda la información necesaria para las siguientes secciones. Por otra parte, esto ilustra cómo la salida de la MIR representa construcciones de nivel superior que se encuentran a menudo al programar en Rust.

BB0

- La función `main()` comienza en BB0.
- Se llama a una función (`std::env::args()`) para obtener un iterador sobre los argumentos pasados al programa.
- El valor de retorno de la función, el iterador, se asigna a la local `_3`.
- La ejecución continúa en BB1.

BB1

- Se genera una referencia al iterador almacenado en `_3` y se almacena en el local `_2` (similar al operador "&" en C). Esto es necesario para llamar a métodos porque los métodos reciben una referencia a un struct del mismo tipo (`&self`) como primer argumento.
- La referencia almacenada en `_2` se pasa por movimiento al método `std::env::Args::len()` y se llama a la función.
- El valor de retorno de la función, el número de argumentos pasados a la función, se asigna al local `_1`.
- La ejecución continúa en BB2 si tiene éxito, en BB4 en caso de *panic*.

BB2

- La variable `_3`, cuyo valor es el iterador sobre los argumentos, se elimina (*dropped*) puesto que ya no es necesaria.
- La ejecución continúa en BB3.

BB3

- La función retorna. El valor de retorno (local `_0`) es de tipo "unit"[12] que es similar a una función `void` en C, es decir, no devuelve nada. Así es como se definió `main()` en el Listado 3.1.

BB4

- La variable `_3`, cuyo valor es el iterador sobre los argumentos, se elimina (*drop*) puesto que ya no es necesaria.
- Si el *drop* tiene éxito, la ejecución continúa en BB5, de lo contrario, finaliza el programa inmediatamente.

BB5

- Continúe desenrollando la pila. Este es el protocolo estándar definido para manejar los casos de error catastrófico que no pueden ser manejados por el programa. Los detalles de implementación se pueden encontrar en la documentación[13]

3.5. Inlining de funciones en la traducción a redes de Petri

En esta sección se presenta un análisis exhaustivo y la motivación de la tercera decisión de diseño enumerada al principio del capítulo, a saber, el inlining de las llamadas a funciones.

El modelado de funciones en PN es un aspecto crucial de la traducción porque es la unidad básica del MIR. Al representar las funciones en la MIR como PN y conectarlas entre sí, el flujo de control y los datos compartidos entre los hilos del programa pueden capturarse en un marco formal. Posteriormente, la red de Petri es analizada por un verificador de modelos con el fin de identificar posibles deadlocks o señales perdidas. Este enfoque es especialmente útil cuando se trabaja con sistemas grandes y complejos que pueden tener muchos hilos y funciones interrelacionados, en los que la situación de deadlock puede no ser evidente ni siquiera para un revisor de código experimentado.

Al traducir funciones MIR a PN, una cuestión clave que se plantea es si reutilizar la misma representación para cada llamada a una función específica o hacer un "inline" de la representación correspondiente cada vez que se llama a la función. Expresado de otro modo, cada función mapea a una subred en la PN final obtenida tras la traducción, es decir, un subgrafo conectado

[12] https://doc.rust-lang.org/std/primitive.unit.html
[13] https://rustc-dev-guide.rust-lang.org/panic-implementation.html

formado por los lugares y transiciones que modelan el comportamiento de la función específica. Esta parte más pequeña de la red puede estar presente sólo una vez en la PN y todas las llamadas a esta función se conectan a ella, o repetirse para cada instancia de una llamada a la función en el código Rust.

Reutilizar el mismo modelo para cada función parece a primera vista más eficiente, ya que la PN obtenido es menor. Sin embargo, este enfoque también puede dar lugar a estados no válidos que no estaban presentes en el programa Rust original. Éstos pueden ser la fuente de falsos positivos durante la detección de bloqueos porque que estos estados extraños pueden violar las garantías de seguridad ofrecidas por el compilador.

Por otro lado, un inline del modelo cada vez que se llama a una función da como resultado un PN más grande que requiere más memoria y tiempo de CPU para ser analizado pero también puede mejorar la precisión del análisis al garantizar que cada llamada a una función esté representada por una estructura de red de Petri independiente que capture sus dependencias de datos específicas en el contexto en el que se produce la llamada a la función en el código.

3.5.1. El caso básico

El impacto de estos sutiles detalles sólo puede comprenderse plenamente con un ejemplo apropiado. En consecuencia, consideremos primero la abstracción más simple de una llamada a una función en el lenguaje de las redes de Petri, formada por una sola transición y dos lugares que representan el inicio y el final de la función. Esto se ve en la Fig. 3.2. La llamada a la función se trata como una caja negra, todos los detalles se abstraen en la transición. Sólo nos importa dónde empieza y dónde acaba la función.

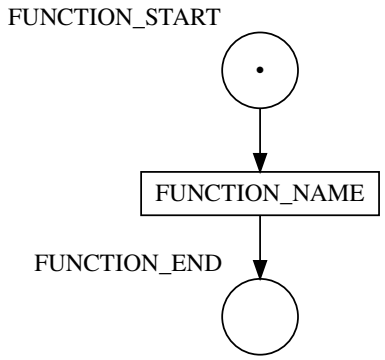

Figura 3.2: El modelo de red de Petri más simple posible para una llamada de función.

Observe ahora una función de este tipo en el contexto de un programa Rust. El Listado 3.4 ofrece un ejemplo sencillo en el que una función es llamada cinco veces consecutivas en un bucle `for`. Una posible PN que modele el programa se encuentra en la Fig. 3.3. Cabe destacar que esta red y las siguientes de esta sección *no* son el resultado de una traducción de la MIR. Son

simplificaciones para ilustrar las dificultades de tratar con funciones llamadas en varios lugares en el código.

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

Listing 3.4: Un programa sencillo de Rust con una llamada repetida a una función.

3.5.2. Una caracterización del problema

El escenario problemático no ha surgido hasta ahora. Sólo se manifiesta cuando se llama a una función en al menos dos lugares distintos del código o, en términos más sencillos, cuando la expresión `simple_function()` aparece dos veces o más. El Listado 3.5 satisface esta condición y está diseñado para mostrar el comportamiento extráneo descrito al principio de la sección.

```
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: Un sencillo programa Rust que llama a una función en dos lugares diferentes.

Como ya se ha dicho, el primer enfoque para modelar el programa consiste en reutilizar el modelo de función para ambas llamadas. Esto se muestra en la Fig. 3.4.

Es evidente para el lector que el programa del Listado 3.5 nunca llama a la macro `panic!` y siempre termina con éxito, dado que la variable `second_call` nunca es `true` antes de la línea 9.

Sin embargo, la PN representada en la Fig. 3.4 es conspicuamente defectuosa, lo que la hace inadecuada como modelo para el programa. La razón es que después de disparar la transición etiquetada `RETURN_simple_function` se coloca un token en `check_flag` pero también en

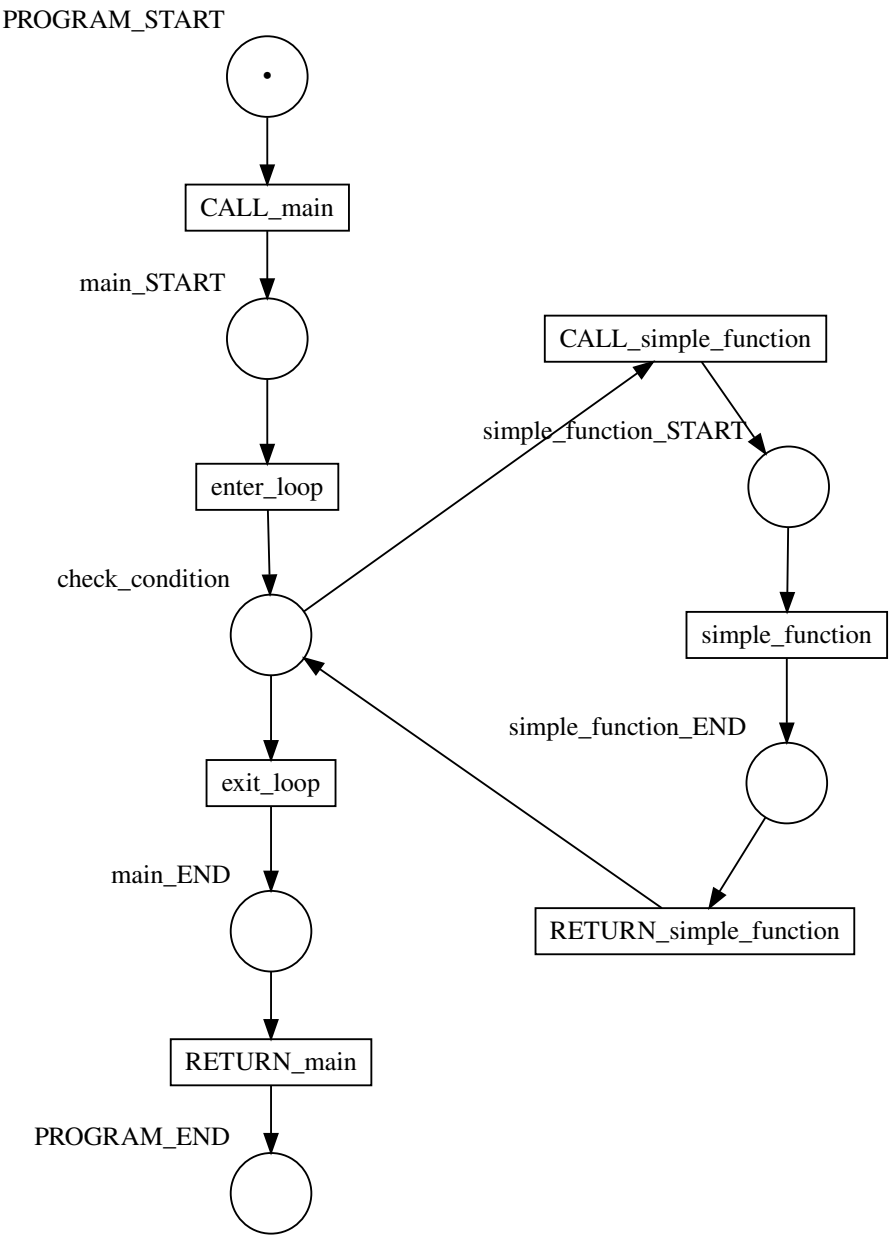

Figura 3.3: Una posible red de Petri para el código del Listado 3.4 aplicando el modelo de la Fig. 3.2.

`main_end_place`. El token en `main_end_place` aparecerá finalmente en `PROGRAM_END`, lo que indica una terminación normal del programa. Esto es técnicamente correcto ya que sabemos que el programa termina con éxito.

No obstante, existen problemas que conciernen al segundo token. El token en `check_flag` puede ser consumido por la transición `flag_is_false` o `flag_is_true`. Si es consumida por

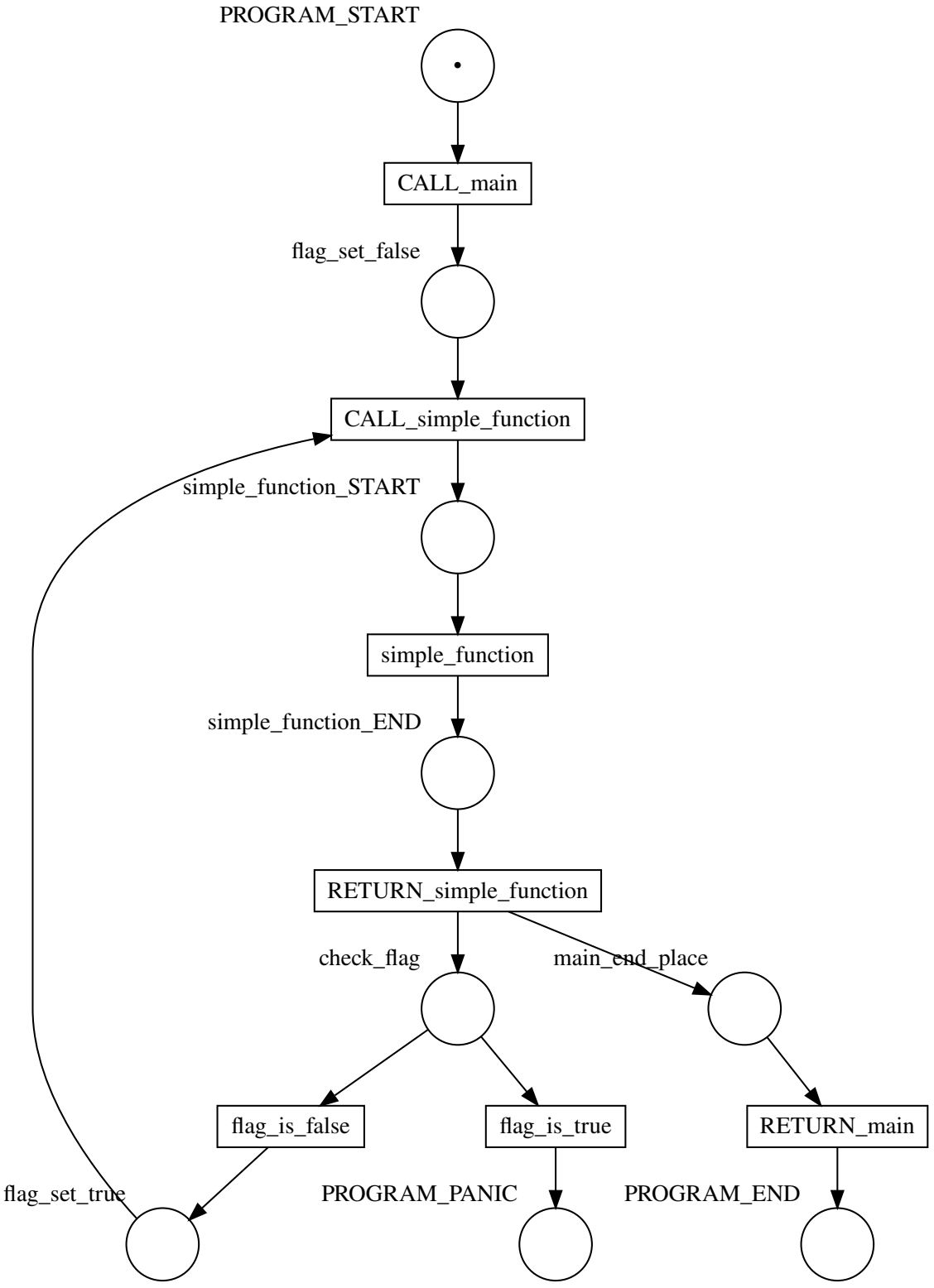

Figura 3.4: Una primera red de Petri (incorrecta) para el código del Listado 3.5.

esta última, se colocará un testigo en `PROGRAM_PANIC`, señalando una terminación errónea del programa. Esto es absurdo porque significa que el programa podría entrar en pánico pero también que *siempre* termina normalmente, como se ha visto en el párrafo anterior.

La situación empeora si seguimos el camino de disparar `flag_is_false`. En ese caso, el token desencadena otra llamada a la función, lo que en principio es correcto, pero nada impide que lo haga una y otra vez. La conclusión es que podría acumularse una cantidad infinita de tokens en `main_end_place` o `PROGRAM_END` en el caso de que, por pura casualidad, la transición `flag_is_true` no se dispare.

Está claro que debemos descartar este modelo y buscar una solución mejor. Una posibilidad es dividir la transición etiquetada `RETURN_simple_function` en dos transiciones separadas según el orden de llamada a la función, como se ilustra en la Fig. 3.5.

Desafortunadamente, este segundo intento viene acompañado de su propio conjunto de estados extraños. En primer lugar, ahora el programa puede salir después de llamar a la función una sola vez. Nada impide que la transición `RETURN_simple_function_2` se dispare primero. Esto equivale a decir que el flujo de ejecución salta de la línea 5 a la línea 11 en el Listado 3.5, lo que obviamente no es una propiedad presente en el código Rust original.

Por otro lado, persiste el problema del bucle infinito. La PN puede seguir disparándose indefinidamente mientras `flag_is_true` y `RETURN_simple_function_2` no se disparen. No hay ninguna garantía de que las transiciones se disparen en un orden específico. Como se vio en la Sec. 1.1.3, el disparo de las transiciones no es determinista.

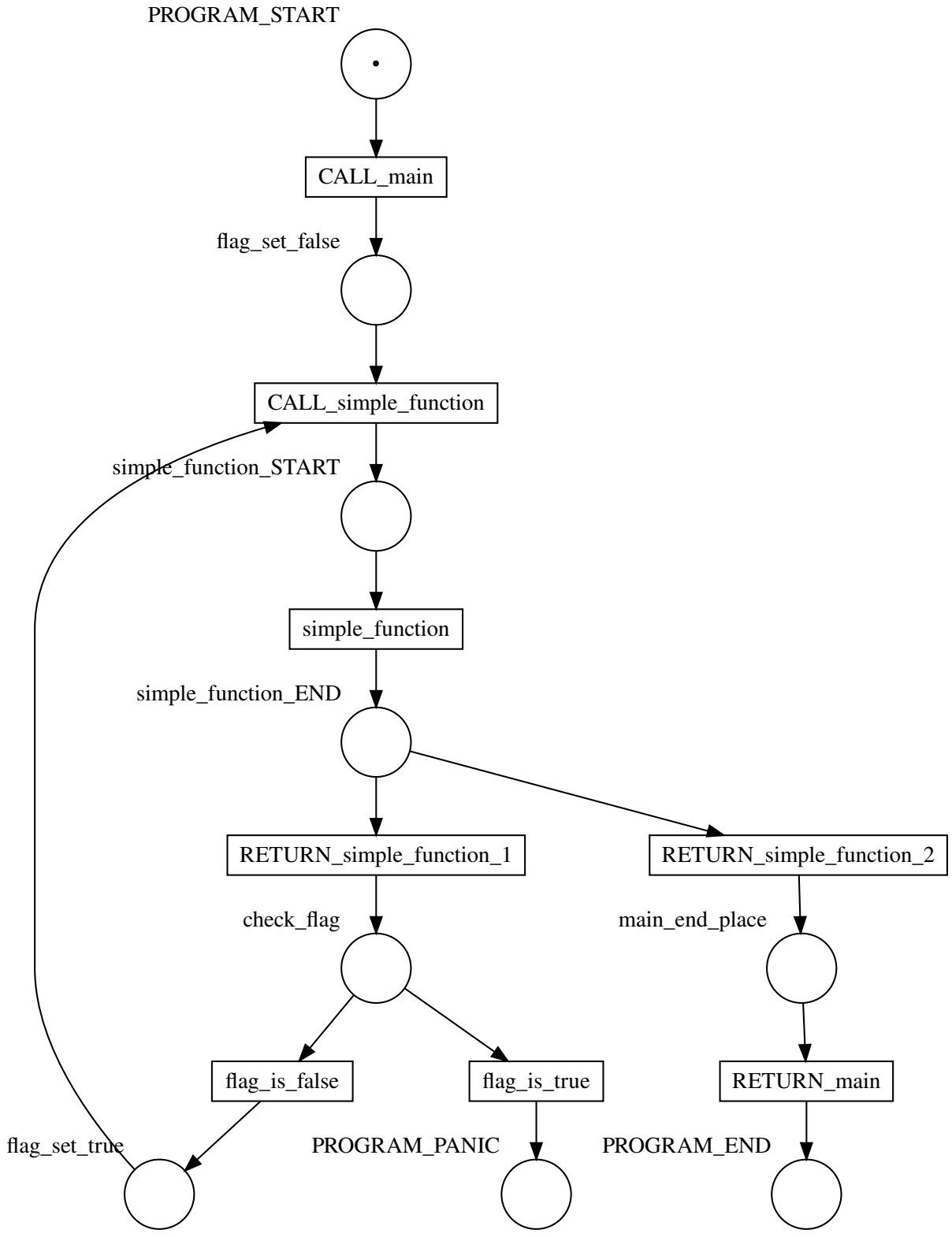

Figura 3.5: Una segunda red de Petri (también incorrecta) para el código del Listado 3.5.

3.5.3. Una solución viable

Una vez observadas las dificultades de modelar las llamadas de función, volvemos nuestra atención al otro enfoque para modelar las llamadas de función: La incrustación o *inlining* de la representación PN. Algunas de las lecciones aprendidas de la subsección anterior son:

- Crear un bucle en la red donde no lo hay en el programa original abre la puerta a secuencias infinitas de disparos de transición. Esto podría a su vez romper la propiedad de *safety* de la PN.
- Como el token simboliza el contador del programa, sólo debe haber un token en la PN en cualquier momento dado.
- El estado del programa puede cambiar entre llamadas a funciones. En consecuencia, deben modelarse estos estados por separado. Dicho de otro modo, el estado al llamar a una función la primera vez puede no ser el mismo que al llamar a la función por segunda vez.

La Fig. 3.6 presenta el enfoque de inlining implementado en la herramienta. La PN en ella es correcta. Se ajusta mejor a la estructura del código Rust. No contiene ningún bucle ni crea tokens adicionales al disparar transiciones, es decir, ninguna de las transiciones tiene dos salidas. Cabe mencionar que la PN resultante es una máquina de estados (Definición 8), tal y como se espera para un programa de un solo hilo. No es el caso de las Fig. 3.4 y 3.5.

Una ventaja significativa del enfoque inlining es que cada llamada a una función se identifica de forma inequívoca. Esto resulta útil a la hora de interpretar la salida del verificador de modelos o los mensajes de error durante la traducción de un programa determinado. El uso de un id incremental no negativo es arbitrario pero conveniente. Además, la precisión de la detección del bloqueo se incrementa porque ciertas clases de estados extraños, como los del PN mostrado en la sección anterior, no están presentes. Minimizar el número de falsos positivos desempeña un papel importante a la hora de considerar qué enfoque aplicar para una herramienta que pretende ser fácil de usar y de configurar.

Una desventaja mencionada anteriormente es que el tamaño de la red resultante es mayor. La penalización exacta en el número de lugares y transiciones adicionales depende de la frecuencia con la que se reutilizan las funciones por término medio en el código base. Es razonable suponer que las funciones se llaman desde varios lugares. Sin embargo, se pueden aplicar ciertas optimizaciones que pueden reducir considerablemente el tamaño de la red, compensando así el efecto de utilizar inlining. Estas optimizaciones se tratan en detalle en las Sec. 6.1 y 6.2.

Por último, un lector atento puede notar que el análisis de la PN de la Fig. 3.6 lleva a la conclusión de que el programa puede llamar a `panic!` y terminar abruptamente, lo que no coincide con la ejecución del programa Rust. Esto es correcto pero es una limitación de las redes de Petri de bajo nivel que no puede resolverse en el marco del modelo y va más allá del alcance de este trabajo. Sec. 6.6 explora las consecuencias de esta restricción y propone posibles soluciones.

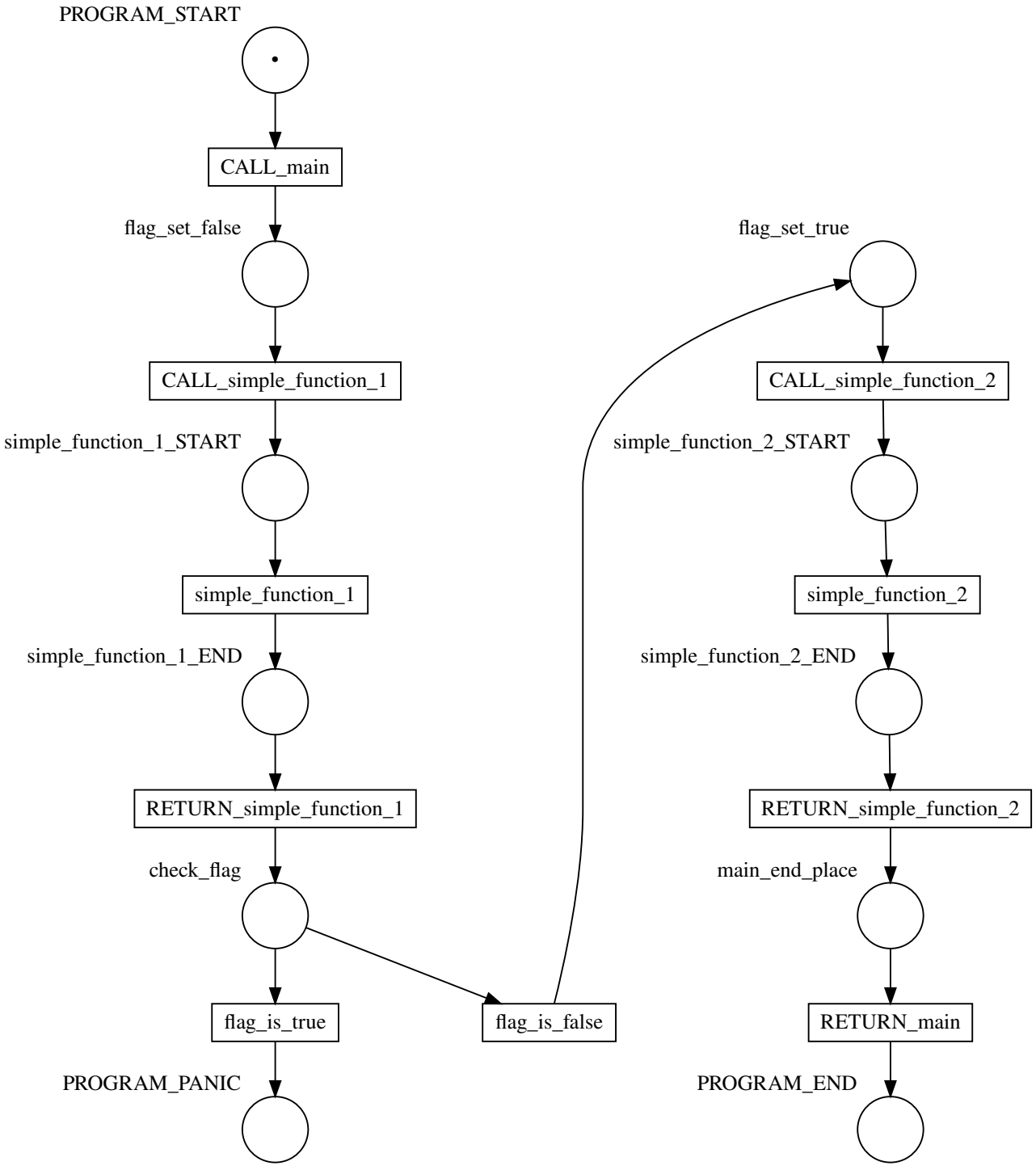

Figura 3.6: Una red de Petri correcta para el código del Listado 3.5 utilizando inlining.

Armados con nuevas ideas y conocimientos sobre las decisiones de diseño, ahora estamos en condiciones de describir completamente la implementación.

Capítulo 4

Implementación de la traducción

Este capítulo está dedicado a explorar los detalles de implementación de la herramienta de detección de deadlocks. Su propósito es proporcionar una visión de alto nivel del código y las estructuras de datos. También se examinan las decisiones de implementación más importantes tomadas a lo largo del proceso de desarrollo.

En las secciones sucesivas, describiremos los componentes centrales de la herramienta de detección de deadlocks, incluida la representación interna de la pila de llamadas, el modelo de memoria de funciones y la traducción de cada componente de una función MIR.

Más adelante, una parte importante de la discusión se dedica a explicar el soporte del multihilo y el modelado de las primitivas de sincronización como redes de Petri. Su implementación requirió cuidadosas consideraciones de diseño para garantizar la correctitud y la eficiencia.

La herramienta soporta actualmente las siguientes estructuras de la biblioteca estándar de Rust para sincronizar el acceso a los recursos compartidos y proporcionar comunicación entre hilos:

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

Aunque se cubren los detalles principales, este capítulo no pretende sustituir a la documentación en el código. La documentación del código en forma de comentarios, pruebas unitarias y pruebas de integración proporciona información exhaustiva sobre los detalles de bajo nivel y el uso de la herramienta. Como ya se ha indicado, el repositorio está disponible públicamente en 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. Consideraciones iniciales

4.1.1. Lugares básicos de un programa Rust

El modelo básico de red de Petri para un programa Rust generado por la herramienta puede verse en la Fig. 4.1. El lugar etiquetado `PROGRAM_START` contiene un token y representa el estado inicial del programa Rust. Este token se "moverá" de declaración en declaración y, por tanto, puede interpretarse como el contador de programa de la CPU.

Correspondientemente, el lugar etiquetado `PROGRAM_END` modela el estado final del programa después de la terminación normal del programa, es decir, al volver de la función `main`, independientemente del código de salida específico. En otras palabras, una función principal que devuelve un código de error debido a parámetros no válidos o a un error interno del programa se sigue considerando una terminación "normal" del programa. En otros casos, sin embargo, el programa puede no llegar nunca a este estado si `main` nunca retorna. Estas se conocen en Rust como "funciones divergentes"[6] y están soportadas por la herramienta.

Por último, el lugar etiquetado `PROGRAM_PANIC` modela la terminación anormal del programa, que ocurre cuando el programa llama a la macro `panic!`.

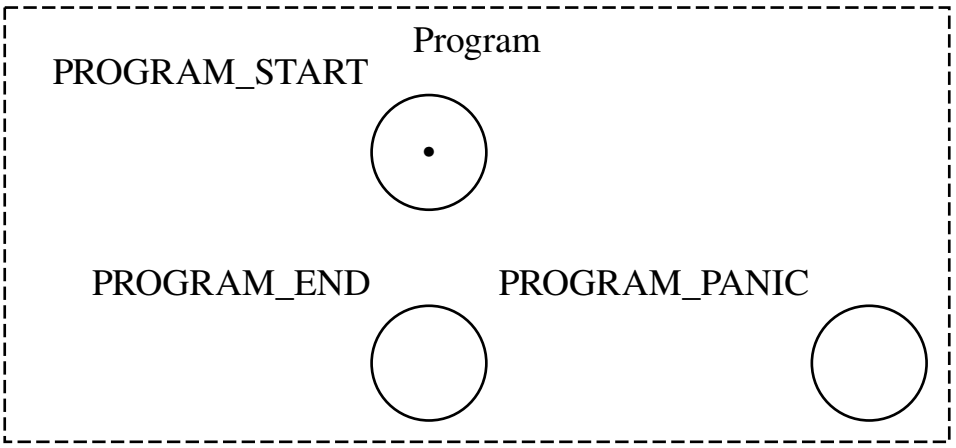

Figura 4.1: Lugares básicos en todo programa Rust.

Se pueden argumentar dos razones para considerar un lugar separado para el estado final de pánico. En primer lugar, es útil para la verificación formal distinguir el caso de pánico del caso de terminación normal. Un programa puede entrar en pánico al detectar una posible violación de sus garantías de seguridad de memoria. En la mayoría de las circunstancias, esta es una opción más inteligente que simplemente ignorar el error y continuar. Por esta razón estos programas no deben marcarse en principio como erróneos o defectuosos pero es aconsejable registrar el estado

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

final a efectos de solución de problemas y depuración. En segundo lugar, incluso si el código del usuario no recurre a `panic!` como mecanismo de gestión de errores, numerosas funciones de la biblioteca estándar de Rust pueden entrar en pánico en circunstancias extraordinarias, por ejemplo, debido a una falta de memoria (out-of-memory (OOM)) o a un error de hardware, o cuando el OS falla al asignar un nuevo hilo, mutex, etc. En consecuencia, es esencial capturar este eventual fallo en el modelo PN.

Hay un último punto sutil que es necesario abordar. El lugar de inicio del programa no es tan trivial como parece. Aunque la función principal `main` se percibe típicamente como la primera función que se ejecuta, en realidad no es así. En su lugar, los programas Rust tienen un *runtime* que se ejecuta antes de llamar a la función principal en el que se inicializan las características específicas del lenguaje y la memoria estática. Normalmente se oye hablar de lenguajes interpretados como Java o Python que tienen un tiempo de ejecución pero los lenguajes de bajo nivel como Rust o C también tienen un pequeño *runtime*. Simplemente es una capa más fina y menos sofisticada. Para los lectores interesados, recientemente se presentó en una conferencia sobre Rust un tour de las funciones antes de main [Levick, 2022].

Teniendo esto en cuenta, nos enfrentamos a la cuestión de si debemos incluir este *runtime* en la traducción de la red de Petri. Por un lado, el código del tiempo de ejecución forma parte efectivamente del binario ejecutado por la CPU. No obstante, es un código dependiente de la plataforma (el *runtime* es ligeramente diferente para cada OS) e independiente de la semántica del programa, es decir, del significado específico del programa que escribió el usuario. Dado que el usuario no tiene ninguna influencia en esta parte del binario, no se le pueden atribuir problemas de sincronización. Como tal, este código no añade valor a la traducción y puede abstraerse de forma segura, reduciendo en el proceso el tamaño de la PN. En conclusión, la decisión es omitir el código en tiempo de ejecución; la traducción comienza en la función `main`.

4.1.2. Pasaje de argumentos e introducción de la *query*

La herramienta está diseñada en torno a una sencilla interfaz de línea de comandos (command-line interface (CLI)). Tras analizar los argumentos de la línea de comandos utilizando la conocida biblioteca `clap`[7], el programa introduce una consulta al compilador rustc para iniciar el proceso de traducción. La mayor parte del trabajo a partir de ese momento está coordinado por la estructura de tipo `Translator`[8].

El sistema de consulta (*query*) se describió brevemente en la Sec. 3.2. En la documentación[9][10] se proporcionan dos ejemplos del uso de este mecanismo. Han demostrado ser extremadamente útiles como punto de partida puesto que proporcionan un excelente y breve ejemplo funcional

[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

de cómo interactuar con *rustc*. En términos más sencillos, son el "¡Hola, mundo!"del trabajo codo con codo con el compilador de Rust.

4.1.3. Requisitos de compilación

Como se mencionó brevemente en la Sec. 3.2.2, la herramienta debe compilarse con la versión nightly de *rustc* para acceder a sus crates y módulos internos. La sección decisiva en el archivo *lib.rs*[11] se representa en el Listado 4.1.

