
UNIVERSIDAD DE Buenos Aires

TESIS DE GRADO DE INGENIERÍA EN INFORMÁTICA

# Detección de bloqueos en tiempo de compilación en Rust 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

# Contenido

# Lista de figuras

# Lista de anuncios

# Acrónimos

**ART** Android Runtime
**AST** árbol de sintaxis
abstracta **BB** bloques
básicos
**CFG** gráfico de flujo de
control **CLI** interfaz de línea
de comandos **CPN** Redes de
Petri coloreadas **CPU** unidad
central de proceso
**Creol** Lenguaje reflexivo concurrente orientado a objetos
**CTL*** Computational Tree Logic*
**DBMS** Sistemas de gestión de bases de
datos FSM Máquina de estados finitos
**HIR** Representación intermedia de alto nivel
**IR** representación intermedia ISA
arquitectura del conjunto de
instrucciones JIT justo a tiempo
**LHS** lado izquierdo
**LIFO** último en entrar, primero en salir
**LoLA** Analizador de redes de Petri de bajo nivel
Optimización del tiempo de enlace **LTO**
Concurso de verificación de modelos de **MCC**
**MIR** Representación intermedia
**NT-PN** Redes de Petri de transición no determinista

**OOM** fuera de memoria

Sistema operativo **OS**

**Redes P/T** redes de lugar/transición

**PIPE2** Editor de redes de Petri independiente de la plataforma 2

Redes de Petri **PN**

**PNML** Lenguaje de marcado de redes de Petri

**RAG** Gráfico de asignación de recursos

**RAII** Adquisición de recursos es inicialización

**RFC** Solicitudes de comentarios

**RHS** lado derecho

**TAPAAL** Herramienta para la verificación de redes de

Petri de arco temporizado **THIR** Representación

intermedia tipificada de alto nivel **TWF** transacción-

espera-para

Comportamiento indefinido **UB**

**WASM** WebAssembly

Lenguaje de marcado extensible **XML**

# Resumen


## 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.

## Detección de bloqueos en tiempo de compilación en Rust mediante redes de Petri

Esta tesis de licenciatura presenta una herramienta de análisis estático para la detección de bloqueos 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. A continuación, la red de Petri se obtiene como salida en uno o varios de l o s siguientes formatos: DOT, Petri Net Markup Language o LoLA. Posteriormente, se utiliza el verificador de modelos LoLA para probar exhaustivamente la ausencia de bloqueos y señales perdidas. La herramienta se publica como un complemento para el gestor de paquetes *cargo* y la totalidad del código fuente está disponible en GitHub[1][2]. La herramienta demuestra de forma práctica la posibilidad de ampliar el compilador de Rust con una pasada adicional para detectar más clases de errores en tiempo de compilación.


[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 a prueba de 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 y las señales perdidas cuando se utilizan variables de condición, 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 comprobación de modelos. Señalaremos el potencial aún sin explotar que ofrece 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 (RP) 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 lugares ni transiciones, preservando así la propiedad bipartita.

Los lugares pueden contener cero o más 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.

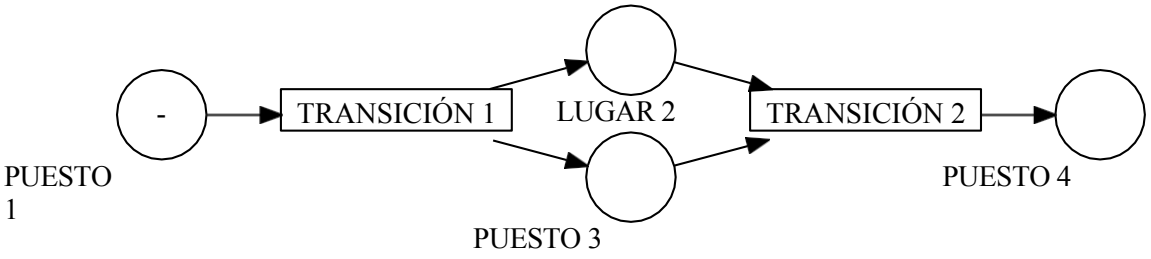


Figura 1.1: Ejemplo de una red de Petri. El LUGAR 1 contiene una ficha.

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 sucesivos disparos.

El disparo de las transiciones habilitadas no es determinista, es decir, se disparan aleatoriamente 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 punto muerto 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 potenciales en un sistema, optimizar su rendimiento y diseñar e implementar sistemas de forma más eficaz.

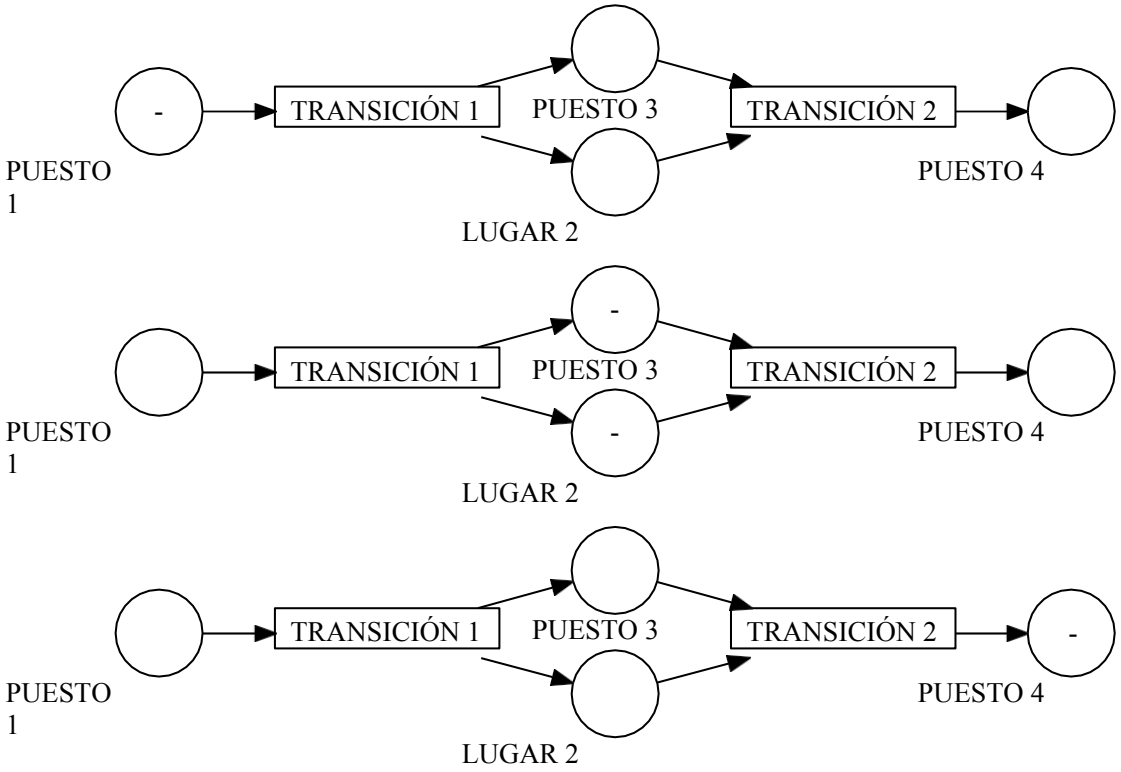


Figura 1.2: Ejemplo de disparo de una transición: La transición 1 se dispara primero, luego se dispara la transición 2.

También pueden utilizarse para modelar procesos industriales [Van der Aalst, 1994], para validar requisitos de software expresados como casos de uso [Silva y 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 bloqueos 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, ponderado 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].

**Definición 1: Red de Petri**

*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),* $W : F \leftarrow \{1, 2, 3, \ldots\}$ *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 fichas*. Pictóricamente, denotamos su colocando *l* puntos negros (fichas) en el lugar *p*. El componente *p* de *M* , denotado por *M* (*p*), es el número de fichas en el lugar *p*.

Una definición alternativa de las redes de Petri utiliza *bolsas* 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, Definición 2.3].

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

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

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

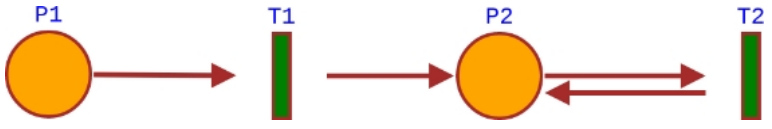


Figura 1.3: Ejemplo de una pequeña red de Petri que contiene un bucle propio.

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

**Definición 2: Bucle propio**

*Un nodo de lugar p y un nodo de transición t definen un bucle propio 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 propios, que se denominan *puras*.

**Definición 3: Red de Petri pura**

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

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

**Definición 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,*

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

### 1.1.3 Cocción de transición

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.

**Definición 5: Regla de disparo de la transición**

*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 está marcado con al menos* $W(p, t)$ *fichas, donde* $W(p, t)$ *es el peso del arco que va de p a 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 fichas* $W(t, p)$ *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 intercalado de paralelismo*, 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.

**Definición 6: Transición de origen**

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

**Definición 7: Transición de sumidero**

*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 de 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 las fichas 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 comedores, se remite al lector a [Peterson, 1981, Cap. 3] y [Reisig, 2013].

#### Máquinas de estado finito

Las máquinas de estados finitos (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

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.

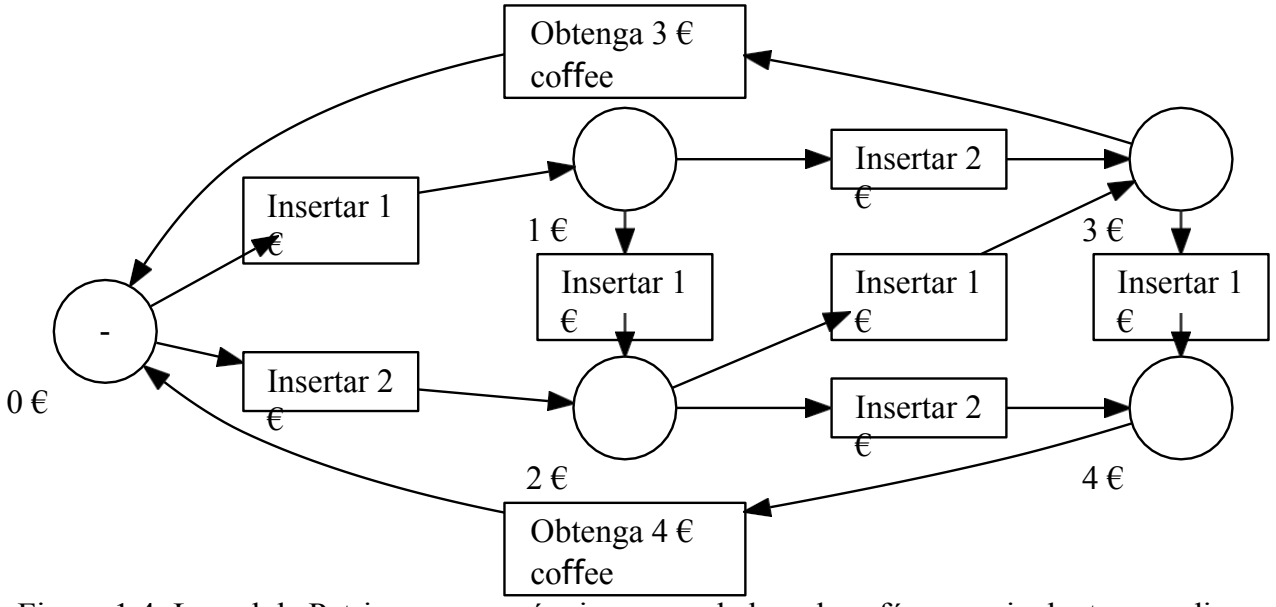


Figura 1.4: La red de Petri para una máquina expendedora de café, es equivalente a un diagrama de estados.

Las transiciones representan la inserción de una moneda del valor etiquetado, por ejemplo, "Inserte una moneda de 1 euro". 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.

**Definición 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. Esto se ve en el lugar inicial de la Fig. 1.4, donde el usuario debe seleccionar qué moneda introducir primero.

### Actividades paralelas

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.