```
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: Extracto del archivo *lib.rs* que muestra cómo utilizar las funciones internas de *rustc*.

`rustc_private` es una *feature flag* que controla el acceso a los crates privados del compilador. Estos crates no se instalan por defecto al instalar la cadena de herramientas de Rust utilizando *rustup*[12]. Por esta razón, es necesario instalar los componentes adicionales *rustc-dev*, *rust-src* y *llvm-tools-preview*. El propósito de cada componente se detalla en [Rust Project, 2023d]. En

[11] https://github.com/hlisdero/cargo-check-deadlock/blob/main/src/lib.rs
[12] https://rustup.rs/

el `README`[13] del repositorio se hallan instrucciones fáciles de seguir para instalar el ambiente de desarrollo.

Que este autor sepa, no existe un método alternativo para acceder a las partes internas del compilador de Rust. Herramientas como Clippy[14] y Kani [15] o kernels como Redox[16] y RustyHermit[17] también utilizan este mecanismo.

4.2. Llamadas a funciones

4.2.1. La pila de llamadas (*The call stack*)

Un programa en Rust se compone, como en otros lenguajes de programación, de funciones. El programa comienza (salvo las advertencias vistas en la Sec. 4.1.1) con una llamada a la función `main` que luego puede llamar a otras funciones. Cabe destacar que las llamadas a funciones pueden situarse en cualquier punto del código. Una función puede ser llamada desde otra función o incluso desde dentro de sí misma, dando lugar a llamadas recursivas.

Las llamadas a funciones se almacenan en memoria en una estructura de datos llamada *(*pila de llamadas) (*call stack*). Cuando se llama a una función en Rust, ésta se introduce en la pila de llamadas, creando un nuevo registro de activación (*stack frame*). Un *stack frame* contiene información importante como las variables locales de la función, los argumentos y la dirección de retorno que indica dónde debe reanudarse el programa una vez que la función finaliza su ejecución.

La pila de llamadas funciona según el principio de último en entrar, primero en salir (last in, first out (LIFO)). A medida que se llaman funciones, cada nuevo registro de activación se coloca encima del anterior. Esto permite que el programa ejecute primero la función llamada más recientemente. Una vez que una función completa su ejecución, se retira de la pila y el programa continúa desde el punto en que lo dejó la función anterior.

Por consiguiente, la pila de llamadas desempeña un papel esencial en la gestión de las llamadas y retornos de funciones porque realiza un seguimiento del flujo de llamadas a funciones y mantiene la información necesaria para que el programa vuelva al punto de ejecución correcto después de que una función complete su tarea.

Por las mismas razones, reflejar la pila de llamadas en el traductor es el enfoque más adecuado para el seguimiento de las llamadas a funciones que deben traducirse, puesto que se alinea con el flujo lógico de la ejecución del programa. A medida que se traducen las funciones, se agregan

[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

y se sacan de la pila de llamadas del `Translator`, imitando el orden en que se llaman en tiempo de ejecución. Esto nos permite manejar las invocaciones de funciones anidadas y seguir el flujo de control de una función a otra durante el proceso de traducción.

4.2.2. Funciones MIR

En la implementación, el `Translator` tiene un stack que soporta las operaciones habituales `push`, `pop` y `peek`. Esta pila almacena estructuras de tipo `MirFunction`[18]. Más adelante veremos que no todas las funciones se traducen como funciones MIR, dado que no todas las funciones tienen una representación en MIR y, en otros casos, es conveniente manejarlas de otro modo. No obstante, las funciones MIR son el "caso común" en el proceso de traducción, el caso por defecto para la mayoría de las funciones definidas por el usuario.

La interfaz disponible proporcionada por *rustc* permite consultar (*query*) el cuerpo MIR de una sola función a la vez, lo que puede hacerse utilizando el `optimized_mir`[19]. Esto implica que no es posible obtener inicialmente el MIR de todo el programa y que el traductor debe obtener el MIR de cada función a medida que las descubre en el código. Pero, ¿cómo identificar cada función? Se sabe por experiencia que las funciones de módulos distintos pueden tener el mismo nombre, lo que hace que el nombre no sea adecuado como identificador. Por suerte, este problema ya está resuelto en el compilador. Las funciones se identifican unívocamente mediante el tipo del compilador `rustc_hir::def_id::DefId`[20]. Este ID es válido para el crate que se está compilando en ese momento y ya está presente en el HIR. El algoritmo de alto nivel puede describirse como sigue.

Cuando comienza la traducción:

1. Consultar el id del punto de entrada del programa (la función principal, `main`).
2. Crear una `MirFunction` con la información necesaria.
3. Agregarlo a la pila.
4. Si es necesario, modificar el contenido de la función MIR mediante `peek`.
5. Traducir el elemento superior de la pila de llamadas.
6. Cuando `main` termine, eliminarlo (`pop`) de la pila de llamadas.

Cuando se descubre un terminator de tipo "call" (véase la Sec. 3.4.1):

1. Consultar el id de la función llamada.

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

[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

2. Crear una `MirFunction` con la información necesaria.
3. Agregarlo a la pila.
4. Si es necesario, modificar el contenido de la función MIR mediante `peek`.
5. Traducir el elemento superior de la pila de llamadas.
6. Cuando la función termine, eliminarlo (`pop`) de la pila de llamadas.

Como se ha visto, el enfoque es consistente para cada función MIR y, en consecuencia, es más fácil de implementar.

El uso de un call stack en el proceso de traducción permite el cambio de contexto (*context switching*) entre funciones MIR y facilita la posibilidad de volver al bloque básico específico desde el que se llamó a una función. Esto posibilita que la traducción del programa se realice función por función de forma lineal, asegurando que se mantienen la estructura y el orden del programa original.

Sin embargo, el enfoque de la pila de llamadas conlleva ciertas limitaciones. En primer lugar, si la misma función se llama varias veces dentro del programa, se traducirá asimismo varias veces. Esto está relacionado con la estrategia de inlining elaborada en la Sec. 3.5. Aunque esto puede potencialmente mitigarse mediante el uso de algún tipo de caché, está fuera del alcance de esta tesis. Esta optimización se discutirá en la Sec. 6.3.