La transición "Fork" se disparará antes que "Task 1" y "Task 2" y que "Join" sólo se disparará después de

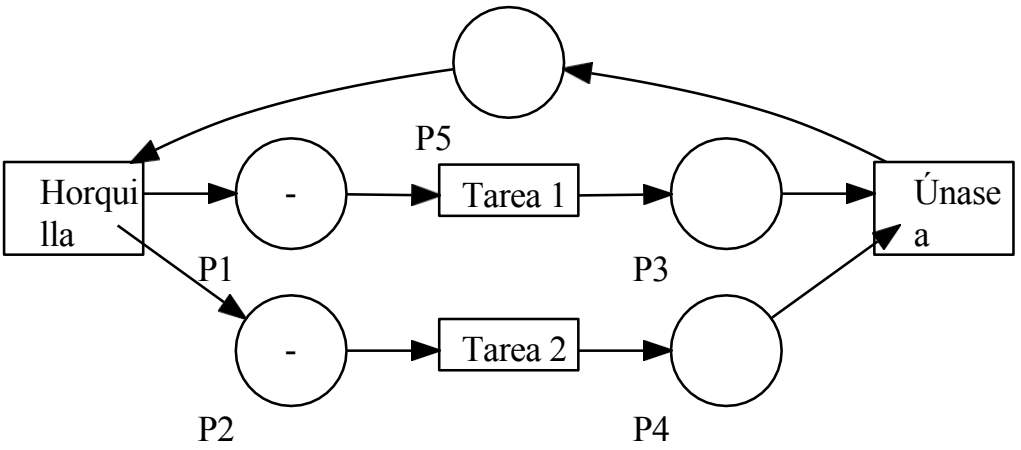


Figura 1.5: La red de Petri que representa dos actividades paralelas en forma de bifurcación.

ambas tareas se completan. Pero tenga en cuenta que el orden en que se ejecutan la "Tast 1" y la "Tast 2" 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.

**Definición 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).

**Definición 10: Grafos marcados**

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

### 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 tiempo de espera.

Se podría incorporar al modelo un tiempo de espera para la operación de envío añadiendo una transición $_{ttimeout}$ con aristas de "Esperar ACK" a "Listo para enviar". Esto mapea la decisión entre recibir el acuse de recibo y el tiempo de espera.

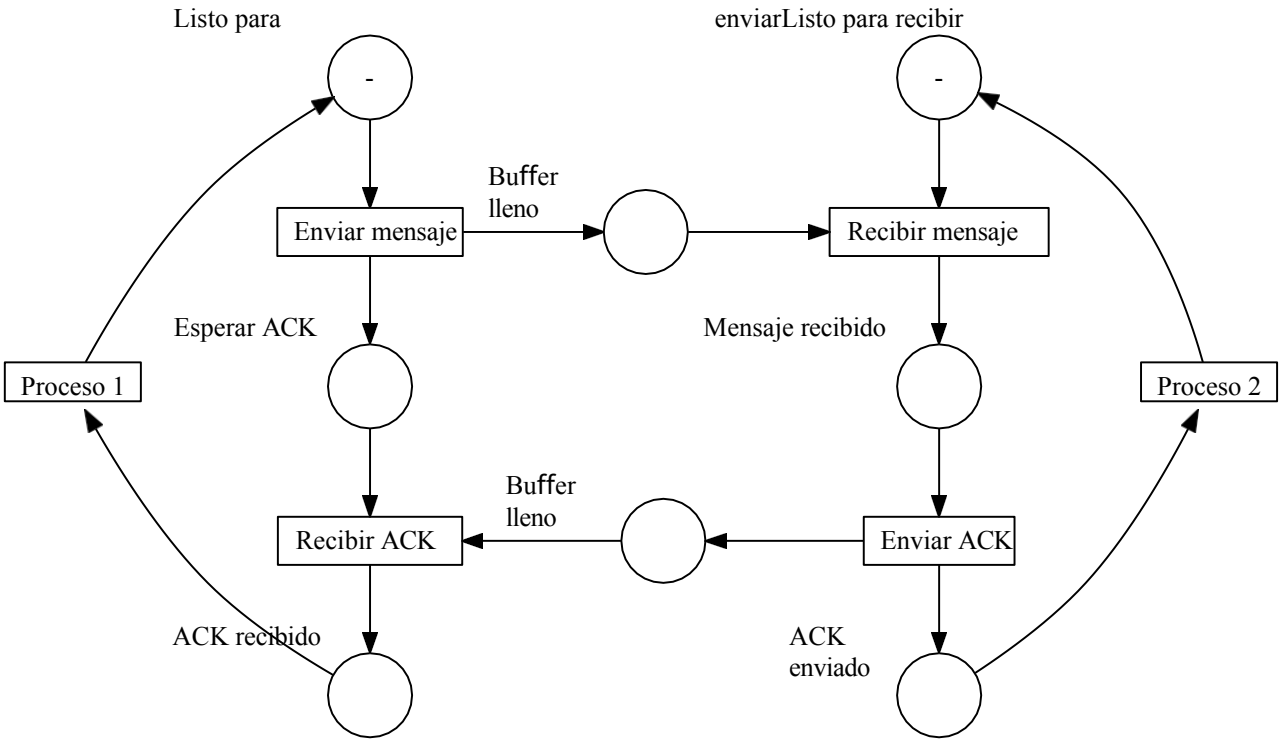


Figura 1.6: Modelo simplificado de red de Petri de un protocolo de comunicación.

**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 ficha 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 $k - 1$ fichas de P3 antes de realizar una escritura (transición T2), evitando así que los lectores entren en el bucle derecho de la red.

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, ya que no hay garantía de que una operación de escritura vaya a producirse finalmente. Por otro lado, el sistema está libre de bloqueos.

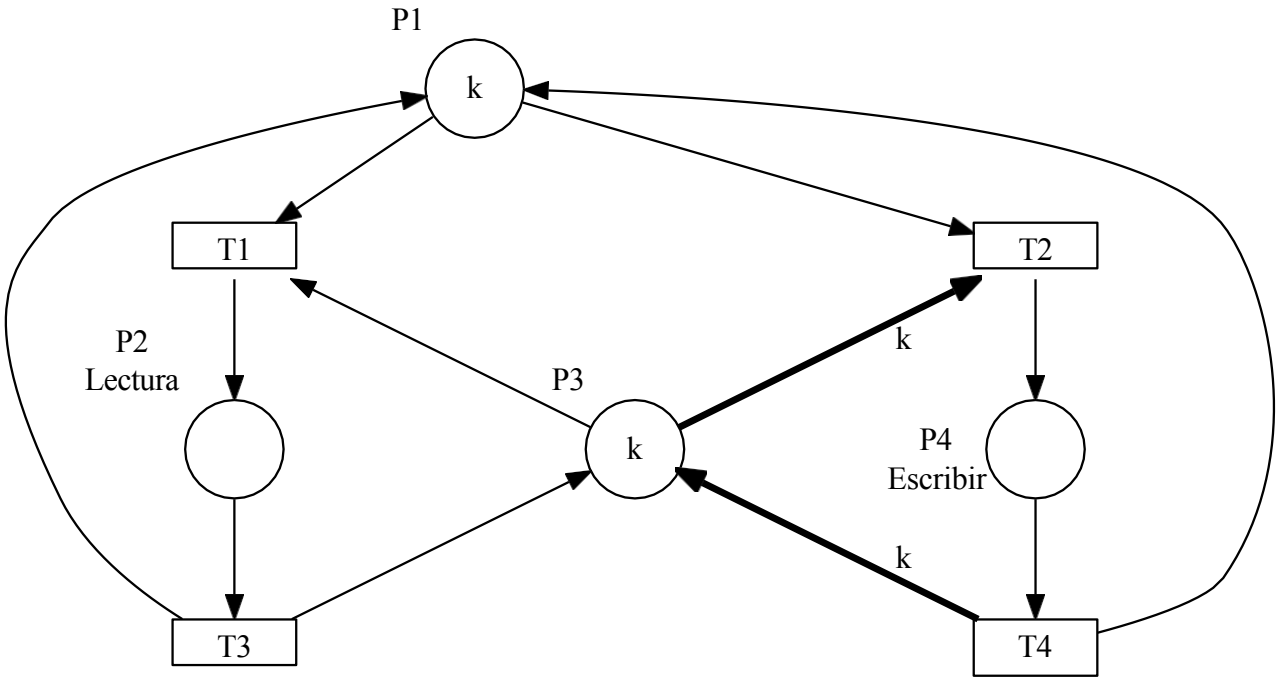


Figura 1.7: Un sistema de redes de Petri con k procesos que leen o escriben.

### 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.

#### Accesibilidad

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 fichas. En otras palabras, cambia el marcado $M$ . Una secuencia de disparos crea una secuencia de marcas en la que cada marca 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 $\sigma = M_0\ t_1\ M_1\ t_2\ M_2\ \text{- - -}\ t_l\ M_l$ o simplemente $\sigma = t_1\ t_2\ \text{- - -}\ t_l$ , ya que las marcas resultantes de cada disparo se derivan de la regla de disparo de transición descrita en la sección 1.1.3.

**Definición 11: Alcanzabilidad**

*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 significada es clara. 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 las marcas de un subconjunto de lugares y podemos ignorar las restantes. Esto da lugar a una variación del problema conocida como *problema de alcanzabilidad del submarcado*.

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-duro) [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 ELEMENTAL. Estos resultados ponen de relieve que el problema de la alcanzabilidad sigue siendo un área activa de investigación en informática teórica.

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

### Limitación y seguridad

Durante la ejecución de una red de Petri, los tokens pueden acumularse en algunos lugares. Las aplicaciones necesitan 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.

**Definición 12: Acotamiento**

*Un lugar de una red de Petri está limitado por k o es seguro por k 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. Se aplica cuando el lugar contiene 1 ó 0 fichas durante la ejecución.

**Definición 13: Seguridad**

*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 figuras 1.4, 1.5 y 1.6 son todas seguras.

La red de la Fig. 1.7 está acotada por k porque todos sus lugares están acotados por k.

### Liveness

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

**Definición 14: Liveness**

*Se dice que una red Patri ($N$, $M_0$ ) está viva (o equivalentemente se dice que $M_0$ es una marca viva 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 está viva.

Esto equivale a decir que la red de Petri está *libre de bloqueo*. Definamos ahora lo que constituye un punto muerto y mostremos ejemplos de ello.

**Definición 15: Bloqueo en redes de Petri**

*Un punto muerto 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 cocción adecuada para pasar del marcado actual a un marcado que permita la transición.

Las redes de las figuras 1.4, 1.5 y 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 sección 1.1.6, ahora podemos presentar una técnica de análisis importante para las redes de Petri: el *árbol de alcanzabilidad*.

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 una marca en otra.

Considere la red de Petri mostrada en la Fig. 1.8. La marca inicial es (1, 0, 0). En esta marcación 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.

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

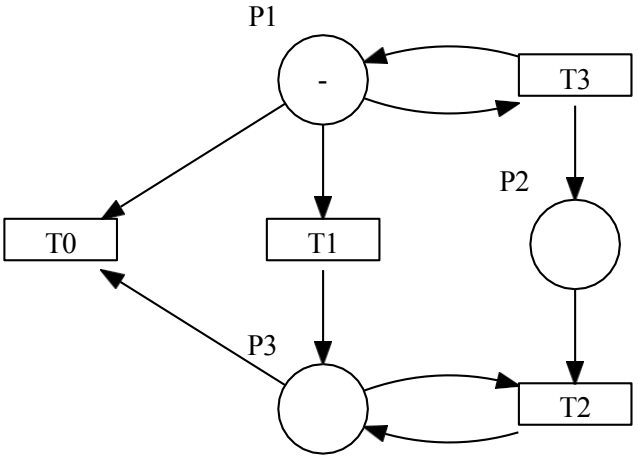


Figura 1.8: Una red de Petri marcada para ilustrar la construcción de un árbol de alcanzabilidad.

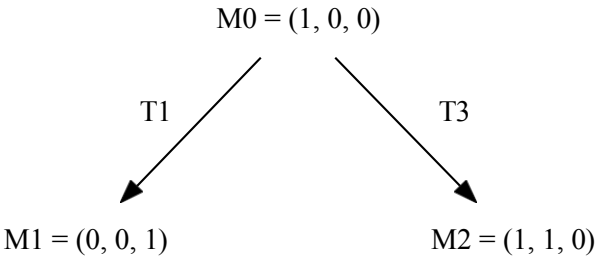


Figura 1.9: Primer paso para construir el árbol de alcanzabilidad de la red de Petri de la figura 1.8.

A partir del marcado (0, 0, 1) no podemos disparar ninguna transición. Esto se conoce como un marcado *muerto*. 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.

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 su lugar 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. Vea 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

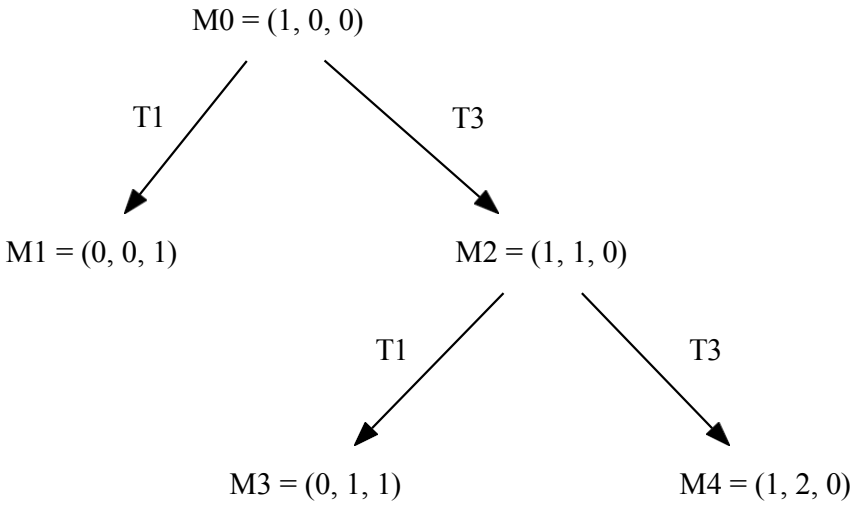


Figura 1.10: Segundo paso en la construcción del árbol de alcanzabilidad de la red de Petri de la figura 1.8.