La implicación más grave de utilizar el enfoque de pila de llamadas es la incapacidad de manejar funciones recursivas. Cuando se procesa una función recursiva durante la traducción, el proceso queda atrapado en un bucle sin fin en el que la pila crece indefinidamente a medida que se introducen en ella nuevos stack frames, lo que provoca un desbordamiento de la pila (*stack overflow*) y la consiguiente interrupción del proceso de traducción. Este problema también se aborda en la Sec. 6.4. Por ahora, es necesario aceptar la limitación de que las funciones recursivas infinitas no pueden traducirse utilizando este marco teórico.

4.2.3. Funciones foráneas y funciones de la biblioteca estándar

En Rust, el compilador incluye por defecto la biblioteca estándar en todos los binarios compilados, enlazándola de forma estática. Para anular este comportamiento, se utiliza el atributo a nivel de crate `#![no_std]` para indicar que el crate enlazará con el core-crate en lugar de con el std-crate. Consulte [Rust on Embedded Devices Working Group, 2023] para más detalles.

Esto significa que la funcionalidad de la biblioteca estándar se convierte en parte integrante del ejecutable resultante. Las llamadas a funciones de la biblioteca estándar aparecen en varios contextos en el código Rust, como cuando se accede a argumentos de la línea de comandos, se invocan iteradores, se utilizan traits como `std::clone::Clone`, `std::deref::Deref::deref`, o se emplean tipos de la biblioteca estándar como `std::result::Result` o `std::option::Option`.

Dada la prevalencia de estas llamadas a funciones en todos los programas Rust, resulta esencial manejarlas por separado en el proceso de traducción. Es evidente que estas funciones de biblioteca estándar, debido a su propósito, no pueden conducir a un deadlock. Por lo tanto, es más práctico tratarlas como cajas negras dentro del proceso de traducción, obviando la necesidad de traducir su MIR. Este enfoque es indispensable para evitar generar una red de Petri excesivamente grande y enrevesada que dificulte la legibilidad y la comprensión.

El esfuerzo de traducción se centra principalmente en el código de usuario, concretamente en las funciones que los desarrolladores escriben para implementar sus funcionalidades deseadas. Al dirigir la atención al código de usuario y excluir la traducción de las funciones de la biblioteca estándar, la red de Petri resulta más manejable, lo que facilita el análisis y la verificación de posibles deadlock dentro de la base de código del usuario. Las llamadas a la biblioteca estándar constituyen, en otras palabras, la "frontera" o el "límite" de la traducción, el punto en el que dejamos de traducir el MIR con precisión y nos basamos en cambio en un modelo simplificado.

Modelo de red de Petri para una función con bloque de limpieza

El modelo presentado en la Fig. 3.2 es la primera aproximación. Sin embargo, existe un detalle de implementación que requiere especial atención. Diversas funciones de la biblioteca estándar contienen no sólo un lugar final ("target block", en la jerga de *rustc*) sino también un lugar de limpieza ("cleanup block"). Esta segunda vía de ejecución se toma cuando la función entra explícitamente en pánico o, más genéricamente, no consigue su objetivo por cualquier motivo. En este caso, el flujo de control continúa hacia un bloque básico diferente donde se liberan las variables y finalmente el programa termina con un código de error de pánico. Dicho de otro modo, el desenrollado de la pila comienza en cuanto una función encuentra un fallo no recuperable.

Teniendo en cuenta que el traductor no puede saber si esta situación anómala podría provocar un deadlock más adelante en el proceso de traducción, es indispensable traducir esta ruta de ejecución alternativa siempre que sea posible. Sólo en contadas excepciones, todas ellas relacionadas con las primitivas de sincronización y tratadas en las secciones respectivas, se ignora explícitamente este bloque de limpieza. El modelo completo para una llamada a función abreviada con un bloque de limpieza puede verse en la Fig. 4.2.

Funciones traducidas con el modelo de red de Petri abreviado

Tras haber discutido la exclusión de las funciones de biblioteca estándar del proceso de traducción, ahora nos centraremos en las funciones que sí requieren traducción utilizando el modelo que hemos presentado antes. Sorprendentemente, incluyen un número considerable de funciones.

- Funciones que forman parte de la biblioteca estándar (std-crate[21]), salvo por la función

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

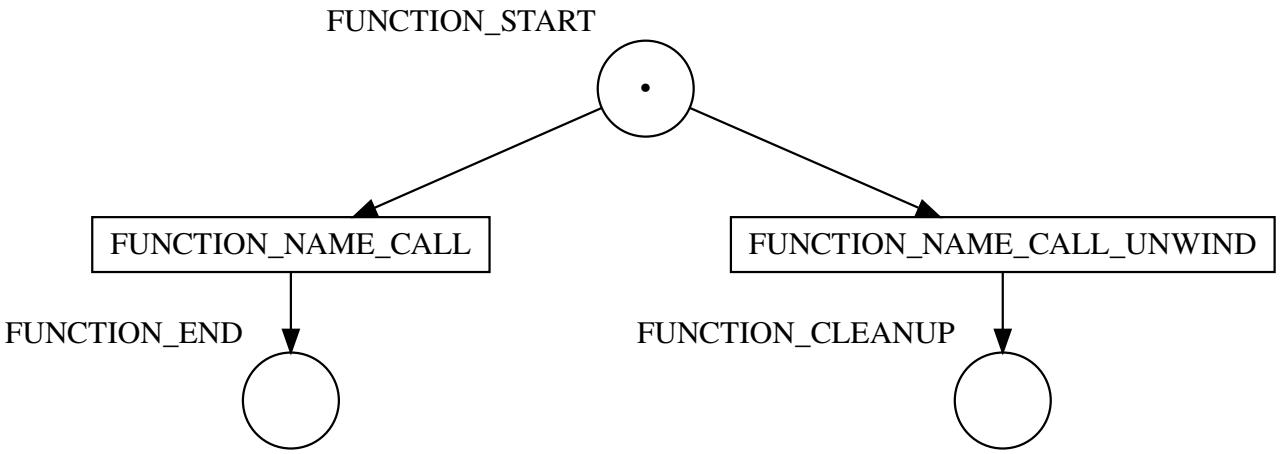

Figura 4.2: El modelo de red de Petri para una función con un bloque de limpieza.

`std::sync::Condvar::wait` detallada en la Sec. 4.8.3.

- Funciones que forman parte de la biblioteca core (core-crate[22]).
- Funciones en el crate alloc: the core allocation and collections library[23].
- Funciones sin representación MIR. Esto puede comprobarse con el método `is_mir_available`[24].
- Funciones externas, es decir, importadas mediante `extern { ... }`. Esto puede comprobarse con el método `is_foreign_item`[25].

En el futuro, las llamadas a funciones en dependencias, es decir, en otros crates, también deberán tratarse de este modo. En conclusión, el caso por defecto para las funciones que *no* fueron definidas por el usuario es tratarlas como una función foránea y utilizar un modelo de red de Petri abreviado para traducirlas.

4.2.4. Funciones divergentes

Las funciones divergentes son un caso especial relativamente fácil de soportar. Se trata simplemente de una función que nunca vuelve a la función que la llamó. Ejemplos de ello son un *wrapper* alrededor de un bucle `while` infinito, una función que sale del proceso o una función que inicia un OS. Basta con conectar el lugar de inicio de la función a una transición de sumidero (Definición 7) como se ve en la Fig. 4.3.

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

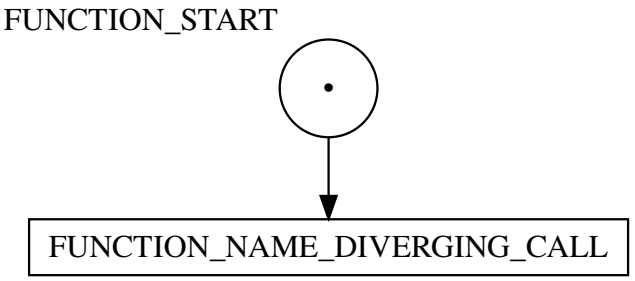

Figura 4.3: El modelo de red de Petri para una función divergente (una función que no retorna).

Nótese que este caso especial no constituye un deadlock y no debe tratarse como tal. Un bucle infinito, es decir, un "busy wait", es en su naturaleza inherente distinto de la espera infinita que caracteriza un deadlock como se vio en la Sec. 1.4.1. En otras palabras, detectar bucles infinitos se acerca más al problema de detectar livelocks que están fuera del alcance de esta tesis. Además, el traductor no puede saber de antemano si la llamada divergente es benigna como una llamada a `std::process::exit` o una llamada a algún tipo de función especial y cuidadosamente diseñada para bloquear el programa.

En el modelo PN actual, el token se consume y la red queda en un estado final sin tokens en los lugares `PROGRAM_END` o `PROGRAM_PANIC` mostrados en la Fig. 4.1. En consecuencia, el verificador del modelo es capaz de distinguir este estado final de los demás casos y concluir que se ha llamado a una función divergente.

4.2.5. Llamadas explícitas de pánico

La macro `panic!` puede verse como un caso especial de función divergente en el que la transición que representa la llamada a la función está conectada al lugar etiquetado `PROGRAM_PANIC` descrito en la Sec. 4.1.1. El traductor detecta una llamada explícita de pánico que es una de las siguientes funciones:

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

La documentación[26] detalla por qué se define el pánico en el core-crate y en el std-crate y cómo se implementa.

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

Véase el Listado 4.2 para un programa simple que entra en pánico. El modelo de red de Petri correspondiente se representa en la Fig. 4.4. Este es uno de los ejemplos ilustrativos incluidos en el repositorio.

```
fn main() {
    panic!();
}
```

Listing 4.2: Un programa sencillo en Rust que llama `panic!`.

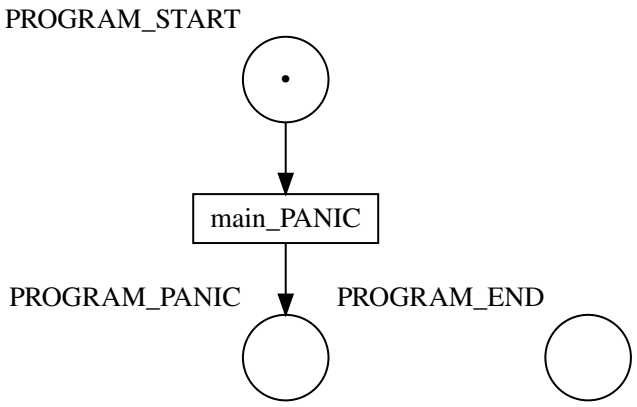

Figura 4.4: El modelo de red de Petri para el Listado 4.2.

4.3. MIR visitor

Esta sección está dedicada a un componente fundamental que sirve de columna vertebral del traductor: el trait MIR Visitor[27]. Este trait facilita la navegación por la MIR del código fuente de Rust. En otras palabras, actúa como el pegamento que une a la perfección los distintos componentes del traductor.

El trait MIR Visitor desempeña un papel fundamental en el proceso de traducción al proporcionar un enfoque estructurado para recorrer y analizar el MIR. Ofrece un conjunto de métodos que pueden implementarse para realizar acciones específicas en distintos puntos durante el recorrido. Implementando este trait, el traductor adquiere la capacidad de explorar sistemáticamente el MIR y extraer la información necesaria para generar la red de Petri correspondiente.

Los métodos implementados dentro del traot `MIRVisitor` sirven como puntos de entrada para manejar los diferentes elementos encontrados durante el recorrido. Estos métodos permiten el procesamiento personalizado de construcciones MIR específicas, por ejemplo, bloques básicos,

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

sentencias, terminadores, asignaciones, constantes, etc. Al definir el comportamiento adecuado para cada método, el traductor puede extraer eficazmente los datos relevantes y tomar decisiones correctas en función de los elementos MIR encontrados.

No es necesario implementar todos los métodos posibles. Si no se definen, los métodos de `MIRVisitor` simplemente llaman al método `super` correspondiente y continúan el recorrido. Por ejemplo, `visit_statement` llama a `super_statement` si no existe una implementación personalizada. En el caso del traductor, los métodos implementados son:

- `visit_basic_block_data` para realizar un seguimiento del bloque básico que se está traduciendo en ese momento.
- `visit_assign` para realizar un seguimiento de las asignaciones de variables de sincronización (mutexes, mutex guards, join handles y condition variables)
- `visit_terminator` para procesar la sentencia terminadora de cada bloque básico, es decir, para conectar los bloques básicos.

Para empezar a visitar el MIR, debe utilizarse el método `visit_body`. El Listado 4.3 muestra la función correspondiente en el traductor.

```
1  /// Main translation loop.
2  /// Translates the function from the top of the call stack.
3  /// Inside the MIR Visitor, when a call to another function happens, this method will be
   ↪  called again
4  /// to jump to the new function. Eventually a "leaf function" will be reached, the functions
   ↪  will exit and the
5  /// elements from the stack will be popped in order.
6  fn translate_top_call_stack(&mut self) {
7      let function = self.call_stack.peek();
8      // Obtain the MIR representation of the function.
9      let body = self.tcx.optimized_mir(function.def_id);
10     // Visit the MIR body of the function using the methods of
       ↪  `rustc_middle::mir::visit::Visitor`.
11     //
       ↪  <https://doc.rust-lang.org/stable/nightly-rustc/rustc_middle/mir/visit/trait.Visitor.html>
12     self.visit_body(body);
13     // Finished processing this function.
14     self.call_stack.pop();
15 }
```

Listing 4.3: El método del `Translator` que inicia el recorrido del MIR.

En conclusión, el trait `MIRVisitor` simplifica notablemente la traducción porque no es necesario implementar un mecanismo propio de recorrido gracias a las interfaces de compilador proporcio-

nadas. Esto también hace que el traductor sea más robusto y resistente a los cambios en *rustc*. Si la representación de cadenas de la MIR cambia, el traductor no se ve afectado. Mientras las interfaces internas para acceder al MIR sigan siendo las mismas, el traductor podrá recorrer el MIR semánticamente y no en función de cómo se imprime al usuario.

Como última observación, existen traits similares para otras representaciones intermedias:

- 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

Varios componentes del compilador implementan estos traits para navegar por las repre- sentaciones intermedias. Por decirlo de forma aproximada, son análogos a los iteradores para colecciones.

4.4. MIR function

En la siguiente sección, profundizaremos en el proceso de traducción de una función MIR. Esta sección pretende ofrecer una comprensión exhaustiva de las técnicas de traducción aplicadas a elementos específicos de la MIR, a saber, los basic blocks (BB), las sentencias y los *terminator statements*. Estos componentes se introdujeron anteriormente en la Sec. 3.4.1.

La implementación en el repositorio lleva el nombre `MirFunction`[28]. Este tipo almacena el lugar de inicio y el lugar final de la función. Estos deben suministrarse a la función MIR porque también representan dónde tuvo lugar la llamada a la función y a dónde debe volver. El lugar final es, en términos más sencillos, el lugar de retorno en la red de Petri. La Fig. 3.2 ilustra esto.

El lugar de inicio de la función se solapa con el lugar que modela el primer bloque básico de la función. Esto se ajusta más a la MIR, ya que el código sólo vive dentro de los bloques básicos, por lo que la llamada a la función comienza en el primer bloque básico (BB0).

La función `MirFunction` también almacena el ID que la identifica. Esto es necesario para realizar llamadas a funciones desde esta función. Asimismo, la función requiere un nombre que es diferente para cada llamada a la función, por lo que recibe un nombre con un índice añadido, lo que lo hace único en toda la red de Petri.

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

A continuación explicaremos cómo se expresa cada componente en el lenguaje de las redes de Petri. Mediante una exploración detallada de las técnicas de traducción empleadas para los bloques básicos, las sentencias y los terminadores, desarrollaremos un modelo formal que capture con precisión el comportamiento de una función MIR para la detección de bloqueos.

4.4.1. Bloques básicos

Un aspecto del proceso de traducción consiste en transformar los bloques básicos en redes de Petri, que sirven como bloque de construcción fundamental para modelar el flujo de control dentro de la función MIR. Como se ve en la Fig. 3.1, un bloque básico en MIR actúa como un contenedor que alberga una secuencia de cero o más sentencias (*statements*), así como una sentencia de terminación obligatoria.

Como nodos de un grafo, la principal propiedad de los bloques básicos es su capacidad para dirigir el flujo de control dentro de un programa. Cada bloque básico puede tener uno o más bloques básicos que apunten a él, indicando los posibles caminos desde los que el flujo de control puede alcanzarlo. Del mismo modo, un bloque básico puede apuntar a otro u otros bloques básicos, señalando los posibles caminos que puede tomar el flujo de control tras ejecutar el bloque básico actual. Cabe mencionar que los bloques básicos aislados sin conexiones no tienen sentido, debido a que nunca se ejecutarían, es decir, son código muerto.

Este comportamiento de bifurcación permite un flujo de control dinámico dentro del programa, ya que varios bloques básicos pueden continuar el flujo de control hacia el mismo bloque básico de destino (por ejemplo, hacia un bloque que realice tareas de limpieza). A la inversa, un bloque básico puede bifurcarse y determinar el siguiente bloque básico basándose en condiciones específicas o en la lógica del programa, por ejemplo, en un `if`, `while`, `match` u otras estructuras de control. Esta versatilidad en el flujo de control proporciona la base para modelar el comportamiento de programas complejos.

El modelo de red de Petri utilizado en la aplicación se basa en un único lugar para modelar cada BB. Podemos abstraernos del funcionamiento interno de los BB y trabajar con un único lugar. La razón de ello es que las conexiones con otros BB dependen únicamente de los terminadores y los *statements* no se modelan en absoluto, como veremos en breve. Por otra parte, la implementación[29] mantiene un registro del nombre de la función a la que pertenece el BB y del número de BB para generar etiquetas únicas.

4.4.2. Statements

Los statements MIR *no* se incorporan intencionalmente al modelo de red de Petri. Teniendo en cuenta que las razones para ello y los beneficios pueden no ser evidentes de inmediato,

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

proporcionaremos una explicación detallada de esta decisión de implementación.

El enfoque que se aplicó anteriormente sí incluía el modelado de statements. Se basaba en el enfoque visto en [Meyer, 2020]. Sin embargo, se observó que esto conducía a la creación de una larga cadena de lugares y transiciones que no contribuía significativamente a la detección de deadlocks o señales perdidas. Adicionalmente inflaba innecesariamente el tamaño de la representación en red de Petri, dificultando su depuración y comprensión. En consecuencia, este enfoque se revisó posteriormente y se eliminó en un commit posterior[30].

En todos los programas probados hasta ahora, los statements no realizaban ninguna acción que justificara su adición a la red de Petri. Al contrario, las redes que incluían los statements eran más grandes y más difíciles de leer. Para facilitar el uso y la adopción de la herramienta, es crucial optimizar la red de Petri para el propósito de la herramienta. Por lo tanto, la decisión fue eliminar todo el código relacionado con el modelado de las declaraciones y corregir los tests para que se ajustaran al nuevo resultado esperado.

La alternativa era desactivar las sentencias con una compile flag, pero eso complicaría las pruebas y, dado que de todos modos no hay ningún caso de uso para modelar las sentencias MIR, se descartó esta opción.

A título ilustrativo, podemos remitirnos a la Fig. 4.5 que muestra una comparación entre el modelo antiguo y el nuevo. Las diferencias entre estas dos representaciones son evidentes, destacando la eliminación de declaraciones del modelo y la consiguiente simplificación conseguida en la red de Petri final.

4.4.3. Terminators

Como se vio en la Sec. 3.4.1, existen diferentes clases de sentencias terminator. La documentación del enum `TerminatorKind`[31] enumera, en el momento de redactar este documento, 14 variantes diferentes. Se requiere que la implementación soporte la mayoría de ellas, visto que se manifiestan tarde o temprano en los programas de prueba incluidos en el repositorio y su traducción influye directamente en las conexiones entre los bloques básicos. Los restantes terminadores que no están implementados no están presentes cuando se consulta el `optimized_mir`, es decir, sólo se utilizan en pasadas previas del compilador.