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 un número infinito de marcas alcanzables en 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.

Observe, por otro lado, que algunas marcas son estrictamente diferentes de las marcas vistas anteriormente, pero permiten el mismo conjunto de transiciones. Decimos en este caso que la marca con fichas adicionales *cubre* 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. Por lo tanto, bastaría con marcar el lugar de acumulación con una etiqueta especial $\omega$, que significa infinito, ya que podríamos obtener tantas fichas 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 $\omega$,
2. una definición del algoritmo y los pasos precisos para construir el árbol de alcanzabilidad,
3. prueba matemática de que el árbol de alcanzabilidad generado por ella es finito,

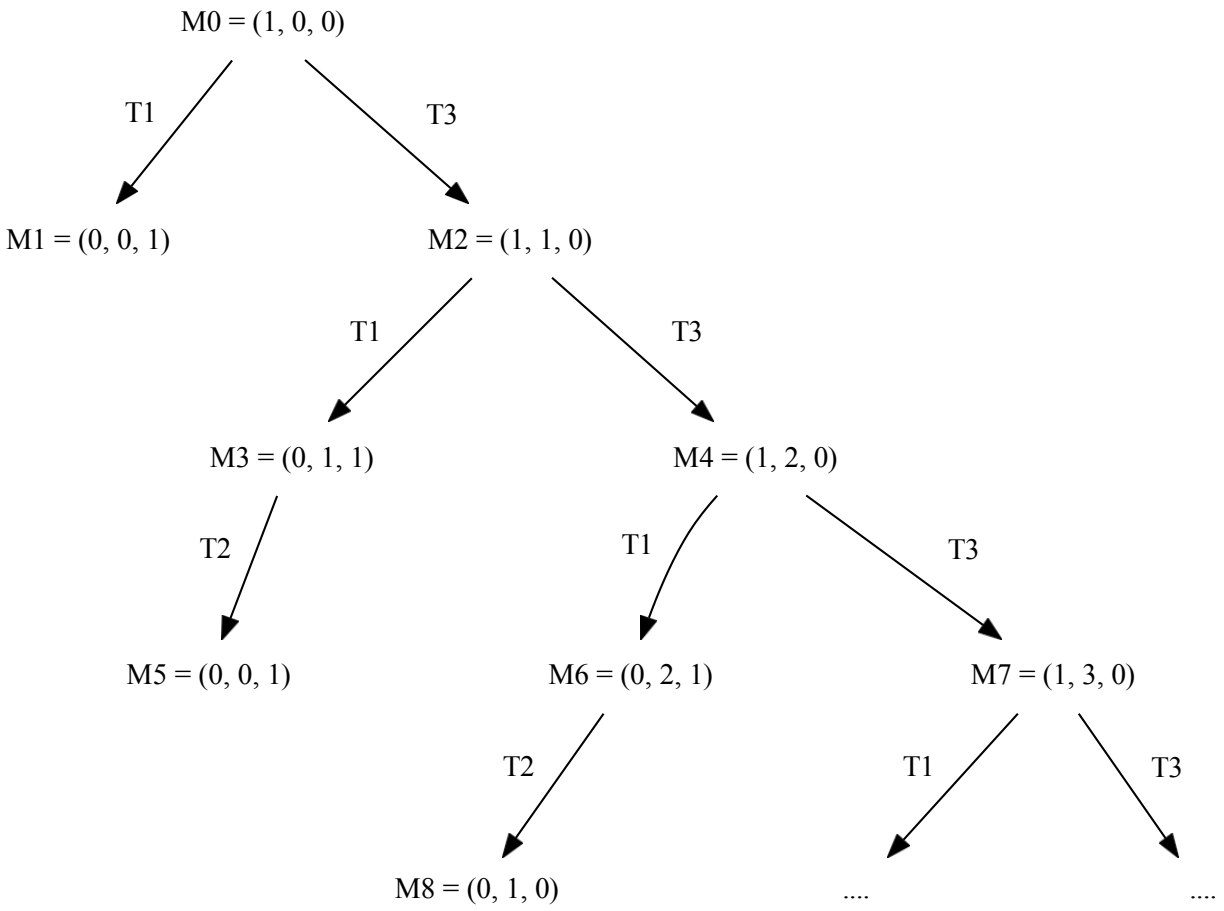


Figura 1.11: El árbol de alcanzabilidad infinita para la red de Petri de la figura 1.8.

4. y la distinción entre el árbol de alcanzabilidad y el *grafo de alcanzabilidad*

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].

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

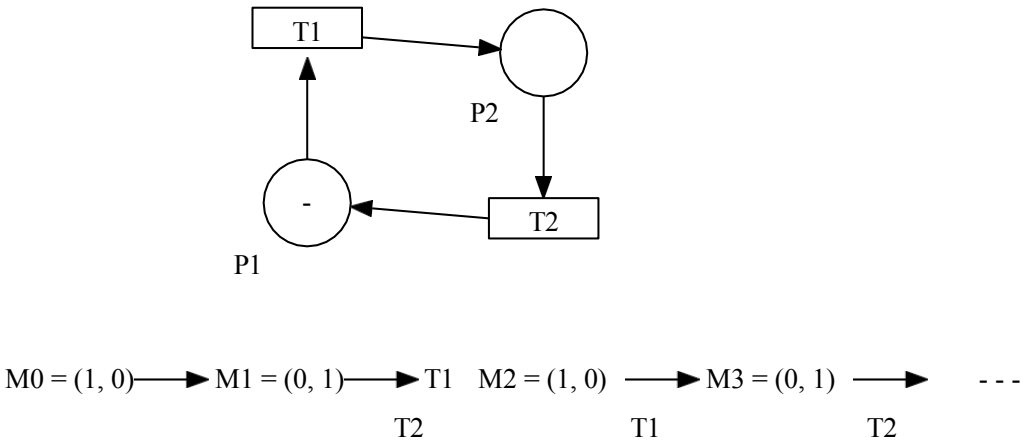


Figura 1.12: Una red de Petri simple con un árbol de alcanzabilidad infinito.