La implementación de la MIR Visitor[32] incluye el método `visit_terminator` como se ha visto antes. Aquí es donde se crean las aristas que conectan un BB con otro. En los párrafos siguientes se discuten los detalles de alto nivel de cada handler. Se omiten algunos detalles de implementación ya que no afectan a la red de Petri.

[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

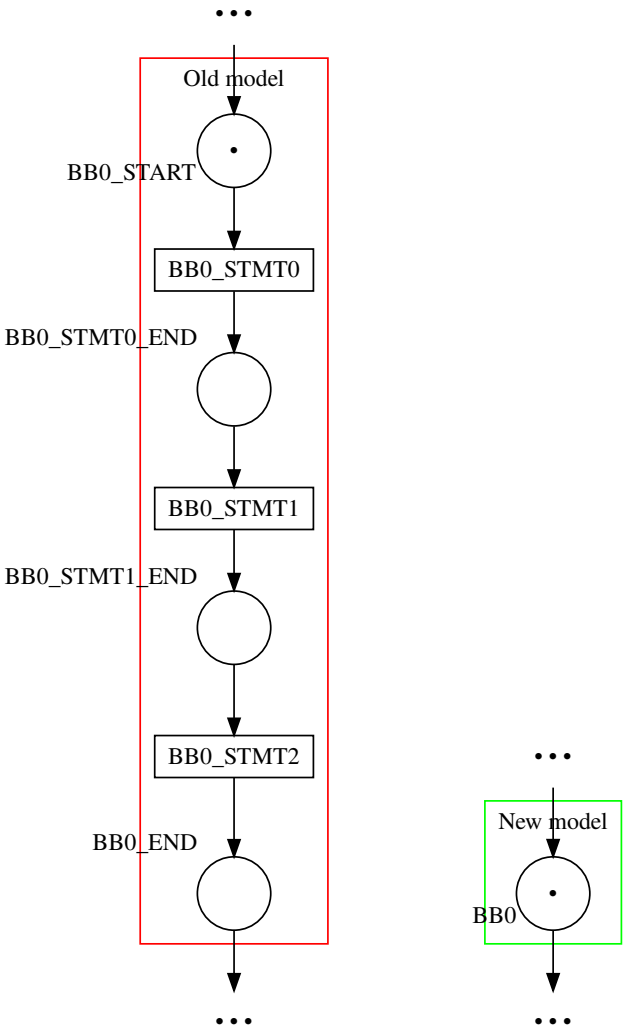

Figura 4.5: Comparación lado a lado de dos posibilidades para modelar los MIR statements.

Goto

Se trata de un tipo de terminación elemental. El lugar de finalización del BB actualmente activo se conecta con el lugar de inicio del BB objetivo a través de una nueva transición con una etiqueta adecuada.

SwitchInt

Este tipo de terminador viene con una colección de bloques básicos objetivo. Para cada BB objetivo, conectamos el lugar final del BB actualmente activo con el lugar inicial del BB objetivo

a través de una nueva transición con una etiqueta adecuada. Esto crea un conflicto tal y como se define en la Sec. 1.1.3.

La etiqueta también debe contener algún tipo de identificador único del bloque desde el que se inicia el salto. Se trata de una precondición para traducir correctamente varios bloques básicos con un `SwitchInt` que salte al mismo bloque.

Resume o Terminate

Se trata de terminadores que modelan respectivamente un desenrollado de la pila (*stack unwinding*) y el aborto inmediato del programa. Ambos se tratan de la misma manera: Conectando el lugar de finalización del BB actualmente activo con el lugar `PROGRAM_PANIC` visto en la Fig. 4.1.

Return

Este es el terminador que provoca el retorno de la función MIR. Aquí se utiliza el lugar final de la función. El lugar de finalización del BB actualmente activo se conecta a él.

Unreachable

Se trata de un caso borde que aparece en algunas `match`, `while` loops, u otras estructuras de control. La documentación lo indica: *Indicates a terminator that can never be reached.* Para tratar este caso, se ha optado por conectar el lugar de finalización del BB activo en ese momento con el lugar `PROGRAM_END` que se ve en la Fig. 4.1. Léase los comentarios en el repositorio para obtener más detalles.

Drop

El trait `std::ops::Drop` se utiliza para especificar el código que debe ejecutarse cuando el tipo sale de *scope* [Klabnik and Nichols, 2023, Chap. 15.3]. Es equivalente al concepto de destructores que se encuentra en otros lenguajes de programación.

El terminador de `Drop` se comporta como una llamada de función con una transición de limpieza. Por lo tanto, aplicamos el modelo mostrado en la Fig. 4.2 con etiquetas de transición modificadas.

Aquí también se produce una comprobación importante, a saber, la comprobación de si se está haciendo `Drop` de una *mutex guard*. Si es así, el mutex correspondiente debe desbloquearse como parte del disparo de la transición. Los detalles precisos se explican en la Sec. 4.7.3.

Call

Este es el tipo de terminador para ejecutar llamadas a funciones. La presencia de un bloque de limpieza y la `UnwindAction`[33] particular así como el nombre y el tipo de la función se analizan para manejarla según la estrategia elaborada en la Sec. 4.2.

Nótese que `UnwindAction` es una refactorización de *rustc* que se introdujo el 7 de abril de 2023. Es un buen ejemplo de una regresión que requirió cambios significativos para adaptarse. Se remite al lector interesado al commit[34] correspondiente.

Assert

Este tipo de terminador está relacionado con la macro `assert!()`[35] y las comprobaciones de desbordamiento (*overflow*) por defecto que *rustc* incorpora al realizar operaciones aritméticas.

La implementación no modela la condición para el assert. Simplemente conecta el lugar final del BB actualmente activo con el lugar inicial del BB objetivo a través de una nueva transición con una etiqueta adecuada.

En algunos casos, también hay un bloque de limpieza. Para ello, se necesita una segunda transición, de forma análoga al caso del `Drop`.

4.5. Memoria de las funciones

A continuación procederemos a explorar en detalle las características de memoria de la función MIR. Es importante reconocer que la necesidad de registrar los valores que se asignan entre lugares de memoria en la MIR surge de los requisitos de la detección de bloqueos y señales perdidas. En términos más sencillos, nos vemos obligados a modelar la memoria únicamente porque es necesario realizar un seguimiento de las variables de sincronización soportadas por el proceso de traducción.

El traductor debe realizar un seguimiento de las variables de los siguientes tipos:

- Mutexes (`std::sync::Mutex`).
- Mutex guards (`std::sync::MutexGuard`).
- Join handles (`std::thread::JoinHandle`).

[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

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

- Condition variables (`std::sync::Condvar`).
- Agregados (*Aggregates*), es decir, contenedores/wrappers como `std::sync::Arc` o tipos que contienen varios valores como tuplas o un tipo estructurado (`struct`).

Antes de llamar a los métodos de estos tipos de variables de sincronización, se crean referencias inmutables o mutables a la ubicación de memoria original. El traductor debe saber de algún modo qué variable de sincronización específica está detrás de una referencia determinada. Conocer el tipo de la ubicación de memoria *no* es suficiente, el *valor* debe ser fácilmente accesible para que el traductor modifique el modelo de red de Petri de la variable de sincronización específica.

4.5.1. Un ejemplo guiado para introducir los desafíos

Para ilustrar la situación descrita anteriormente, considere el programa Rust mostrado en el Listado 4.4. Se trata de nuevo de uno de los programas de ejemplo que se encuentran en el repositorio. Como debería ser evidente para el lector, este programa se bloquea cuando se ejecuta. La razón es que la función `std::sync::Mutex::lock` está siendo invocada dos veces sobre el mismo mutex. Para detectar este deadlock, el traductor debe ser capaz, como mínimo, de identificar que la invocación de `lock` tiene lugar sobre el mismo mutex.

```
1 fn main() {
2   let data = std::sync::Mutex::new(0);
3   let _d1 = data.lock();
4   let _d2 = data.lock(); // cannot lock, since d1 is still active
5 }
```

Listing 4.4: Un deadlock causado por llamar a lock dos veces sobre el mismo mutex.

Observe ahora un extracto de la MIR del mismo programa erróneo en el Listado 4.5. Se han eliminado los comentarios para mayor claridad. En BB0 el mutex se crea mediante una llamada a `std::sync::Mutex::new`. El nuevo mutex es el valor de retorno de la función. Se asigna a la variable local `_1`. A continuación, la ejecución continúa en BB1. Concéntrese en la primera sentencia de BB1: Una referencia inmutable a la variable local `_1` se almacena en `_3`. A continuación, la referencia se traslada a la función `std::sync::Mutex::lock`. Esta referencia es consumida por *lock*, es decir, la variable local `_3` no se utiliza en ningún otro lugar de la MIR porque, a partir de ese momento, la propiedad de la referencia se transfiere a la función `std::sync::Mutex::lock`.

Inmediatamente después de la sentencia, el traductor se encuentra con el terminator de BB1. Contiene una llamada a `std::sync::Mutex::lock`. ¿Cómo sabría el traductor, al traducir esta llamada, que `_3` es efectivamente el mutex almacenado en `_1`? Este es el problema que pretende resolver el modelado de la memoria de la función.

```
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: Un extracto de la MIR del programa del Listado 4.4.

El problema va aún más lejos. La variable local `_2` contiene un mutex guard después de la llamada a lock, que también debería registrarse. Observe cómo BB2 repite las mismas operaciones que BB1 pero utiliza variables locales diferentes, `_5` y `_4`. El traductor debería saber que `_5` es también un alias de `_1`. Además, las guardas del mutex en `_2` y `_4` serán eventualmente eliminados, lo que indirectamente desbloquea el mutex. Tiene que haber un enlace desde la guarda de mutex en `_2` y `_4` al mutex en `_1`. Más concisamente, el traductor debe supervisar qué mutex está detrás de cada guarda de mutex.

Para complejizar más las cosas, cada función MIR tiene su propia memoria de pila, con sus variables locales separadas `_0`, `_1`, `_2`, `_3`, etcétera. Por lo tanto, la asignación de ubicaciones de memoria a variables de sincronización no puede ser una única estructura global. En su lugar, depende del contexto de la función actual que se esté traduciendo. Por último, una variable de sincronización puede migrar de una función a otra y el traductor debe ser capaz de reasignarlas correctamente.

Esto basta como breve ejemplo práctico de los retos que plantea el modelado de la memoria. Ahora podemos presentar la solución que se ha aplicado.

4.5.2. Una asignación de `rustc_middle::mir::Place` a un contador de referencias compartido

La implementación se denomina adecuadamente `Memory`[36]. Como se anticipó en la sección anterior, La memoria está estrechamente conectada al contexto de la función MIR.

En lugar de mover valores entre distintas ubicaciones de memoria, como se observa en la MIR, nuestra solución se basa en el concepto más sencillo de "vinculación" o "enlazado". Esto implica asociar un `rustc_middle::mir::Place` específico con el valor correspondiente. Esta asociación no se elimina al trasladar la variable a una función diferente. Tampoco diferencia una copia superficial del valor de tomar una referencia o una referencia mutable. En pocas palabras, se trata de un mapeo global entre lugares y valores.

Para dar cabida a la posibilidad de vincular el mismo valor a varios lugares, en particular cuando varias posiciones de memoria mantienen una referencia inmutable al valor, se hace necesario que el valor almacenado sea una referencia a la variable de sincronización. Para aclararlo, esto introduce un segundo nivel de indirección. Para facilitar las operaciones de clonación necesarias, hemos optado por utilizar `std::rc::Rc`, que es un puntero inteligente proporcionado por la biblioteca estándar de Rust. La propiedad del valor referenciado (la variable de sincronización) es compartida y cada vez que se clona el valor, se incrementa un contador interno. Cuando el contador llega a cero, el valor se libera [Klabnik and Nichols, 2023, Chap. 15.4].

La `Memory` hace uso de una estructura de datos `std::collections::HashMap` que establece un mapeo entre instancias de `rustc_middle::mir::Place` y un enum con 5 variantes correspondientes a los 5 tipos mencionados anteriormente que el traductor rastrea. 4 de estas 5 variantes encierran una referencia `std::rc::Rc` a la variable de sincronización. El caso de los agregados contiene en cambio un vector de `Value`. Esto hace posible anidar valores agregados unos dentro de otros, lo que es un requisito crítico para soportar programas más complejos con `structs` anidados.

El uso de un hash map permite recuperar y gestionar eficazmente los valores asociados durante

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

```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: Resumen de las definiciones de tipos de la implementación de `Memory`.

el proceso de traducción. La `Memory` también se encarga de proporcionar los typedefs para las diferentes referencias a las variables de sincronización. El Listado 4.6 muestra un pedazo del código fuente con las definiciones de tipos esenciales utilizadas en la implementación. Las mejoras a la implementación actual se discuten en la Sec. 6.5.

4.5.3. Interceptando asignaciones

La pieza que falta en el rompecabezas del modelo de memoria es dónde enlazar exactamente las posiciones de memoria. Hay tres lugares distintos en el código en los que esto tiene lugar.

Por un lado, las funciones traductoras encargadas de procesar los métodos de los mutexes, las variables de condición y los hilos crean nuevas variables de sincronización que se vinculan al valor de retorno del método correspondiente. Aquí comienza la vida útil de cada variable de sincronización. Los detalles específicos se amplían en las Sec. 4.7.3 y 4.8.3

Por otro lado, la variable de sincronización puede asignarse en cualquier otro BB. Por esta razón, el traductor incorpora una implementación personalizada del método `visit_assign` para interceptar cada asignación en la MIR. El Listado 4.7 muestra con precisión que todos los casos de copia, desplazamiento o referencia al lado derecho (right-hand side (RHS)) se gestionan mediante el mismo mecanismo: El lado izquierdo (left-hand side (LHS)) está vinculado al lado derecho (right-hand side (RHS)) si el tipo de la variable es una variable de sincronización admitida. El listado también muestra cómo el compilador utiliza enums anidados para modelar sus datos. Dentro de las variantes de un valor del lado derecho (`rustc_middle::mir::Rvalue`), se pueden encontrar operandos (`rustc_middle::mir::Operand`). Estos operandos también aparecen al pasar argumentos de función.

El caso más peculiar es la asignación agregada. Se materializa a partir de asignaciones en el código fuente de Rust que crean tuplas, closures o `structs`. Requiere un manejo especial ya que el valor a enlazar en la memoria debe ensamblarse a partir de los constituyentes del valor agregado que son una variable de sincronización. Esto implica que la `Memory` sólo conserva la parte del valor agregado formada por las variables de sincronización.

El seguimiento de las asignaciones de las variables de sincronización en el momento en que son devueltas por las funciones es otro mecanismo crucial. Afortunadamente, esto puede lograrse implementando una comprobación uniforme en todas las funciones, independientemente de si se modelan utilizando el modelo simple (Fig. 3.2) o el modelo de función con limpieza (Fig. 4.2). Como ventaja, esta comprobación uniforme soporta fácilmente `std::arc::Arc` sin necesidad de ningún esfuerzo adicional.

En todos los casos, el manejo de las asignaciones no tiene ningún impacto en la red de Petri. No se añaden lugares ni transiciones al interceptar las asignaciones.

Por último, algunas posiciones de memoria se pasan a un nuevo hilo al llamar a `std::thread::spawn` y se mapean de nuevo a la memoria de la función del hilo. La siguiente sección demostrará el método utilizado para lograr esto.

```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: La implementación personalizada de `visit_assign` para rastrear variables de sincronización.

4.6. Multihilo

El soporte multihilo es un requisito previo para la detección de puntos muertos y señales perdidas. Para soportar programas del mundo real en los que los bloqueos o las señales perdidas son posibles en primer lugar, resulta esencial soportar la existencia de varios hilos que compartan recursos. En primer lugar, se presentarán los fundamentos para después idear un modelo PN

que capture el comportamiento de los hilos en el código Rust.

4.6.1. Vida útil del hilo en Rust

La vida (*lifetime*) de un hilo comienza cuando se invoca la función `std::thread::spawn`[37]. Esta recibe como argumento una closure o función, que representa el código que el nuevo hilo ejecutará concurrentemente con los demás hilos del programa. El nuevo hilo puede empezar a ejecutarse inmediatamente después de ser *spawned*, pero no hay garantía de que lo haga.

A diferencia de otros lenguajes de programación como C, C++ o Java, en Rust no existe la noción de una variable de hilo inicializada previamente al inicio del hilo. En su lugar, la función `std::thread::spawn` devuelve un `std::thread::JoinHandle`, que es, como su nombre indica, un handle para llamar a `join` al final de la vida del hilo.

Durante su existencia, un hilo puede ejecutar de forma independiente su código designado y realizar diversas operaciones concurrentemente con otros hilos. Puede acceder a recursos compartidos y comunicarse con otros hilos a través de mecanismos de sincronización como mutexes, variables de condición, canales u operaciones atómicas. Esto permite el procesamiento concurrente y el paralelismo en los programas Rust.

Para garantizar una coordinación adecuada entre hilos, Rust proporciona un mecanismo para unir hilos. El método `std::thread::JoinHandle::join`[38] permite al hilo principal o a otro hilo esperar hasta la finalización de otro hilo. Al llamar a join sobre un join handle, el hilo llamante se bloquea hasta que el hilo iniciado previamente finaliza su ejecución. Una vez que un hilo finaliza su ejecución y es unido por otro hilo, finaliza su vida útil y se liberan los recursos correspondientes del sistema. De lo contrario, los hilos que no se unieron correctamente pueden potencialmente perder recursos.

Si se libera el join handle, el hilo ya no puede unirse y se convierte implícitamente en *desacoplado* (*detached*). Un hilo *detached* se refiere a un hilo sin un join handle válido. Continuará su ejecución de forma independiente hasta que finalice o el programa termine. Son útiles en escenarios en los que el hilo generador no necesita esperar a que este complete su tarea. Por ejemplo, en tareas en segundo plano de larga duración o cuando el hilo principal termina independientemente del progreso del hilo desprendido. Sin embargo, es importante destacar que la ejecución de los hilos desvinculados puede continuar *incluso* después de que el hilo principal haya terminado.

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

4.6.2. Modelo de red de Petri para un hilo

Para incorporar hilos adicionales al modelo PN, se añade una subred distinta a la red principal para representar cada hilo. Esta subred encapsula la ruta de ejecución del nuevo hilo generado y funciona como un contexto aislado. Establece interfaces precisas que conectan de nuevo con la red principal. La *closure* proporcionada a la función `spawn`, al ser una función MIR, puede invocar otras funciones que a su vez requieren traducción. Por lo tanto, el procesamiento de la función de un hilo sigue un enfoque similar al de la lógica de traducción habitual.

El aspecto de concurrencia de la ejecución del nuevo hilo se modela mediante la generación de un nuevo token en la transición que representa la llamada a `spawn`. Este token puede interpretarse, del mismo modo que el token en `PROGRAM_START`, como el contador de instrucciones del nuevo hilo. Esencialmente, la operación spawn constituye una "bifurcación" en el flujo de tokens: Un token entra en la transición y dos tokens salen de ella. El primero avanza por el camino del hilo principal para ejecutar la sentencia posterior, mientras que el segundo se dirige al primer BB de la función pasada al hilo

Cada hilo identificado en el código fuente posee unos lugares designados de inicio y fin etiquetados como `THREAD_<index>_START` y `THREAD_<index>_END`, respectivamente. El índice es obligatorio para prever la propiedad de unicidad de la etiqueta en todo el programa. Cabe destacar que esto sigue el patrón de los lugares básicos del programa detallados en la Sec. 4.1.1.

Los hilos carecen de un lugar de pánico separado, ya que invocar `panic!` dentro de un hilo sólo termina la ejecución de ese hilo específico. No nos interesa diferenciar entre los estados finales de los hilos; el requisito principal es determinar si un hilo ha terminado o no. Aquí basta con un único lugar de finalización para ambos casos.

El comportamiento de unión sirve como operación inversa de la generación. La transición correspondiente a la llamada `join` consume dos tokens pero genera sólo un token. Como resultado, la condición de espera se modela de forma directa: El hilo principal puede continuar, o sea, la transición `join` puede dispararse, si y sólo si el hilo a unir ha finalizado la ejecución, alcanzando su respectivo lugar `THREAD_END`.

Recapitulando, el hilo se traslada a una subred separada que interactúa con la red principal sólo en tres lugares:

- La transición de `spawn` donde comienza el hilo.
- La transición de `join` (opcional) en la que se utiliza el join handle.
- Las conexiones debidas a variables de sincronización, analizadas más adelante en las secciones dedicadas de este capítulo.

4.6.3. Un ejemplo práctico

Observe el Listado 4.8 y su correspondiente modelo de red de Petri en la Fig. 4.6. Se trata de uno de los programas de prueba que se encuentran en el repositorio. Observe la "bifurcación" en la transición de spawn descrita en la subsección previa. La rama izquierda es el nuevo hilo, mientras que la rama derecha es el hilo principal. Está claro que las rutas se dividen en el `spawn` y se fusionan en el `join`. Observe también que no hay un lugar de pánico separado para el hilo, lo que indica que un fallo en un hilo no afecta a los demás hilos.

```
fn main() {
  let thread_join_handle = std::thread::spawn(move || {
      // some work here
  });
  // some work here
  let _res = thread_join_handle.join();
}
```

Listing 4.8: Un programa básico con dos hilos para demostrar el soporte multihilo.

4.6.4. Algoritmos para la traducción de hilos

Para terminar esta sección, describiremos brevemente los algoritmos utilizados para traducir los hilos. Inicialmente, cabe mencionar que, dado que la traducción la realiza un único hilo (la herramienta no admite múltiples hilos traduciendo el código fuente), hay que tomar una decisión sobre cuándo traducir los hilos generados:

- Traducción inmediata: Traduce el hilo en cuanto lo encuentra. El traductor "cambia" al hilo generado.
- Traducción diferida: Almacena toda la información relevante sobre el nuevo hilo y lo traduce después del hilo principal.

La solución actual adopta este último enfoque.

Cuando se encuentra una llamada a `std::thread::spawn`:

1. Traducir la llamada a la función utilizando el modelo visto en la Fig. 4.2.
2. Recuperar el primer argumento pasado a la función: Un valor agregado que contiene las variables capturadas por la clausura y la función que ejecutará el hilo.
3. Extraer el ID de la función que debe ejecutar el hilo.
4. Extraer los valores capturados por la clausura.

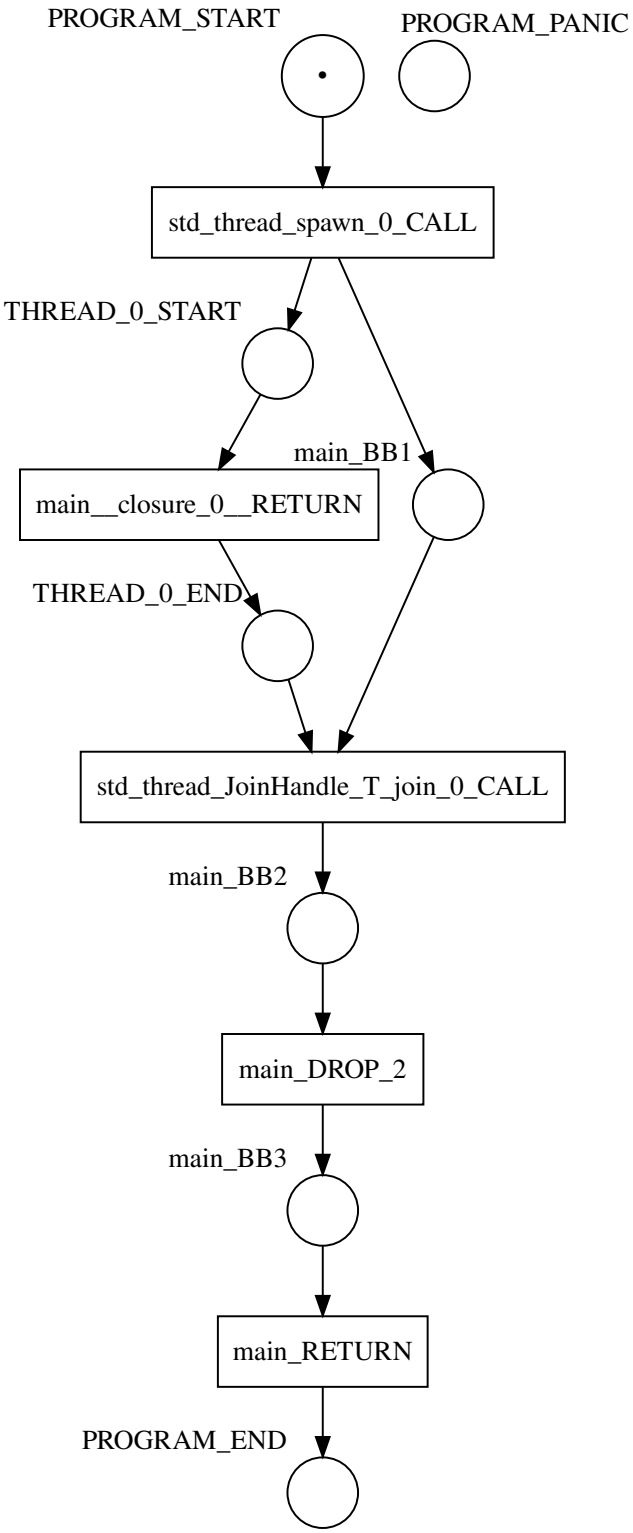

Figura 4.6: El modelo de red de Petri para el programa del Listado 4.8.

5. Crear un nuevo `Thread`[39] para almacenar la información necesaria para la traducción retardada.
6. Vincular el valor de retorno de `std::thread::spawn`, el nuevo join handle, al `Thread`.
7. Agregar el hilo a una cola de hilos detectados en el `Translator`.

Cuando se encuentra una llamada a `std::thread::JoinHandle::join`:

1. Traducir la llamada a la función utilizando el modelo visto en la Fig. 3.2. Ignorar el lugar de limpieza ya que debemos obligar a la PN a "esperar" a que el hilo salga. Esto equivale a suponer que la función `join` nunca falla.
2. Recuperar el primer argumento pasado a la función: El join handle. La posición de memoria está vinculada al hilo correspondiente gracias a la interceptación de asignaciones explicada en la Sec. 4.5.3.
3. Establecer la transición de join del `Thread` referenciado por el join handle.

Cuando el hilo principal termina de traducir, es decir, cuando la función `main` ya ha sido procesada, el `Translator` entra en un bucle para traducir los hilos descubiertos hasta el momento en orden.

1. Crear un nuevo lugar de inicio y final para el hilo.
2. Conectar la transición de spawn al lugar de inicio.
3. Si se ha encontrado una transición de join, conectar el lugar final a ella.
4. Sustituir el lugar `PROGRAM_PANIC` por el lugar `THREAD_<index>_END` para traducir correctamente las sentencias terminadoras como `Unwind` (Sec. 4.4.3).
5. Agregar la función del hilo a la pila de llamadas.
6. Mover las variables de sincronización a la memoria de la función del hilo, es decir, asigne el agregado (tuple, `struct`, etc.) y sus campos a la memoria de la función del hilo.
7. Traducir el elemento superior de la pila de llamadas.

Como se anticipó antes, el algoritmo muestra semejanzas con el procedimiento general para las llamadas a funciones esbozado en la Sec. 4.2. Por último, la implementación es capaz de manejar programas en los que los hilos engendran sus propios hilos de forma anidada. Los hilos simplemente se añaden a la cola y, a medida que avanza el bucle, los hilos anidados también se trasladan.

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

4.7. Mutex (`std::sync::Mutex`)

Un mutex, abreviatura de exclusión mutua, es un mecanismo de sincronización utilizado para controlar el acceso a un recurso compartido en un programa concurrente. Permite que múltiples hilos accedan al recurso compartido recurso de forma mutuamente excluyente, asegurando que sólo un hilo pueda acceder al recurso a la vez.

En esta sección, se explica el modelo de red de Petri para un mutex en Rust, después se presenta un ejemplo práctico para facilitar la comprensión y, por último, se esbozan los algoritmos utilizados para la traducción de funciones mutex.

4.7.1. Modelo de red de Petri

En Rust, un mutex se crea envolviendo los datos compartidos en un tipo `Mutex<T>`, donde `T` es el tipo del recurso compartido. El tipo `std::sync::Mutex` expone un método llamado `lock` para adquirir el bloqueo del recurso compartido. Si el mutex está actualmente desbloqueado, el hilo adquiere con éxito el lock y puede proceder a acceder al recurso. Si el mutex ya está bloqueado por otro hilo, el hilo que intente adquirir el lock se bloqueará hasta que el lock esté disponible. El método `lock` devuelve una guarda de mutex (`std::sync::MutexGuard`) que garantiza el acceso exclusivo al recurso hasta que se libere.

A diferencia de la semántica de `unlock` presente en C o C++, el mutex incluido por la biblioteca estándar de Rust se desbloquea implícitamente, es decir, sin llamar a una función. El mutex implementa la Adquisición de Recursos es Inicialización (Resource Acquisition Is Initialization (RAII)) y libera el bloqueo automáticamente cuando sale de scope, evitando los deadlocks. Alternativamente, liberar una variable local de tipo `std::sync::MutexGuard` equivale a desbloquear el mutex correspondiente.

Un mutex puede modelarse en redes de Petri como un único lugar que representa el estado del mutex, indicando si está bloqueado o desbloqueado. El lugar se etiqueta para reflejar su propósito como mutex. Además, el lugar se marca con un token inicialmente para significar que el mutex comienza en el estado desbloqueado.

Las transiciones que bloquean el mutex consumen el token del lugar del mutex. Si el token está ausente, la transición no puede dispararse. El mutex debe estar en estado desbloqueado para activar la transición de bloqueo que es el comportamiento deseado.

Las transiciones que desbloquean el mutex producen como salida un token en el lugar del mutex. La transición puede dispararse mientras el programa llegue a ese punto de la ejecución. Después de que la transición se dispare el lugar del mutex vuelve a contener un token que puede ser consumido por una transición de bloqueo. Dos tipos de transiciones pueden desbloquear el mutex:

1. Un terminador `Drop` (Sec. 4.4.3) cuando el lugar liberado es de tipo `std::sync::MutexGuard`.

2. La transición para una llamada a `std::mem::Drop` que libera la memoria ocupada por el valor pasado como argumento explícitamente.

Al conectar el lugar del mutex con las transiciones de bloqueo y desbloqueo mediante arcos de entrada y salida, establecemos la relación entre el estado del mutex y las acciones que lo manipulan. Este enfoque de modelado permite representar el comportamiento del mutex en una PN y facilita el análisis de sus interacciones con otras partes del sistema.

El modelo de red de Petri presentado aquí es bien conocido en la literatura y se ha aplicado con éxito en otras herramientas. Puede encontrarse, entre otros, en [Kavi et al., 2002, Moshtaghi, 2001, Meyer, 2020, Zhang and Liua, 2022].

4.7.2. Un ejemplo práctico

Considere el modelo de red de Petri mostrado en la Fig. 4.7 correspondiente al programa del Listado 4.4. El MIR se ilustra en 4.5. Este programa de prueba es uno de los ejemplos incluidos en el repositorio.

Observe que hay dos transiciones de bloqueo que corresponden a las dos llamadas a `lock` en el código fuente. Los índices reflejan el orden de su aparición en el programa, lo que explica las etiquetas `std_sync_Mutex_T_lock_0_CALL` y `std_sync_Mutex_T_lock_1_CALL`. Ambas tienen un arco de entrada desde el lugar del mutex `MUTEX_0`.

Como ya se ha mencionado, los terminadores `Drop` pueden desbloquear un mutex. No importa si fallan o no (el caso de error incluye el sufijo `_UNWIND`), un arco saliente fluye de vuelta al lugar del mutex para reponer el token.

Cabe destacar que hay más arcos entrantes al lugar mutex que salientes, lo que pone de relieve la importancia de seguir las guardas de mutex a lo largo de la MIR utilizando la estrategia explicada en la Sec. 4.5.3.

4.7.3. Algoritmos para la traducción del mutex

Para concluir esta sección, ofreceremos un breve resumen de los algoritmos empleados en la traducción de funciones mutex.

Cuando se encuentra una llamada a `std::sync::Mutex::new`:

1. Traducir la llamada a la función utilizando el modelo visto en la Fig. 4.2.
2. Crear una nueva structura `Mutex`[40] con un índice para identificarlo inequívocamente en toda la PN.

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

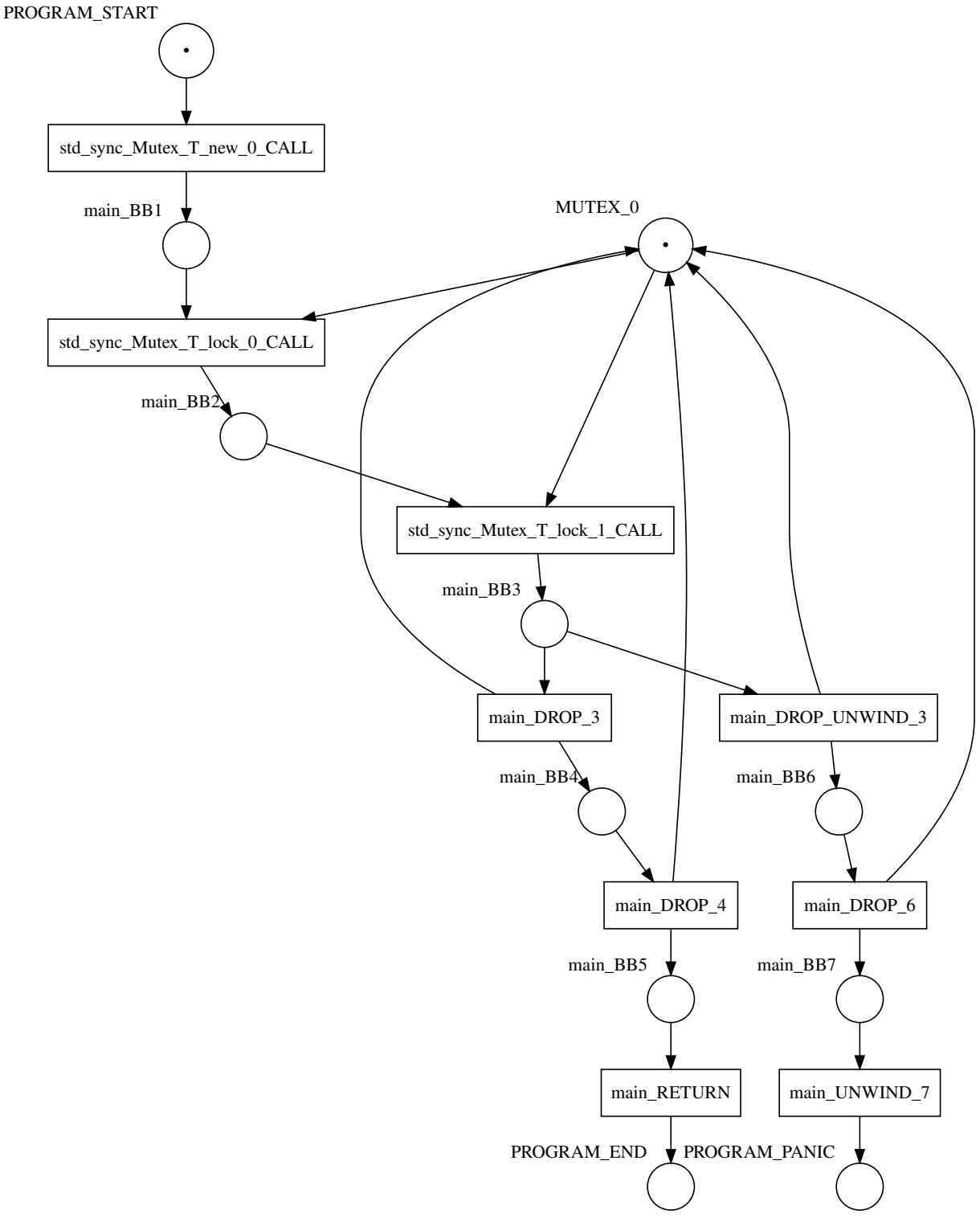

Figura 4.7: El modelo de red de Petri para el programa del Listado 4.4.

3. Vincular el valor de retorno de `std::sync::Mutex::new`, el nuevo mutex, a la estructura `Mutex`.

Cuando se encuentra una llamada a `std::sync::Mutex::lock`:

1. Traducir la llamada a la función utilizando el modelo visto en la Fig. 3.2. Ignorar el lugar de limpieza ya que debemos obligar a la red de Petri a "esperar" a que se marque el lugar mutex. Esto equivale a suponer que la función `lock` nunca falla.
2. Recuperar la auto-referencia `self` al mutex sobre el que se llama a la función.
3. Añadir un arco desde el lugar del mutex subyacente hacia la transición que representa la llamada a la función.
4. Crear una nueva `MutexGuard` con una referencia al `Mutex`.
5. Vincular el valor de retorno de `std::sync::Mutex::lock`, la nueva guarda de mutex, a la estructura `MutexGuard`.

Cuando se encuentra una llamada a `std::mem::drop`:

1. Traducir la llamada a la función utilizando el modelo visto en la Fig. 4.2.
2. Extraer el argumento pasado a la función.
3. Si el argumento está vinculado a una guarda de mutex, añadir un arco desde la transición de la llamada a la función hasta el lugar del mutex referenciado por la guarda de mutex.
4. Si se proporcionó un lugar de limpieza, añadir también un arco de desbloqueo desde la transición de limpieza al lugar del mutex referenciado por la guarda de mutex.

Cuando se encuentra un terminador del tipo `rustc_middle::mir::TerminatorKind::Drop`:

1. Si la variable a liberar está vinculada a una guarda de mutex, añadir un arco desde la transición de la llamada a la función hasta el lugar del mutex referenciado por la guarda de mutex.
2. Si se proporcionó un lugar de limpieza, añadir también un arco de desbloqueo desde la transición de limpieza al lugar del mutex referenciado por la guarda de mutex.

En la próxima sección profundizaremos en los ajustes necesarios de estos algoritmos para establecer un modelo unificado para las variables de condición que es esencial para detectar las señales perdidas. Dado que estas modificaciones se comprenden mejor en el marco de las variables de condición, las dilucidaremos en ese contexto específico.

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

Una condition variable es una primitiva de sincronización que permite que uno o más hilos esperen una determinada condición antes de proceder con su ejecución. Los hilos esperan hasta que otro hilo les notifica que se ha cumplido la condición deseada.

Las variables de condición suelen estar asociadas a un mutex que garantiza el acceso exclusivo a los datos compartidos de los que depende la condición. Cuando un hilo espera en una condition variable, libera el mutex asociado, permitiendo que otros hilos avancen. Cuando la condición se convierte en verdadera o se produce algún evento, un hilo notificador señala la variable de condición, permitiendo que uno o más hilos en espera reanuden su ejecución.

La semántica de las variables de condición, así como ejemplos en pseudocódigo, se introdujeron en la Sec. 1.5. La comprensión del comportamiento preciso de las condition variables en todas las circunstancias es un requisito previo para esta sección.

Esta sección ofrece una explicación detallada del modelo de red de Petri utilizado para representar las condition variable de la biblioteca estándar de Rust. Le sigue un ejemplo práctico que pretende aumentar la claridad de los conceptos. Por último, se esbozan los algoritmos para la traducción de funciones de variables de condición.

4.8.1. Modelo de red de Petri

En este caso concreto, el modelo de red de Petri debe examinarse detenidamente puesto que implica no sólo a la propia variable de condición, sino también a la variable que mantiene la condición sobre la que espera el hilo bloqueado y al mutex que sincroniza el acceso a dicha condición.

En general, esta interacción puede ser extremadamente compleja. La misma variable de condición podría utilizarse para señalar o rastrear un número arbitrario de condiciones distintas. En consecuencia, pueden pasarse diferentes mutexes como argumento a la llamada de `wait`. Además, un número arbitrario de hilos puede bloquearse en una variable de condición y Rust admite la operación de *broadcast* para despertar a todos los hilos en espera a la vez mediante el método `notify_all`[41] (véase la Sec. 1.5). Y lo que es más importante, la condición en sí puede ser de cualquier tipo y tomar una larga secuencia de valores durante la ejecución, en función de los cuales los hilos en espera podrían actuar de diversas maneras en cada escenario.

Por todo ello, es inevitable hacer suposiciones sobre los casos de uso soportados para variables de condición reducir la complejidad de la tarea. Abarcar y manejar todas las posibilidades queda fuera del alcance de esta tesis.

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

Supuestos

1. *Llamada única*: Sólo hay una llamada a esperar (`wait`) por variable de condición. Equivalentemente, `condvar.wait()` aparece en un único lugar del código fuente para una `condvar` determinada. Por ejemplo, puede estar dentro de un bucle pero no puede estar en dos funciones diferentes.
2. *Cola de un solo elemento*: Hay como máximo un hilo de espera por condition variable.
3. *Condición booleana*: La condición es un flag booleano. Está activo o no, dos valores posibles en total. Esperar en una condición que puede tomar 3 o más valores *no* es soportado por este modelo.
4. *Establecimiento obligatorio de la condición / Sin "notificación falsa"*: Si un hilo bloquea el mutex y accede mutablemente a la condición compartida, entonces siempre almacena un valor diferente. En términos más sencillos, los hilos que miran el valor, no lo cambian e inmediatamente llaman `signal` sobre la condition variable no están permitidos.
5. *Exclusión de broadcast*: El método `std::sync::Condvar::notify_all` está fuera de alcance.

Podría implementarse el soporte para múltiples llamadas a `wait` y múltiples hilos de espera, pero se requiere un considerable esfuerzo de implementación. Por lo tanto, los Supuestos 1 y 2 pueden superarse con el modelo propuesto.

Admitir condiciones no booleanas y detectar qué valor toma la condición después de cada acceso exige reconsiderar a fondo el enfoque de modelado para representar valores de datos concretos en redes de Petri simples. En consecuencia, los Supuestos 3 y 4 son especialmente desafiantes y podrían ser objeto de futuras investigaciones sobre modelos en redes de Petri de nivel superior. Véase la Sec. 6.6 para algunas reflexiones en este sentido.

Análisis del modelo propuesto

La Fig. 4.8 muestra el modelo de red de Petri utilizado en la implementación. El mismo diagrama en formato DOT, PNG y SVG puede encontrarse en el repositorio como documentación.

Los lugares de entrada son:

- `input`: El lugar de inicio de la función wait. El modelo admite los métodos de la biblioteca estándar `std::sync::Condvar::wait` y su variación `std::sync::Condvar::wait_while`.
- `condition_not_set`: El lugar está marcado cuando la condición es `false`.
- `condition_set`: El lugar está marcado cuando la condición es `true`.
- `notify`: El lugar donde el hilo notificador coloca un token para despertar al hilo en espera.

El lugar de salida (`output`) es el lugar de finalización de la llamada a la función `wait` o `wait_while`. La ejecución del hilo continúa a partir de ahí.

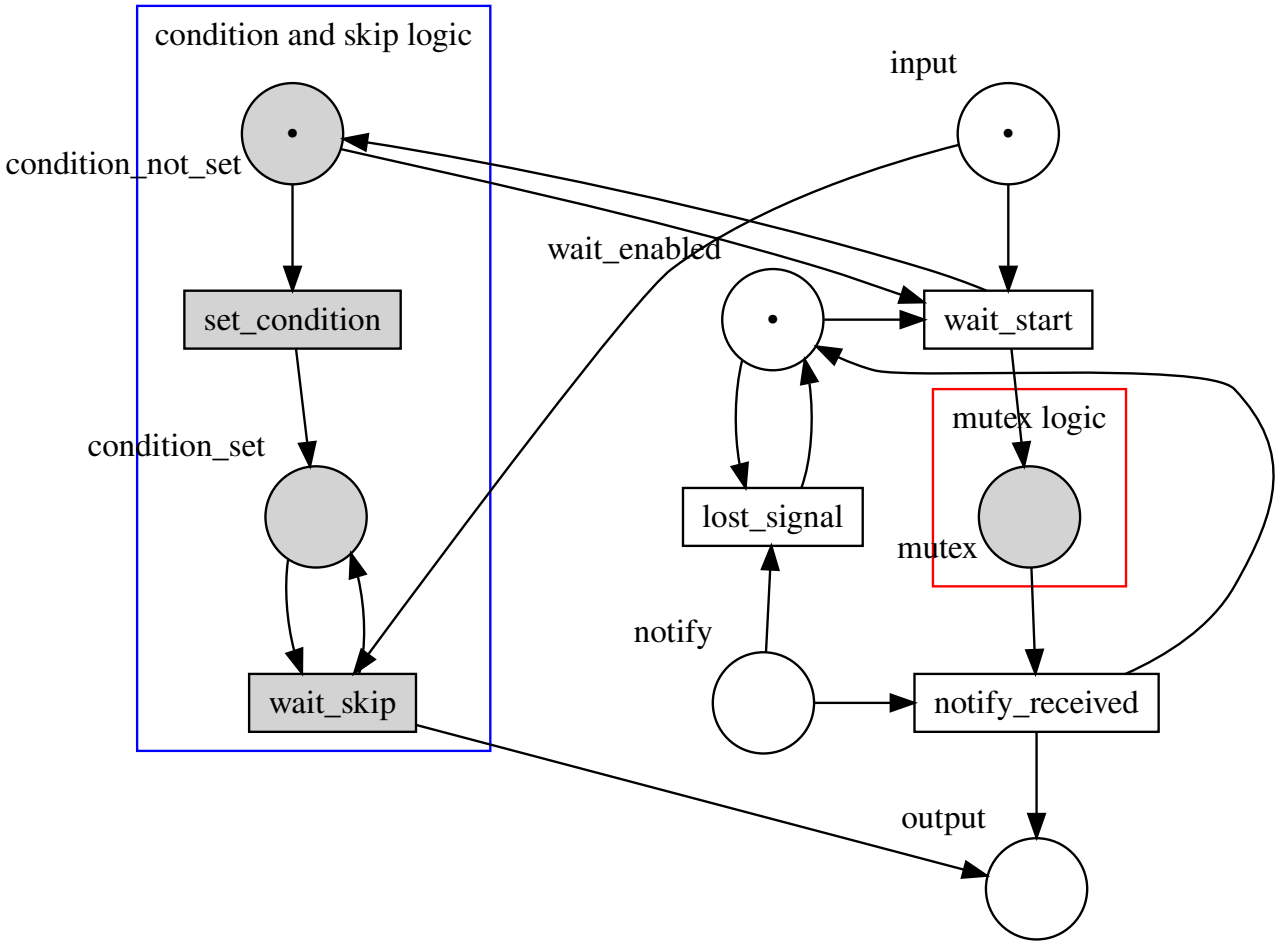

Figura 4.8: El modelo de red de Petri para las condition variables.

Existen dos formas posibles de pasar de `input` a `output`, representadas por dos transiciones:

- `wait_start`: Este es el "caso común", el hilo se bloquea y espera la señal.
- `wait_skip`: Este es el camino alternativo que toma el hilo cuando la condición ya se ha cumplido. El hilo no espera, en su lugar, se salta la espera y alcanza la salida `output` en un solo salto.

Es esencial observar que la parte en gris de la izquierda de la Fig. 4.8 controla qué transición se activa y qué transición se desactiva. En cuanto se establece un testigo en `condition_set`, `wait_start` se desactiva. Antes de eso, ocurre lo contrario: `wait_start` puede dispararse pero `wait_skip` no.

Nótese los arcos entre `condition_not_set` y `wait_start`. La ficha se regenera cada vez que se dispara `wait_start`. Lo mismo ocurre con `condition_set` y `wait_skip`. Estos arcos restauran los lugares de condición a su estado anterior. Modelar más de 2 valores como una PN implicaría una red más enrevesada, Además, sería imposible saber en tiempo de compilación cuántos valores posibles toma la condición para generar el número correcto de lugares. Esta limitación es la justificación de los Supuestos 3 y 4.

Ahora concéntrese en el lado derecho de la Fig. 4.8. En el centro de la variable de condición, encontramos el lugar para el mutex. Como era de esperar, se desbloquea cuando comienza la espera y se bloquea cuando se recibe la notificación.

El lugar etiquetado `wait_enabled` desempeña un papel central. Por un lado, consume el testigo de `notify` si `wait_start` no se disparó. Este es el caso arquetípico de señal perdida que nos gustaría detectar. Por otro lado, el token en `wait_enabled` se consume cuando `wait_start` se dispara. Esto evita que la condition variable "acepte" otros hilos (Supuesto 2) y conserva el token en `notify`, asegurando que la señal perdida no pueda ocurrir.

Por último, la transición `notify_received` combina los requisitos para que el hilo abandone la espera: El mutex debe estar desbloqueado y `notify_one` ha sido llamado. Para restaurar el estado inicial de la condition variable, regenera el testigo en `wait_enabled`.

Requisitos de traducción a nivel global del modelo de red de Petri

Un desafío fundamental que surge durante la implementación del modelo de la Fig. 4.8 es que las conexiones a través de la frontera azul del diagrama no pueden establecerse en el caso general cuando se procesa la llamada a `wait`. Analizaremos el problema y explicaremos cómo lo afronta la solución.

La transición en la que se establece la condición, denominada `set_condition` en el diagrama, es la siguiente candidata. En la implementación actual, la transición seleccionada para cumplir este papel es la llamada a `std::ops::DerefMut::deref_mut` cuando se está desreferenciando un mutex o una guarda de mutex.

Considere el Listado 4.9, otro programa de prueba del repositorio. En la línea 9, la guarda de mutex es dereferenciada para escribirle el valor `true`, cumpliendo la condición para la condition variable. En la MIR, esto se corresponde con una llamada a `std::ops::DerefMut::deref_mut`. Aunque no es el lugar exacto donde se escribe el valor (que en realidad sería una sentencia dentro de un BB), se aproxima lo suficiente y satisface nuestras necesidades.