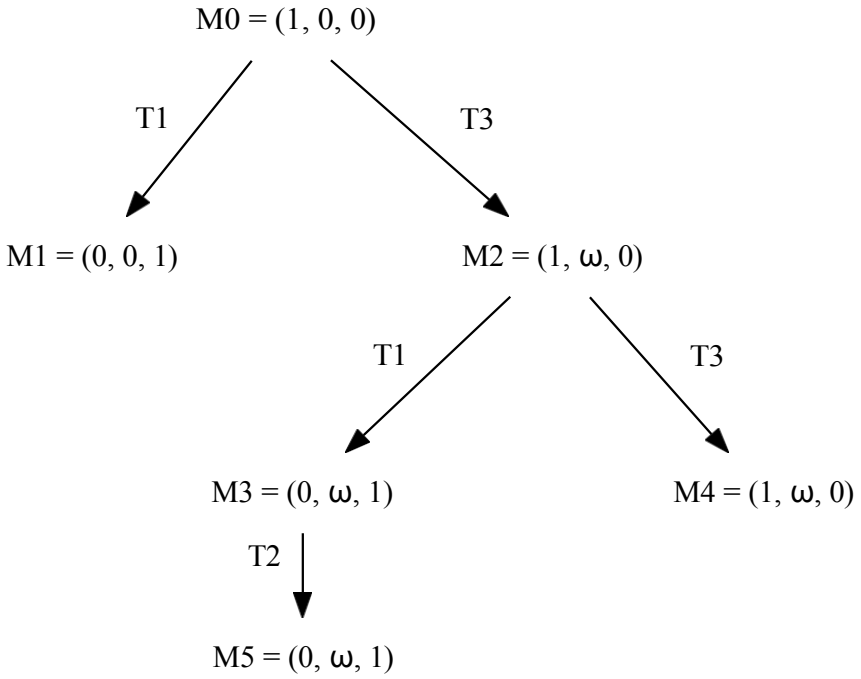


Figura 1.13: El árbol de alcanzabilidad finito para la red de Petri de la figura 1.8.

El modelo de memoria de Rust basado en el concepto de *propiedad* 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:

- Doble libre [Klabnik y Nichols, 2023, cap. 4.1]

- Uso después de libre [Klabnik y Nichols, 2023, Cap. 4.1]
- Punteros colgantes [Klabnik y Nichols, 2023, Cap. 4.2]
- Carreras de datos [Klabnik y Nichols, 2023, Cap. 4.2] (con algunas advertencias importantes expuestas en [Proyecto Rust, 2023c, Cap. 8.1])
- Pasar variables no seguras para hilos [Klabnik y Nichols, 2023, Cap. 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 estrictas reglas, el programa simplemente no compilará.

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 programación. Incluye características como la inferencia de tipos, los genéricos, los enums y la concordancia de patrones. Cada variable tiene un tipo, pero éste suele ser inferido por el compilador.
- Rendimiento: El rendimiento de Rust es comparable al de C y C++, y a menudo es más rápido que muchos otros lenguajes de programación populares como Java, Go, Python o Javascript. El rendimiento de Rust se consigue mediante una combinación de características como las ab- stracciones de coste cero, un tiempo de ejecución mínimo y una gestión eficiente de la memoria.
- Concurrencia: Rust tiene soporte incorporado para la concurrencia. Soporta varios paradigmas de concurrencia como el estado compartido, el paso de mensajes y la programación asíncrona. No obliga al desarrollador a implementar la concurrencia de una manera específica.
- Propiedad y préstamo: Rust utiliza un modelo de propiedad único para gestionar la memoria, lo que permite una asignación y desasignación de memoria eficientes sin riesgo de fugas de memoria o carreras de datos. Además, no depende de un recolector de basura, lo que ahorra recursos. El *verificador de préstamos* garantiza que sólo haya un propietario de un recurso en un momento dado.
- Impulsado por la comunidad: Rust cuenta con una vibrante y creciente comunidad de desarrolladores que con- tribuyen al desarrollo y al ecosistema del lenguaje. Cualquiera puede contribuir al lan-

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

guage y sugerir mejoras. La documentación también es de código abierto y las decisiones importantes se documentan en forma de Solicitudes de Comentarios (RFC)[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 y Nichols, 2023, Apéndice 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

El verificador de préstamos de Rust es un componente esencial de su modelo de propiedad, diseñado para garantizar la seguridad de la memoria y evitar las carreras de datos en el código concurrente. El verificador de préstamos 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.

La idea central detrás del verificador de préstamos es que cada trozo 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*, 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 del ámbito.

Para mayor claridad, presentaremos a continuación algunas de las reglas clave aplicadas por el verificador de préstamos:

- 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.

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

Puede costar cierto esfuerzo escribir código Rust que satisfaga estas reglas. El comprobador de préstamos suele señalarse como un aspecto del lenguaje que resulta confuso para los recién llegados. Sin embargo, esta disciplina compensa en términos de mayor seguridad de memoria y rendimiento. 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, carreras de datos y otros fallos, al tiempo que enseña buenas prácticas y patrones de codificación.

### Tratamiento 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 enteros, las funciones Rust que pueden encontrar errores devuelven un tipo std::result::Result[6] que puede contener el resultado del cálculo o un tipo de error cus- tom acompañado de una descripción del error. *rustc* requiere que el programador escriba código para ambos casos y el lenguaje proporciona mecanismos para facilitar la gestión de errores [Klabnik y Nichols, 2023, Cap. 9.2].

En Rust, la atención se centra 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, 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 [Klabnik y Nichols, 2023, Cap. 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 pánicos pueden ser capturados por hilos padre 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.

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

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

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 Ninguno. 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 pone de relieve 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.

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 fallos relacionados con la gestión de la memoria sin sacrificar el rendimiento. Entre ellos, podemos citar algunos ejemplos representativos:

- El proyecto de código abierto Android fomenta el uso de Rust para los componentes del SO por debajo del tiempo de ejecución de Android (ART) [Stoep y Hines, 2021].
- El núcleo Linux, que 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 alentado desde julio de 2022 [García, 2022].
- En Cloudflare, se construyó desde cero un nuevo proxy HTTP en Rust para superar las limitaciones archi- tectónicas de NGINX, reduciendo el uso de CPU en un 70% y el de memoria en un 67% [Wu y Hauck, 2022].
- En Discord, la reimplementación en Rust de un servicio crucial escrito en Go proporcionó grandes beneficios en el rendimiento y resolvió una penalización de rendimiento debida a la recogida de basura en Go [Howarth, 2020].
- En npm Inc, la empresa detrás del registro npm, Rust permitió escalar los servicios ligados a la CPU 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 y Couprie, 2017].