```
1  fn main() {
2        let pair = std::sync::Arc::new((std::sync::Mutex::new(false),
         ↪  std::sync::Condvar::new()));
3        let pair2 = std::sync::Arc::clone(&pair);
4
5        // Inside of our lock, spawn a new thread, and then wait for it to start.
6        std::thread::spawn(move || {
7            let (lock, cvar) = &*pair2;
8            let mut started = lock.lock().unwrap();
9            *started = true;
10           // We notify the condvar that the value has changed.
11           cvar.notify_one();
12       });
13
14       // Wait for the thread to start up.
15       let (lock, cvar) = &*pair;
16       let mut started = lock.lock().unwrap();
17       while !*started {
18           started = cvar.wait(started).unwrap();
19       }
20 }
```

Listing 4.9: Un programa que requiere información global de la red de Petri para ser traducido.

Lamentablemente, las conexiones con la variable de condición tampoco pueden establecerse al procesar la llamada a `deref_mut`. La razón es que no hay ninguna garantía de que la condition variable ya se haya visto. Todavía podría estar por delante en el camino que sigue la traducción. En el Listado 4.9, el hilo principal se traduce primero, por lo que la variable de condición se descubre primero. Pero si los papeles de los hilos s e intercambian, entonces la traducción no puede realizarse.

Llegamos así a una conclusión desalentadora. Para conectar el modelo de la variable de condición con los lugares que modelan la condición y las transiciones en las que se establece, necesitamos toda la red de Petri o en otras palabras, necesitamos información *global* para traducir eficazmente la primitiva de sincronización.

Como resultado, es inevitable incorporar algún clase de etapa de postprocesamiento a la traducción. Las tareas también deben realizarse en un orden específico. Primero debe descubrirse

el mutex. Después puede vinculárselo a una variable de condición si se encuentra tal condition variable (el código fuente podría no utilizar ninguna). De ahí que sea aconsejable introducir una noción de "prioridad" en las tareas de postprocesamiento.

El `Translator` se basa en un `std::collections::BinaryHeap` para implementar una cola de prioridad de `PostprocessingTask`[42]. Las tareas son devueltas por los métodos que traducen primitivas de sincronización si es necesario. Una vez traducidos todos los hilos, el `Translator` aborda las tareas de postprocesamiento. Al completarlas según su prioridad, garantizamos que la información esté disponible en el orden requerido.

Table of possible inputs and expected outputs

Como complemento a la explicación de los apartados anteriores, he aquí una tabla que resume la salida esperada para una entrada determinada. El lector puede comparar la Fig. 4.8 para comprobar que el modelo produce la salida correcta para cada escenario.

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)

Cuadro 4.1: Resumen de los posibles estados del modelo de red de Petri para las condition variables.

4.8.2. Un ejemplo práctico

Debido a las limitaciones de tamaño de la red de Petri, nos vemos obligados a seleccionar un programa a pequeña escala con fines de demostración. Sería inviable incluir en una sola página la red de Petri completa de un programa realista con condition variables y múltiples hilos. Para obtener ejemplos más completos, se anima a los lectores a explorar el repositorio, el cual contiene una colección de programas más intrincados incluidos como parte de las pruebas de integración.

[42] https://github.com/hlisdero/cargo-check-deadlock/blob/main/src/translator/function.rs#L92

A pesar de las limitaciones de espacio, el ejemplo del Listado 4.10 comprende los elementos centrales del modelo presentado anteriormente. La red de Petri completa puede verse en la Fig. 4.9.

Observe la siguiente secuencia de transiciones:

1. El mutex se bloquea en `std_sync_Mutex_T_lock_0_CALL`.
2. `std_sync_Condvar_notify_one_0_CALL` coloca un token en `CONDVAR_0_NOTIFY`.
3. El flujo del token continúa a `main_BB5` justo antes de `CONDVAR_0_WAIT_START`.

Cabe destacar que no se produce ningún deadlock si la transición `CONDVAR_0_WAIT_START` se dispara *antes* de `CONDVAR_0_LOST_SIGNAL`. Dicho brevemente, existe un conflicto entre las transiciones `CONDVAR_0_WAIT_START` y `CONDVAR_0_LOST_SIGNAL` por el token en `CONDVAR_0_NOTIFY`. No obstante, el verificador de modelos comprueba *todos* los posibles disparos y descubrirá el caso de la señal perdida sin dificultades.

Otra observación digna de mención es que este programa ilustra el efecto que tendrían los caminos alternativos de limpieza en la detección de señales perdidas. Si hubiera una segunda transición al mismo nivel que `CONDVAR_0_WAIT_START` o `std_sync_Condvar_notify_one_0_CALL`, la señal podría "escapar" al lugar `PROGRAM_PANIC` y el deadlock sería indetectable para el verificador de modelos.

Es indispensable que la traducción "obligue" a la red de Petri a permanecer bloqueada y a no abrir caminos alternativos que podrían ser utilizados por el verificador de modelos para llegar a la conclusión de que la red nunca se bloquea.

```
1 fn main() {
2       let mutex = std::sync::Mutex::new(false);
3       let cvar = std::sync::Condvar::new();
4       let mutex_guard = mutex.lock().unwrap();
5       cvar.notify_one();
6       let _result = cvar.wait(mutex_guard);
7 }
```

Listing 4.10: Un programa básico para mostrar la traducción de variables de condición.

4.8.3. Algoritmos para la traducción de condition variables

Para concluir esta sección, presentaremos un resumen conciso de los algoritmos utilizados en la traducción de condition variables. Posteriormente se detallan también las adiciones necesarias a los algoritmos mutex.

Cuando se encuentra una llamada a `std::sync::Condvar::new`:

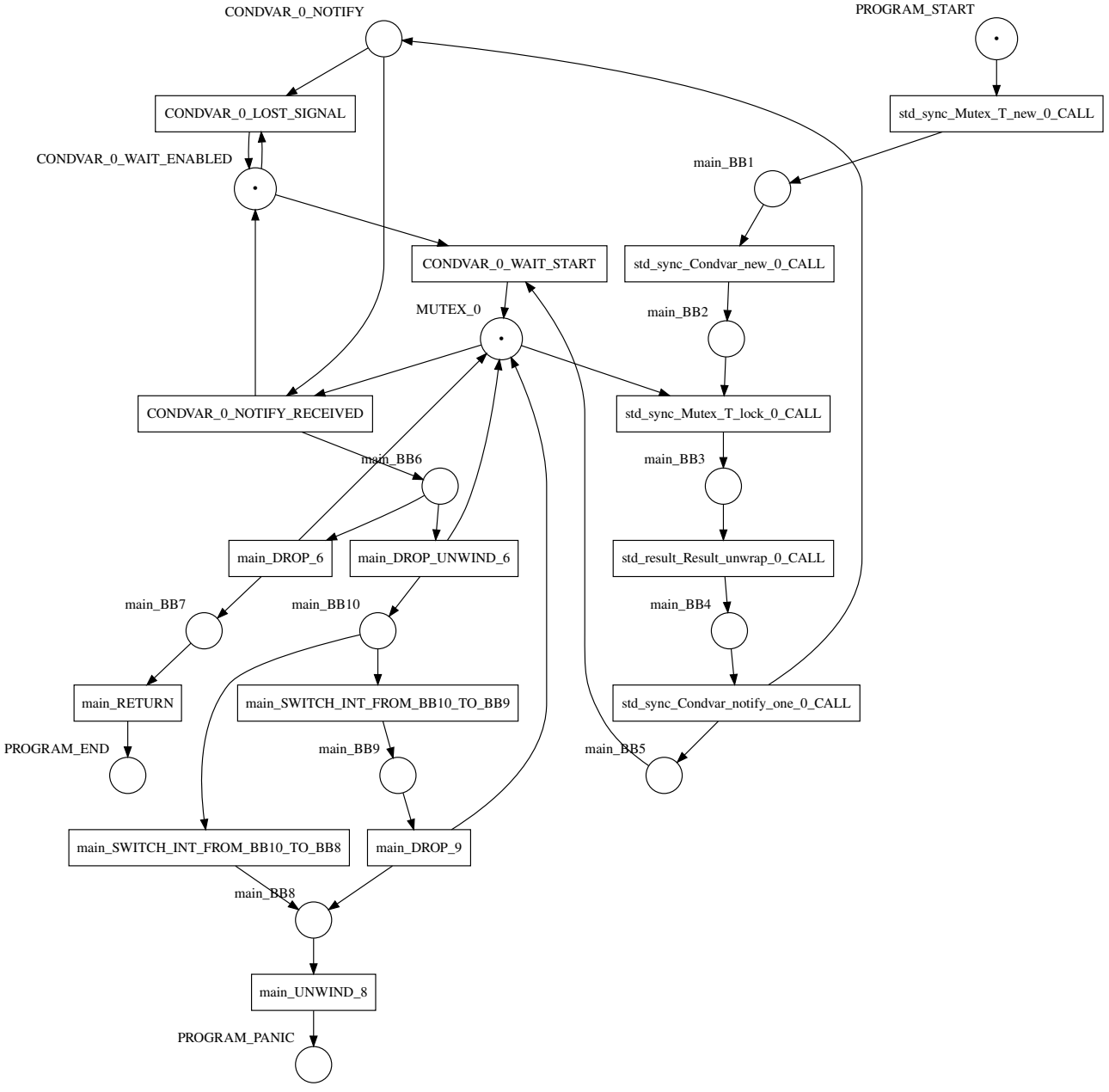

Figura 4.9: El modelo de red de Petri para el programa del Listado 4.10.

1. Traducir la llamada a la función utilizando el modelo visto en la Fig. 4.2.
2. Crear una nueva estructura `Condvar`[43] con un índice para identificarla inequívocamente en toda la red.
3. Vincular el valor de retorno de `std::sync::Condvar::new`, la nueva condition variable, a la estructura `Condvar`.

Cuando se encuentra una llamada a `std::sync::Condvar::notify_one`:

1. Traducir la llamada a la función utilizando el modelo visto en la Fig. 3.2. Ignorar el lugar de limpieza porque, de lo contrario, cualquier llamada podría fallar, lo que equivaldría a que la operación notificar no estuviera presente en el programa, dando lugar a una falsa señal perdida. Esto equivale a suponer que la función `notify_one` nunca falla.
2. Recuperar la auto-referencia `self` a la condition variable sobre la que se llama a la función.
3. Añadir un arco desde la transición que representa la llamada a la función hasta el lugar de notificación de la condition variable referenciada por `self`.

Cuando se encuentra una llamada a `std::sync::Condvar::wait` o `std::sync::Condvar::wait_while`:

1. Ignorar el lugar de limpieza porque, de lo contrario, cualquier llamada puede fallar, lo que equivale a que la operación de espera no está presente en el programa, lo que conduce a un resultado incorrecto. Debemos obligar a la red a "esperar" a que se envíe la señal de notificación. Esto equivale a suponer que las funciones `wait` o `wait_while` nunca fallan.
2. Recuperar la auto-referencia `self` a la condition variable sobre la que se llama a la función.
3. Extraer la guarda de mutex pasada como argumento a la función.
4. Si la condition variable ya está conectada a una llamada a wait, la traducción falla. Esto hace cumplir los Supuestos 1 y 2 vistos al principio de la sección.
5. En caso contrario, conectar los lugares de inicio y fin a las transiciones `wait_start` y `notify_received` respectivamente.
6. Vincular el valor de retorno, la misma guarda de mutex que se pasó como argumento, a la `MutexGuard`.
7. Notificar al traductor que el mutex recibido debe vincularse a este `Condvar`. Para este propósito utilizar la variante enum `PostprocessingTask::LinkMutexToCondvar`. Esta tarea se procesará después de traducir todos los hilos.

Cuando todos los hilos han terminado de traducir, es decir, cuando la cola de hilos por procesar está vacía, el `Translator` entra en un loop para completar las tareas de postprocesamiento por orden de prioridad:

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

1. Crear al principio del loop un vector vacío de referencias a mutex.
2. Saca del `std::collections::BinaryHeap` la tarea con la prioridad más baja. Esta será por diseño una `PostprocessingTask::NewMutex`. Añadir la referencia a mutex al vector.
3. Después de procesar todas las tareas de menor prioridad, el `Translator` tiene referencias a todos los mutex del código. Continuar sacando tareas de la cola de prioridad.
4. Eventualmente, una `PostprocessingTask::LinkMutexToCondvar` es extraída. Vincular cada mutex a la condition variable, lo que crea los lugares `condition_set` y `condition_not_set` para la condición en sí. También conectar las transiciones `deref_mut` a estos lugares para modificar el valor de la condición. Por último, conectar los lugares de la condición a las transiciones de la variable de condición para desactivar el `wait`.

Modificaciones en los algoritmos para el mutex

Como ya se ha dicho, los algoritmos mutex requieren algunas adiciones para realizar con éxito la detección de señales perdidas.

Agregar lo siguiente al manejador de la función `std::sync::Mutex::new`:

1. Notificar al traductor que se ha creado un nuevo mutex. Para este propósito utilizar la variante enum `PostprocessingTask::NewMutex`. Esta tarea se procesará después de traducir todos los hilos.

Cuando se encuentra una llamada a `std::result::Result::<T, E>::unwrap`:

1. Comprobar si la auto-referencia `self` es un mutex o un mutex guard.
2. Traducir la llamada a la función utilizando el modelo visto en la Fig. 3.2. Ignorar el lugar de limpieza porque, de lo contrario, cualquier llamada puede fallar, como si la operación de bloqueo del mutex no estuviera presente en el programa, lo que daría lugar a una falsa señal perdida. Esto equivale a suponer que la función `unwrap` nunca falla cuando se aplica a una variable vinculada a un mutex o a una guarda de mutex.

Cuando se encuentra una llamada a `std::ops::Deref::deref` o `std::ops::DerefMut::deref_mut`:

1. Comprobar si la auto-referencia `self` es un mutex o un mutex guard.
2. Traducir la llamada a la función utilizando el modelo visto en la Fig. 3.2. Ignorar el lugar de limpieza porque, de lo contrario, cualquier llamada puede fallar, como si la operación de bloqueo del mutex no estuviera presente en el programa, lo que daría lugar a una falsa señal perdida. Esto equivale a suponer que las funciones `deref` y `deref_mut` nunca fallan cuando se realiza una desreferencia a una variable vinculada a un mutex o a una guarda mutex.
3. Si el valor está siendo dereferenciado mutablemente (`deref_mut`), extraer el primer argumento pasado a la función: El mutex o el mutex guard. Añadir la transición `deref_mut`

al mutex para conectar la condición con una condition variable específica en la etapa de postprocesamiento.

4. En caso contrario no haga nada. El caso inmutable no necesita ser añadido al mutex.

Ahora debería estar claro para el lector que los algoritmos para la detección de señales perdidas son fundamentalmente de mayor complejidad y deben manejar más casos borde que los de detección de simples deadlocks causados por un uso incorrecto de los mutexes o por llamar a `join` para hilos que nunca terminan.

Cabe mencionar que algunos casos borde surgen debido a la inclusión de la lógica de limpieza de la MIR en el modelo PN. Si en su lugar la implementación omitiera esto, bajo la hipótesis de que las funciones de la biblioteca estándar Rust nunca pueden entrar en pánico, entonces los algoritmos se volverían más sencillos. La Sec. 6.2 se ocupa de la cuestión de no modelar los flujos alternativos de limpieza.

Capítulo 5

Probando la implementación

La inclusión de un capítulo dedicado a las pruebas en la tesis subraya la importancia de este aspecto indispensable del proceso de desarrollo. Las pruebas desempeñan un papel fundamental para garantizar la fiabilidad y correctitud de la implementación del software. Se ha desarrollado un completo conjunto de pruebas para cubrir la extensa funcionalidad y comportamiento del traductor y la biblioteca de redes de Petri.

Las pruebas abarcan múltiples niveles que se dilucidarán en las secciones siguientes. En el nivel más bajo, se realizan pruebas unitarias para verificar la corrección de las estructuras de datos empleadas en el traductor y la biblioteca de redes de Petri. Estas pruebas se dirigen a componentes individuales, examinando a fondo su funcionalidad de forma aislada.

Además de las pruebas unitarias, se ha desarrollado de forma incremental un conjunto de pruebas de integración para evaluar la conformidad del traductor al comportamiento esperado. Estas pruebas consisten en programas de prueba que simulan escenarios sencillos en los que se compara el archivo de salida con los resultados esperados. Esta metodología de pruebas ayuda a descubrir cualquier regresión en el compilador y confirma que el traductor funciona de forma fiable en los casos de uso soportados.

Adicionalmente incorporamos una descripción de cómo generar el MIR y visualizar el resultado de la traducción para asistir en el proceso de depuración. Las herramientas permiten exponer los detalles internos de forma accesible y comprensible.

Más adelante en este capítulo se explica el uso del verificador de modelos LoLA y su integración en el traductor. El verificador de modelos proporciona más características que el conjunto mínimo que se integró en el traductor para responder al mero problema de la detección de deadlocks. Por lo tanto, es beneficioso explorar qué características proporciona el verificador de modelos para depurar la traducción a una PN.

Para finalizar se demuestran las capacidades de la herramienta mediante dos programas de prueba que modelan problemas clásicos de la programación concurrente.

5.1. Pruebas unitarias

Las pruebas unitarias constituyen la base del conjunto de pruebas. La biblioteca de PN y las estructuras de datos utilizadas en el traductor dependen en gran medida de ellas. Al probar meticulosamente las estructuras de datos subyacentes, se pueden identificar y resolver posibles problemas en una fase temprana del ciclo de desarrollo, antes de seguir trabajando en los componentes de nivel superior del traductor.

5.1.1. Biblioteca de redes de Petri

La biblioteca de PN `netcrab` utiliza pruebas unitarias para verificar que la adición de lugares, transiciones y arcos a una red se comporta como se espera. El traductor realiza estas operaciones con frecuencia, por lo que es importante verificarlas. Asimismo se comprueban los iteradores sobre los que se construyen los formatos de exportación.

Por otra parte, cada uno de los tres formatos de exportación (DOT, PNML and LoLA) está acompañado por pruebas unitarias para comprobar que la salida es generada correctamente para los siguientes casos sencillos:

- Red vacía.
- Un red con 5 lugares y 0 transiciones.
- Un red con 5 lugares y 0 transiciones.
- Un red con 5 lugares marcados con diferentes números de fichas.
- Un red con topología en cadena.
- Un red con 1 lugar y 1 transición conectados en bucle.

Estas pruebas ayudaron a solucionar errores relacionados con el formato LoLA. Para ver ejemplos concretos, consulte este commit[1] o este otro[2].

5.1.2. Pila (*Stack*)

La sencilla estructura de datos `Stack`[3] se emplea para implementar la pila de llamadas en el `Translator`, como se vio en la Sec. 4.2. Las pruebas unitarias demuestran los métodos admitidos y testean algunos casos de uso sencillos.

[1] https://github.com/hlisdero/netcrab/commit/5745e0da5d27bd709ef479f45a6d2e75974d3745
[2] https://github.com/hlisdero/netcrab/commit/dbce3f8999ece32e6731527c303a7b59858991f9
[3] https://github.com/hlisdero/cargo-check-deadlock/blob/main/src/data_structures/stack.rs

5.1.3. Contador de mapa hash

De forma análoga a la pila, el `HashMapCounter`[4] contiene algunas pruebas unitarias para verificar que los métodos funcionan según lo previsto. Esta estructura de datos constituye la base del contador de funciones del `Translator` que lleva la cuenta de cuántas veces se ha llamado a cada función para generar un índice incremental unívoco para las etiquetas de las transiciones.

5.2. Integration tests

A continuación examinaremos las pruebas de integración que constituyen la columna vertebral del marco de pruebas del traductor. Actualmente existen dos tipos de pruebas:

- Pruebas de traducción.
- Pruebas de detección de bloqueo.

El proceso de pruebas ha resultado inestimable a lo largo del desarrollo del traductor, permitiendo la detección temprana de errores y regresiones en *rustc*. Apoyándose en las pruebas anteriores y construyendo características de forma incremental, la implementación avanzó con confianza y paso firme hacia adelante. Las capacidades de prueba que ofrece Rust, incluido su soporte para pruebas unitarias y de integración, han sido valiosas para garantizar la calidad y fiabilidad del traductor.

5.2.1. Pruebas de traducción

En las pruebas de traducción, se procesa un programa dado *sin* realizar el análisis de bloqueo. Como resultado, se generan tres archivos de texto que contienen el modelo en formatos DOT, PNML y LoLA. Posteriormente estos archivos se comparan con la salida esperada, que se almacena en el repositorio y sirve también como documentación.

El resultado esperado se verificó manualmente con la ayuda de las herramientas presentadas en la Sec. 5.3. Se committeó en el repositorio cuando el traductor pudo superar la prueba por primera vez. Si se produce una regresión en *rustc*, los archivos de salida esperada se actualizan en consecuencia. Esto ha ocurrido algunas veces en el pasado. Véase, por ejemplo, este commit[5] o este[6].

[4] https://github.com/hlisdero/cargo-check-deadlock/blob/main/src/data_structures/hash_map_counter.rs

[5] https://github.com/hlisdero/cargo-check-deadlock/commit/881a3873a3b060e70bc727f670f9426d14327fa2

[6] https://github.com/hlisdero/cargo-check-deadlock/commit/b032fa3cc13e631950a802dcd3f755c548afde86

5.2.2. Pruebas de detección de bloqueo

Las pruebas de detección de deadlock se acercan más a una prueba *end-to-end* del traductor. Generan el archivo en formato LoLA para el programa de prueba y ordenan al traductor que realice el análisis de deadlock. El resultado se contrasta entonces con el comportamiento conocido del programa de prueba, es decir, si se bloquea o no se bloquea. Si LoLA produce un resultado incorrecto, entonces la prueba falla. En tal caso, debe analizarse el modelo PN para encontrar la fuente del error. Consulte la sección 5.3.3 para obtener más detalles sobre cómo abordar esta cuestión.

Observe el Listado 5.1 que contiene el contenido del archivo `.lola` para el programa representado en el Listado 4.4. Este es el formato de archivo que requiere el verificador de modelos. Es relativamente más sencillo que PNML, el cual está basado en XML.

Debido a su considerable longitud, es el único ejemplo del formato LoLA en esta tesis. Se lo incluye aquí por razones de completitud. El repositorio contiene varios ejemplos más, todos de los cuales se utilizan en las pruebas de integración.

```
1  PLACE
2      MUTEX_0,
3      PROGRAM_END,
4      PROGRAM_PANIC,
5      PROGRAM_START,
6      main_BB1,
7      main_BB2,
8      main_BB3,
9      main_BB4,
10     main_BB5,
11     main_BB6,
12     main_BB7;
13
14 MARKING
15     MUTEX_0 : 1,
16     PROGRAM_END : 0,
17     PROGRAM_PANIC : 0,
18     PROGRAM_START : 1,
19     main_BB1 : 0,
20     main_BB2 : 0,
21     main_BB3 : 0,
22     main_BB4 : 0,
23     main_BB5 : 0,
24     main_BB6 : 0,
25     main_BB7 : 0;
26
```

```
TRANSITION main_DROP_3
  CONSUME
    main_BB3 : 1;
  PRODUCE
    MUTEX_0 : 1,
    main_BB4 : 1;
TRANSITION main_DROP_4
  CONSUME
    main_BB4 : 1;
  PRODUCE
    MUTEX_0 : 1,
    main_BB5 : 1;
TRANSITION main_DROP_6
  CONSUME
    main_BB6 : 1;
  PRODUCE
    MUTEX_0 : 1,
    main_BB7 : 1;
TRANSITION main_DROP_UNWIND_3
  CONSUME
    main_BB3 : 1;
  PRODUCE
    MUTEX_0 : 1,
    main_BB6 : 1;
TRANSITION main_RETURN
  CONSUME
    main_BB5 : 1;
  PRODUCE
    PROGRAM_END : 1;
TRANSITION main_UNWIND_7
  CONSUME
    main_BB7 : 1;
  PRODUCE
    PROGRAM_PANIC : 1;
TRANSITION std_sync_Mutex_T_lock_0_CALL
  CONSUME
    MUTEX_0 : 1,
    main_BB1 : 1;
  PRODUCE
    main_BB2 : 1;
TRANSITION std_sync_Mutex_T_lock_1_CALL
  CONSUME
    MUTEX_0 : 1,
```

```
70      main_BB2 : 1;
71    PRODUCE
72      main_BB3 : 1;
73  TRANSITION std_sync_Mutex_T_new_0_CALL
74    CONSUME
75      PROGRAM_START : 1;
76    PRODUCE
77      main_BB1 : 1;
```

Listing 5.1: La salida LoLA para el programa del Listado 4.4.

5.2.3. Estructura de ficheros de las pruebas

Los programas de prueba se encuentran en la carpeta `examples/programs`. Para cada programa de prueba, hay una carpeta en `examples/results` que contiene los tres archivos `net.dot`, `net.pnml` y `net.lola`.

Los tests se agrupan en categorías:

- Basic: Para programas básicos como "¡Hola, mundo!z una simple calculadora aritmética.
- Condvar: Para programas relativos a condition variables.
- Function call: Para los programas que prueban los diferentes tipos de llamadas de función vistos en la Sec. 4.2.
- Mutex: Para programas hacen uso de mutexes.
- Statement: Para programas que comprueban construcciones específicas como una `match`, un bucle infinito, una `Option`, una llamada a `panic!`, o `std::process::abort`.
- Thread: Para programas en los que intervienen varios hilos.

La estructura de las carpetas en `examples/` imita la estructura de archivos de las pruebas de integración en `tests/`. Como de costumbre, todo el conjunto de pruebas puede ejecutarse con el comando `cargo test`.

5.2.4. Implementación de las pruebas

Las pruebas de integración se basan en los crates `assert_cmd`, `assert_fs` y `predicates`. La idea de verificar la salida del programa invocando directamente al binario se tomó de un libro especializado en la construcción de aplicaciones CLI en Rust [Rust CLI WG, 2023, Chap. 1.6]. Fue un recurso útil también para experimentar con `clap` para parsear los argumentos.

Además, las pruebas de integración utilizan un submódulo compartido [Klabnik and Nichols, 2023, Chap. 11.3] que contiene dos cómodas macros que nos ahorran escribir casi todo el código boilerplate. Estas macros se definieron utilizando [Wirth and Keep, 2023] como referencia principal y con la inspiración proporcionada por [Oaten, 2023].

El Listado 5.2 muestra la macro responsable de generar las pruebas de traducción, mientras que el Listado 5.3 ofrece un ejemplo de cómo se aplica en el repositorio. En aras de la completitud, el Listado 5.4 representa la función utilizada para las pruebas de traducción.

```
macro_rules! generate_tests_for_example_program {
    ($program_path:literal, $result_folder_path:literal) => {
        #[test]
        fn generates_correct_output_files() {
            super::utils::assert_output_files($program_path, $result_folder_path);
        }
    };
}
```

Listing 5.2: La macro que genera las pruebas de traducción.

5.3. Visualizando del resultado

La visualización del resultado es esencial para comprender el resultado de la detección del deadlock. Por esta razón invertimos tiempo en investigar diferentes formas de lograr el mismo resultado, con y sin una instalación local necesaria para que fuera lo más fácil de usar posible.

Estas instrucciones también se pueden encontrar en el `README`[7] del repositorio.

5.3.1. Localmente

Para ver la representación MIR del código fuente, se puede compilar el código con la flag correspondiente: `rustc --emit=mir <path_to_source_code>`

Es importante tener en cuenta que la cadena de herramientas nocturna puede producir MIR diferentes en comparación con la versión estable del compilador. Remitimos al lector a la Sec. 3.2.2 para más información.

Para graficar una red en formato `.dot`, instale la herramienta dot siguiendo las instrucciones de la página web de GraphViz[8].

[7] https://github.com/hlisdero/cargo-check-deadlock/blob/main/README.md
[8] https://graphviz.org/download/

```rust
mod utils;

mod calculator {
    super::utils::generate_tests_for_example_program!(
        "./examples/programs/basic/calculator.rs",
        "./examples/results/basic/calculator/"
    );
}

mod greet {
    super::utils::generate_tests_for_example_program!(
        "./examples/programs/basic/greet.rs",
        "./examples/results/basic/greet/"
    );
}

mod hello_world {
    super::utils::generate_tests_for_example_program!(
        "./examples/programs/basic/hello_world.rs",
        "./examples/results/basic/hello_world/"
    );
}
```

Listing 5.3: El contenido del archivo `basic.rs` que enumera todas las pruebas de traducción de la categoría básica.