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

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" por los programadores en la encuesta Stack Overflow Developer Survey [Stack Overflow, 2022].

### 1.2.3 Importancia de la seguridad de la memoria

En esta subsección, se presentan pruebas convincentes que apoyan el uso de un lenguaje de programación seguro para 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 sean difíciles de corregir en los sistemas de 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 elevada cifra puede observarse en proyectos como:

- Proyecto Android de código abierto [Stepanov, 2020],
- los componentes Bluetooth y multimedia de Android [Stoep y Zhang, 2020],
- los Proyectos 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, Fernández, 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 racer 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 marco de instrumentación Valgrind[9].

En [Jaeger y 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 tipográfica 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 Comportamiento Indefinido (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

[9] https://valgrind.org/

la recogida de basura 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 pieza de software, no está exento de vulnerabilidades de seguridad. En el pasado se han descubierto graves fallos en la biblioteca estándar [Davidoff, 2018]. Además, la generación de código en Rust también incluye mitigaciones a exploits de diversa índole [Proyecto Rust, 2023b, Cap. 11]. Sin embargo, esto dista mucho de los conocidos problemas en C y C++.

## 1.3 Corrección de programas concurrentes

En el área de la computación concurrente, uno de los principales retos es demostrar la corrección 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 corrección 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 y Tanenbaum, 2017], se definen dos tipos de propiedades de corrección:

- **Propiedades de seguridad**: La propiedad debe ser *siempre* verdadera.
- **Propiedades de vida**: La propiedad debe convertirse finalmente en

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 punto muerto**: 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 semáforos (propuestos por [Dijkstra, 2002]), los monitores (propuestos por [Hansen, 1972, Hansen, 1973]) 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 significativa para la industria. El funcionamiento sin bloqueos es un requisito inevitable para muchos proyectos, como los sistemas operativos [Arpaci-Dusseau y Arpaci-Dusseau, 2018], los vehículos autónomos [Perronnet et al., 2019] y las aeronaves [Carreño y Muñoz, 2005, Monzón y Fernández-Sánchez, 2009].

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

## 1.4 Bloqueos

Los bloqueos son un problema común que surge en los sistemas concurrentes, que son sistemas en los que varios hilos o procesos se ejecutan simultáneamente y potencialmente comparten recursos. Se han estudiado al menos desde [Dijkstra, 1964], que acuñó el término "abrazo mortal" en holandés, que no se puso de moda.

Un bloqueo 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 estados del programa.

Los bloqueos muertos pueden ser un problema grave en los sistemas concurrentes, ya que pueden provocar que el sistema deje de responder o incluso que se bloquee. Por lo tanto, sería ventajoso poder detectar y prevenir los bloqueos muertos. 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 bloqueos son la memoria del sistema, los dispositivos de entrada/salida, los bloqueos y otros tipos de primitivas de sincronización.

Los bloqueos pueden ser difíciles de detectar y prevenir porque dependen de la sincronización precisa de los acontecimientos en el sistema. Incluso en los casos en los que pueden detectarse los bloqueos muertos, resolverlos puede ser difícil, ya que puede requerir liberar recursos que ya han sido adquiridos o hacer retroceder 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, considere 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 punto muerto, ya que ninguno de los dos procesos puede progresar. La Fig. 1.14 ilustra esta situación. El ciclo que aparece en ella indica un punto muerto, como se explicará en la siguiente sección.

### 1.4.1 Condiciones necesarias

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

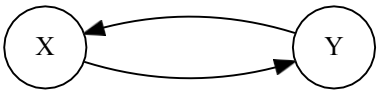


Figura 1.14: Ejemplo de un gráfico de estados con un ciclo que indica un punto muerto.

1. **Exclusión mutua**: Al menos un recurso del sistema debe mantenerse en 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**: 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 tanteo**: Los recursos no pueden adelantarse, 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**: 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.

Normalmente, las tres primeras condiciones son características del sistema estudiado, es decir, los pro- tocollos 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. Pero sólo indican la posibilidad de un punto muerto 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 tanteo, 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 (SO) moderno. El programador 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 y Arpaci-Dusseau, 2018, Cap. 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

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 sus bloqueos 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 (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 GAR, 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 (SGBD) 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 (TWF), y examinándolo en busca de cualquier ciclo. Si se identifica un ciclo (bloqueo), 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 SGBD se trata ampliamente en [Bernstein et al., 1987].

**Aceptar o ignorar por completo los puntos muertos**

En algunos casos, puede ser admisible aceptar simplemente el riesgo de que se produzcan bloqueos y `gestionarlos` a medida que vayan apareciendo. Este enfoque puede ser adecuado en sistemas en los que el coste de prevenir o detectar los bloqueos muertos sea demasiado elevado, o en los que la frecuencia de los bloqueos muertos 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 importantes también muestran este comportamiento. Por otro lado, un sistema vital no puede permitirse fingir que su funcionamiento estará libre de bloqueos por ningún motivo.

## 1.5 Variables de condición

Las variables de condición 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:

- wait: Bloquea el hilo o proceso actual. En algunas implementaciones, el mutex asociado se libera como parte de la operación.
- señal: Despierta un hilo o proceso que espera en la variable de condición. En algunas imple- mentaciones, el bloqueo mutex asociado es adquirido inmediatamente por el hilo o proceso señalizado.

Las variables de condición 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 debe continuar ejecutándose. El mutex garantiza que sólo un hilo o proceso pueda acceder a la variable de condición 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 programación específica puede variar en función de la implementación.

A lo largo de los años, se han desarrollado diversas implementaciones y optimizaciones para las variables de condición 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 variables de condición forman parte de la biblioteca estándar POSIX para hilos [Nichols et al., 1996].

y en la actualidad se utilizan ampliamente en lenguajes y sistemas de programación concurrentes. Se encuentran entre otros en:

- UNIX10,
- Óxido11
- Pitón12
- Go13
- Java14

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 omitidas o activaciones espurias. 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 variable de condición 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.

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 establecido flag. T2 espera en cv hasta que recibe una señal de T1. El listado 1.1 muestra el pseudocódigo correspondiente.

Supongamos ahora que T1 activa la bandera y emite una señal a cv, pero T2 aún no ha entrado en estado de espera en cv debido a algún retraso en la programación. 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 establece la bandera en true.
2. T1 indica a cv que despierte a T2.
3. T1 libera la cerradura.
4. T2 adquiere el bloqueo y comprueba si flag ha cambiado. Como flag sigue siendo falso, T2 entra en estado de espera en cv.