```rust
pub fn assert_output_files(source_code_file: &str, output_folder: &str) {
    let mut cmd = Command::cargo_bin("cargo-check-deadlock").expect("Command not found");

    // Current workdir is always the project root folder
    cmd.arg("check-deadlock")
        .arg(source_code_file)
        .arg(format!("--output-folder={output_folder}"))
        .arg("--dot")
        .arg("--pnml")
        .arg("--filename=test")
        .arg("--skip-analysis");

    cmd.assert().success();

    for extension in ["lola", "dot", "pnml"] {
        let output_path = PathBuf::from(format!("{output_folder}test.{extension}"));
        let expected_output_path = PathBuf::from(format!("{output_folder}net.{extension}"));

        let file_contents =
            std::fs::read_to_string(&output_path).expect("Could not read output file to
            ↪  string");

        let expected_file_contents = std::fs::read_to_string(&expected_output_path)
            .expect("Could not read file with expected contents to string");

        if file_contents != expected_file_contents {
            panic!(
                "The contents of {} do not match the contents of {}",
                output_path.to_string_lossy(),
                expected_output_path.to_string_lossy()
            );
        }

        std::fs::remove_file(output_path).expect("Could not delete output file");
    }
}
```

Listing 5.4: La función que verifica el contenido de los archivos de salida.

Ejecute `dot -Tpng net.dot -o outfile.png` para generar una imagen PNG a partir del archivo de salida `.dot`.

Ejecute `dot -Tsvg net.dot -o outfile.svg` para generar una imagen SVG a partir del archivo de salida `.dot`.

Encontrará más información y otros posibles formatos de imagen en la documentación[9].

5.3.2. En línea

Para ver la representación MIR del código fuente, se puede utilizar el Rust Playground[10].

Debe seleccionar la opción "MIR" en lugar de "Run" en el menú desplegable. Tenga en cuenta que debe utilizarse la versión nightly en lugar de la versión estable de *rustc*.

Para graficar un resultado DOT dado, la herramienta Graphviz Online[11] ofrece una alternativa fiable a las herramientas instaladas localmente. Existen alternativas como Edotor[12] o SketchViz[13].

5.3.3. Depuración (*Debugging*)

El programa soporta los flags de verbosidad definidas en el crate `clap_verbosity_flag`[14]. A modo de ejemplo, ejecutar el programa con la bandera `-vvv` imprime mensajes de *debug* que pueden ser útiles para localizar qué línea de la MIR no se está traduciendo correctamente.

Entonces deberá invocar la herramienta del siguiente modo:

```
cargo check-deadlock <path_to_program>/rust_program.rs -vvv
```

El comprobador de modelos LoLA admite la impresión de una "witness path" que muestra una secuencia de disparos de transición que conducen a un deadlock. Esto es extremadamente útil cuando se extiende la funcionalidad del traductor y el análisis de la red de Petri no coincide con el resultado esperado para un programa dado.

En el repositorio se incluye un práctico script llamado `run_lola_and_print_witness_path.sh` para imprimir la ruta testigo de un archivo `.lola`. La Fig. 5.1 ilustra el resultado de ejecutar el script en el archivo mostrado en el Listado 4.5.

[9] https://graphviz.org/doc/info/command.html
[10] https://play.rust-lang.org/
[11] https://dreampuf.github.io/GraphvizOnline/
[12] https://edotor.net/
[13] https://sketchviz.com/new
[14] https://docs.rs/clap-verbosity-flag/latest/clap_verbosity_flag/

```
lola found in $PATH.
lola: NET
lola:   reading net from examples/results/mutex/double_lock_deadlock/net.lola
lola:   finished parsing
lola:   closed net file examples/results/mutex/double_lock_deadlock/net.lola
lola:   20/65536 symbol table entries, 0 collisions
lola:   preprocessing...
lola:   finding significant places
lola:   11 places, 9 transitions, 9 significant places
lola:   computing forward-conflicting sets
lola:   computing back-conflicting sets
lola:   14 transition conflict sets
lola: TASK
lola:   read: EF ((DEADLOCK AND (PROGRAM_END = 0 AND PROGRAM_PANIC = 0)))
lola:   formula length: 59
lola:   checking reachability
lola:   Planning: workflow for reachability check: search (--findpath=off)
lola: STORE
lola:   using a bit-perfect encoder (--encoder=bit)
lola:   using 36 bytes per marking, with 0 unused bits
lola:   using a prefix tree store (--store=prefix)
lola: SEARCH
lola:   using reachability  graph (--search=depth)
lola:   using reachability preserving stubborn set method with insertion algorithm (--stubborn=tarjan)
lola: RUNNING
lola: RESULT
lola:   result: yes
lola:   The predicate is reachable.
lola:   3 markings, 2 edges
lola:   print witness path (--path)
lola:   writing witness path to stdout
std_sync_Mutex_T_new_0_CALL
std_sync_Mutex_T_lock_0_CALL
lola:   closed witness path file stdout
```

Figura 5.1: Salida de la ruta testigo generada por LoLA para el programa del Listado 4.4.

5.4. Integrando LoLA a la solución

Como se indica en la Sec. 2.5.3, LoLA es el verificador de modelos elegido para esta tesis. Actúa como un backend que se encarga de verificar la ausencia de deadlocks. Integrarlo no fue, por desgracia, trivial, como se detallará en los siguientes párrafos.

5.4.1. Compilación

En primer lugar, la compilación a partir del código fuente no funcionaba en el hardware del que se disponía. Fue necesario realizar cambios en el código, ya que las versiones más recientes del compilador de C++ tienden a ser más estrictas y rechazan o generan advertencias para el

código que antes se aceptaba. Además, una de las dependencias, kimwitu++[15], debe compilarse también a partir del código fuente ya que no está empaquetado para las distribuciones de Linux.

Para conservar en el futuro una copia funcional del verificador de modelos, indispensable para realizar el análisis de deadlocks, se ha creado una réplica[16] en GitHub en la que se ofrecen instrucciones detalladas a los usuarios. Con ello se pretende que la instalación desde el código fuente sea lo más sencilla posible.

5.4.2. Invocación del verificador de modelos

La segunda dificultad es que LoLA se compila como un ejecutable, no como una biblioteca, por lo que nuestra herramienta no podía enlazar con él. En su lugar, nos vemos obligados a ejecutar el binario desde `cargo-check-deadlock` para ejecutar el comprobador de modelos pasando los argumentos correctos[17]. El ejecutable de LoLA forma parte del repositorio porque es necesario para ejecutar las pruebas de integración en la pipeline CI/CD implementada en GitHub Actions. Un usuario también puede copiar este ejecutable para instalar LoLA en lugar de compilarlo desde cero.

A fin de cuentas el usuario es responsable de instalar el comprobador de modelos por separado para permitir que nuestra herramienta lo invoque. Un script `copy_lola_executable_to_cargo_home.sh` incluido en el repositorio facilita la tarea de copiar el archivo a una carpeta que ya esté en el \$PATH. También consideramos otras posibilidades, pero ninguna resultó viable:

In the end, the user is responsible for installing the model checker separately to allow our tool to invoke it. A script `copy_lola_executable_to_cargo_home.sh` included in the repository facilitates the task of copying the file to a folder that is already in the \$PATH. We also considered other possibilities but none were feasible:

1. Usar build scripts (`build.rs`) como se describe en el Cargo Book [Rust Project, 2023a, Chap. 3.8].
2. Modificar LoLA para convertirlo en una biblioteca.
3. Mover un ejecutable precompilado a la carpeta de instalación cuando se ejecute `cargo install`.
4. Definir LoLA como un binario en el `Cargo.toml` [Rust Project, 2023a, Chap 3.2.1] y que, con un poco de suerte, sea movido al directorio cargo bin.
5. Definir LoLA como un *example* en el `Cargo.toml` [Rust Project, 2023a, Chap 3.2.1] y que, con un poco de suerte, sea movido al directorio cargo bin.
6. Utilizar una herramienta de build de uso general como `make`.

[15] https://www.nongnu.org/kimwitu-pp/
[16] https://github.com/hlisdero/lola
[17] https://github.com/hlisdero/cargo-check-deadlock/blob/main/src/model_checker/lola.rs

En el proceso de resolución de este segundo problema, aprendimos que cargo es adecuado principalmente para tratar con dependencias expresadas como crates de Rust, que deben compilarse cuando se instalan, no con archivos/artefactos arbitrarios. En resumen, no está pensada para ser una herramienta de compilación de uso general como `make`.

5.4.3. Expresar la propiedad a comprobar

El tercer reto es encontrar una fórmula de Computational Tree Logic* (CTL*) para ordenar a LoLA que busque los deadlocks. Afortunadamente, podemos reutilizar la fórmula encontrada en [Meyer, 2020]:

$$\text{EF (DEADLOCK AND (PROGRAM\_END} = 0 \text{ AND PROGRAM\_PANIC} = 0))$$

La fórmula representa la propiedad a comprobar. Cabe destacar que no todos los deadlocks son interesantes para nuestro análisis. Nuestro objetivo es identificar los casos de deadlocks en los que la ejecución del programa se bloquea *inesperadamente*. Este escenario se alinea con una red de Petri muerta como se ve en la Definición 14 en la que no está habilitada ninguna transición y la red alcanza un estado final. Sin embargo, debemos actuar con cautela, ya que hay casos en los que se *espera* que la PN esté muerta, como cuando el programa termina o entra en pánico. Estos son estados normales en los que la ejecución se detiene. Por lo tanto, si llegamos al lugar `PROGRAM_END` o `PROGRAM_PANIC`, la ejecución ha sido satisfactoria, no se trata de un deadlock en el sentido de la Sec. 1.4.1. En conclusión, excluimos los lugares `PROGRAM_END` y `PROGRAM_PANIC` exigiendo que *no estén marcados* para que se cumpla la condición de deadlock. Esto se expresa mediante el "$= 0$" en la fórmula CTL*.

Por último, debemos considerar el aspecto temporal. Para especificar que nuestra propiedad de estado se cumple eventualmente y encontrar un camino relevante, podemos utilizar los operadores "EF" en combinación. La "F" significa "eventualmente" y la "E" es el cuantificador de camino existencial [Meyer, 2020]. Así que la fórmula se lee como:

"Existe eventualmente un camino tal que DEADLOCK (ninguna transición puede dispararse) y el lugar PROGRAM_PANIC tiene cero fichas y el lugar PROGRAM_END tiene cero fichas"

Se pueden construir otras fórmulas para comprobar otras propiedades. Para este trabajo, tomamos esta fórmula como dada y dejamos al usuario la posibilidad de comprobar otras propiedades si así lo desea. Para una breve introducción a CTL*, véase [Meyer, 2020].

5.5. Notable test programs

To wrap up this chapter, we introduce two noteworthy test programs that illustrate the current capabilities of the tool developed in this thesis. Our intention is to inspire others to contribute to this project or, at the very least, generate interest in the field of model checking.

First, Listing 5.5 showcases a simple version of the famous Dining Philosophers Problem proposed by Dijkstra. This version, affectionately nicknamed "Dating Philosophers", has only two philosophers and two forks on the table. A mutex needs to be locked to access each fork. When the philosophers try to grab both forks to eat, the program deadlocks, which is easy to verify by inspection. This deadlock is successfully detected by the tool. Moreover, a more complex version with 5 philosophers, for which the deadlock is also detected, is included in the repository[18]. It was omitted here due to the space constraints.

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let fork0 = Arc::new(Mutex::new(0));
    let fork1 = Arc::new(Mutex::new(1));

    let philosopher0 = {
        let left_fork = fork0.clone();
        let right_fork = fork1.clone();
        thread::spawn(move || {
            let _left = left_fork.lock().unwrap();
            let _right = right_fork.lock().unwrap();
        })
    };

    let philosopher1 = {
        let left_fork = fork1.clone();
        let right_fork = fork0.clone();
        thread::spawn(move || {
            let _left = left_fork.lock().unwrap();
            let _right = right_fork.lock().unwrap();
        })
    };

    // Wait for all threads to finish
    philosopher0.join().unwrap();
    philosopher1.join().unwrap();
}
```

Listing 5.5: A reduced version of the dining philosophers problem that deadlocks.

Second, observe the program in Listing 5.6. It models the classical producer-consumer problem.

[18] https://github.com/hlisdero/cargo-check-deadlock/blob/main/examples/programs/thread/dining_philosophers.rs

It uses a condition variable and a buffer with capacity for a single element. The access to the buffer is protected by a mutex. The producer generates 10 elements sequentially and the consumer processes them as they become available. The tool successfully verifies the absence of deadlock in the program.

```rust
use std::sync::{Arc, Condvar, Mutex};
use std::thread;

fn main() {
    let buffer = Arc::new((Mutex::new(0), Condvar::new(), Condvar::new()));

    let producer_buffer = buffer.clone();
    let consumer_buffer = buffer.clone();

    let _producer = thread::spawn(move || {
        for i in 1..10 {
            let (lock, cvar_producer, cvar_consumer) = &*producer_buffer;
            let mut buffer = lock.lock().unwrap();

            while *buffer != 0 {
                buffer = cvar_producer.wait(buffer).unwrap();
            }

            *buffer = i;
            println!("Produced: {}", i);

            cvar_consumer.notify_one();
        }
    });

    let _consumer = thread::spawn(move || loop {
        let (lock, cvar_producer, cvar_consumer) = &*consumer_buffer;
        let mut buffer = lock.lock().unwrap();

        while *buffer == 0 {
            buffer = cvar_consumer.wait(buffer).unwrap();
        }

        let item = *buffer;
        *buffer = 0;
        println!("Consumed: {}", item);

        cvar_producer.notify_one();
    });
}
```

Listing 5.6: A solution to the producer-consumer problem.

Capítulo 6

Trabajos futuros

6.1. 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. 6.1 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.

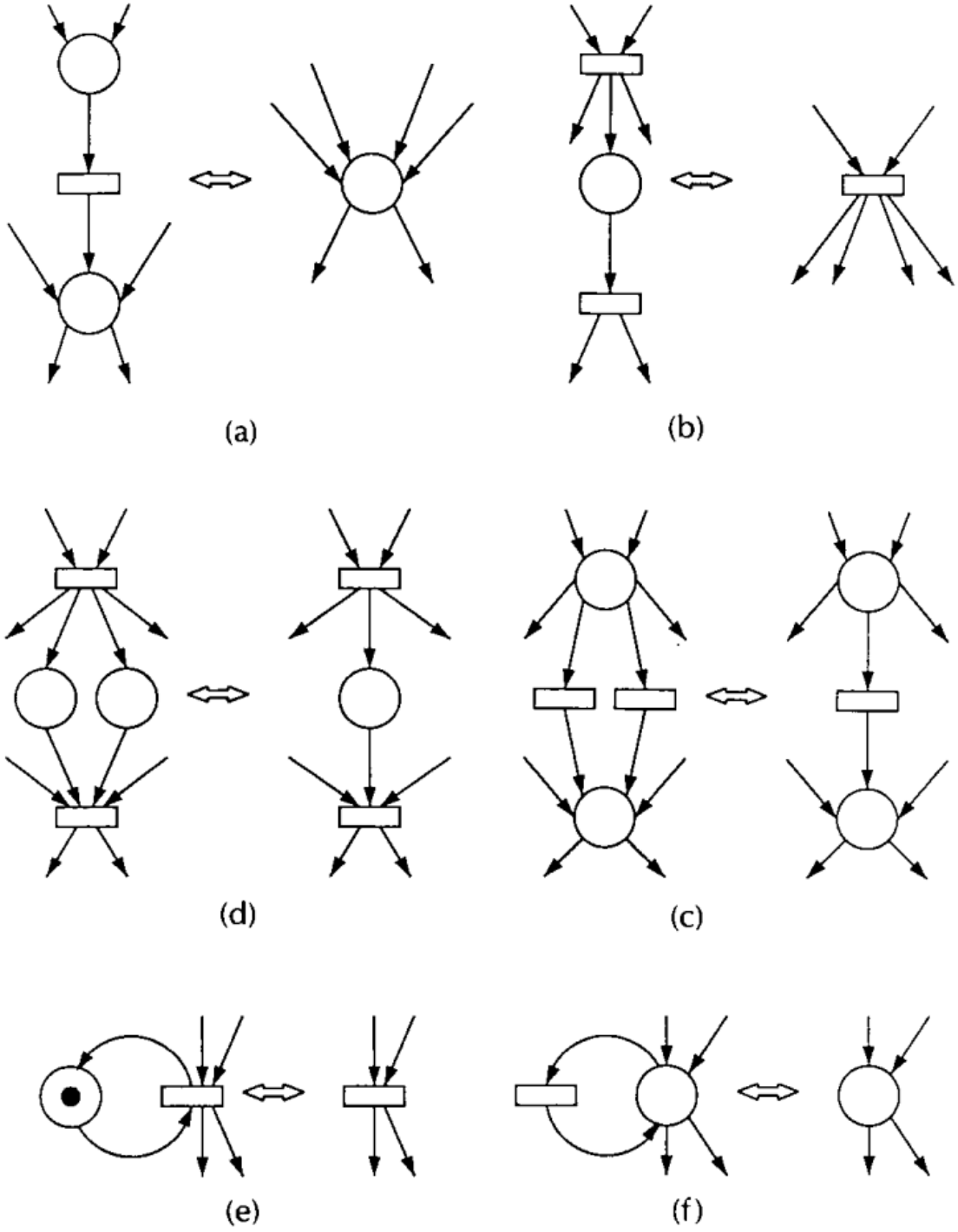

Figura 6.1: The reduction rules presented in Murata's paper.

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.

6.2. 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

6.3. 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.

6.4. 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.

6.5. 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.

6.6. 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].

Capítulo 7

Trabajos relacionados

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 language designed for specifying distributed systems. In this paper, the program is made of

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

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.

Capítulo 8

Conclusiones

This thesis has explored the translation of Rust programs into Petri net models for the purpose of deadlock and missed signal detection. Throughout the study, various aspects of the translation process have been examined, including the handling of function calls, threads, mutexes, and condition variables. The translator we developed has demonstrated its capability to accurately capture the concurrency and synchronization behavior of rather simple Rust programs.

The translation approach presented in this thesis has shown promising results, successfully modeling and detecting deadlocks in a range of test programs, comprising even two classical problems of concurrent programming. By harnessing the expressive power of Petri nets, the translator provides a visual representation of program behavior, facilitating the identification of potential synchronization issues. Most importantly, the translation produces a model that can be analyzed by a myriad of model checking tools, leveraging the existing academic work to bring solutions to industry problems. The incorporation of a succinct model for condition variables enhances the modeling capabilities and enables the detection of missed signals, which are a more intricate class of deadlock in concurrent systems.

Moving forward, there are several avenues for future research and improvement. One potential direction is the exploration of more complex programs and real-world applications to evaluate the scalability and effectiveness of the translation approach. Additionally, further refinement and optimization of the translation algorithms could enhance the efficiency of the analysis, specially higher-level models that would allow modeling the memory more effectively.

Overall, this thesis has made a significant contribution by developing a translator that bridges the gap between Rust programs and Petri nets. The insights gained from this research have shed light on the challenges and opportunities in modeling and analyzing concurrent systems at compile-time. Ideally, a programming language whose compiler detects concurrency problems would be a godsend for many applications. Building on the strengths of Petri nets, this possibility could be advanced further in the Rust programming language.

On a different note, the contribution of this thesis extends beyond the immediate benefits of

the proposed translator and its capabilities. By providing a solid, well-documented base for the translation of Rust programs into Petri nets, this work aims to make a meaningful contribution to the Rust community as a whole. It serves as a stepping stone for future endeavors, offering a reliable foundation upon which other tools and research projects can be built. It opens up new possibilities for exploring the analysis and verification of concurrent Rust programs using Petri nets. This, in turn, has the potential to drive further advancements in the field, stimulating innovation and promoting a deeper understanding of concurrent programming in Rust. With its comprehensive documentation and clear implementation, the translator not only facilitates immediate use but also serves as a valuable resource for those interested in studying or extending the translation techniques employed. Ultimately, this work aspires to ignite curiosity and inspire further contributions to the Rust ecosystem, fostering collaboration and growth in the community.

Bibliografía

[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.

[Oaten, 2023] Oaten, T. (2023). Rust's Witchcraft. https://www.youtube.com/watch?v=MWRPYBoCEaY. Accessed on 2023-04-08.

[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 CLI WG, 2023] Rust CLI WG (2023). Command Line Applications in Rust. https://rust-cli.github.io/book/. Accessed on 2023-06-08.

[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 Cargo Book. https://doc.rust-lang.org/cargo/. Accessed on 2023-06-08.

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

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

[Rust Project, 2023d] Rust Project (2023d). The rustup Book. https://rust-lang.github.io/rustup/index.html. Accessed on 2023-05-02.

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

[Wirth and Keep, 2023] Wirth, L. and Keep, D. (2023). The Little Book of Rust Macros. https://veykril.github.io/tlborm/introduction.html. Accessed on 2023-06-08.

[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*.