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

```
// T1
bloqueo.adquirir()
bandera = verdadero
cv.signal()  // Señal a T2 para que se despierte
lock.release()

// T2
bloqueo.adquirir()
while (flag == false)   // Espere hasta que flag haya cambiado
    cv.wait(lock)
lock.release()
```

Listado 1.1: Pseudocódigo para un ejemplo de señal perdida.

5. Debido a retrasos en la programación 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

Un despertar espurio 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 variable de condición 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 la bandera a verdadero 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 establece la bandera en true.
2. T1 indica a cv que despierte a T2.
3. T1 libera la cerradura.
4. T2 adquiere el bloqueo y comprueba si flag es verdadero. Como flag sigue siendo falso, T2 entra en estado de espera en cv.

5. Debido a algún detalle de implementación interna de la variable de condición 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 variable de condición 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 esperar, 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. En cada pasada, el código se acerca cada vez más a la representación final. Estas fases están hoy en día bien definidas y diferentes compiladores implementan alguna forma de ellas [Aho et al., 2014, Cap. 1.2].

La primera pasada de un compilador típico es la fase de **análisis léxico**. 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* 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**, también conocida como fase del analizador sintáctico. En esta fase, los tokens producidos por el léxico se analizan según las reglas de la gramática del lenguaje de programación. El analizador *sintáctico* construye un árbol de análisis sintáctico o un árbol sintáctico abstracto (AST) que representa la estructura del código.

La tercera pasada es la fase de **análisis semántico**, en la que el compilador comprueba la corrección semántica del código, como la comprobación de errores de tipo, variables no definidas y op- eraciones no válidas. El *analizador semántico* 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**. 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.

Por último, puede haber cero o más fases de **optimización del código**. Éstas 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 constante [Aho et al., 2014, cap. 8.5.4],
- Desenrollado de bucles [Aho et al., 2014, cap. 10.5],
- asignación de registros [Aho et al., 2014, cap. 8.1.4],
- propagación constante [Aho et al., 2014, Cap. 9],
- Análisis de vida útil [Aho et al., 2014, Cap. 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 a través de los bloques básicos. En Rust, un ejemplo de optimización global es la optimización del tiempo de enlace (LTO) [Huss, 2020].

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

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 general.

## 1.7 Comprobación de modelos

La comprobación de modelos es una técnica utilizada en el desarrollo de software para verificar formalmente la corrección 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 y la ausencia de puntos muertos.

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, que son secuencias de eventos que violan 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.

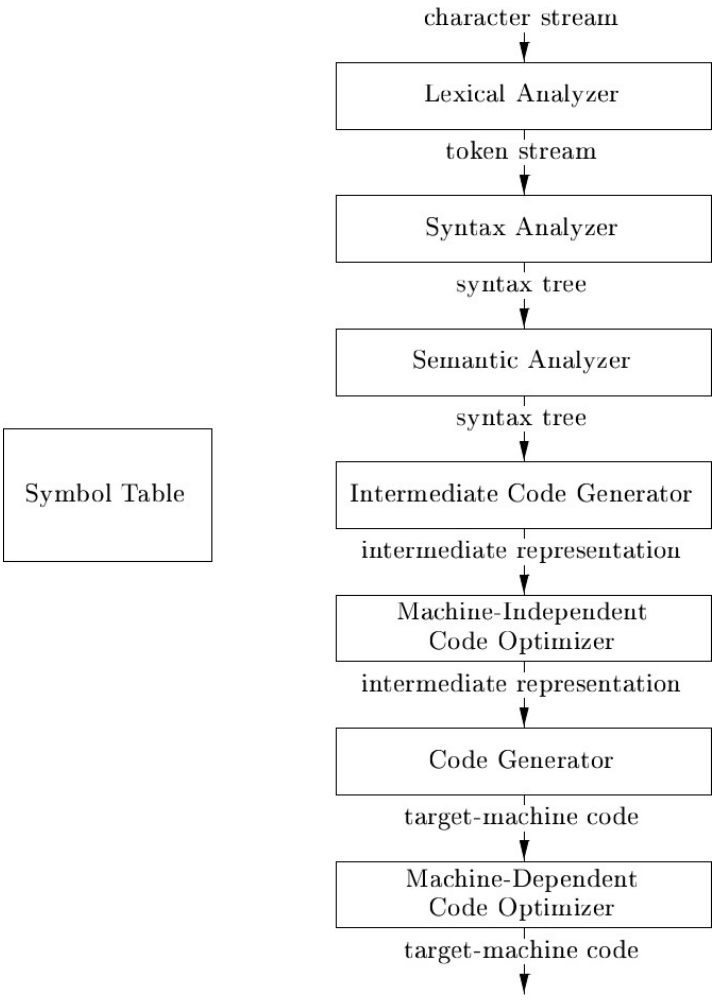


Figura 1.15: Fases de un compilador.

La comprobación de modelos se ha convertido en una técnica ampliamente aplicada en el desarrollo de sistemas de software críticos, como los sistemas de control aeroespaciales [Carreño y Muñoz, 2005, Monzón y Fernández-Sánchez, 2009] y de automoción [Perronnet et al., 2019], los dispositivos médicos y los sistemas financieros. Al verificar la corrección 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 corrección del software. A diferencia de los métodos de prueba 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 corrección de sistemas de software complejos. Esto reduce el tiempo y el coste del desarrollo de software y aumenta la confianza en

la corrección del 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 prueba formal de la corrección 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 documento 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 al desarrollador con prontitud 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 información rápida sobre el uso correcto de las primitivas de sincronización, ahorrando tiempo y, por tanto, 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 más amplia de desarrolladores de Rust.

# Capítulo 2

# Estado de la técnica

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 bloqueos. Algunas de estas publicaciones anteriores contienen enfoques que han guiado este trabajo.

En las dos secciones siguientes, 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.

Como siguiente paso, exploramos la comunidad de investigación que hay detrás del Concurso de Comprobación de Modelos (MCC) y los comprobadores 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 del 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 él se enumera explícitamente que la mayoría de las herramientas de verificación formal no admiten la concurrencia [Reid, 2021].

El intérprete *Miri* [1] desarrollado por el proyecto Rust en GitHub es un intérprete experimental para la representación intermedia del lenguaje Rust (Mid-level Intermediate Represen- tation, comúnmente conocida como "MIR") que permite ejecutar binarios estándar de proyectos de carga en

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

una forma granularizada, instrucción por instrucción, para comprobar la ausencia de Comportamientos Indefinidos (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.

[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 inseguro que, en realidad, 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.

[Proyecto Kani, 2023] es otra conocida herramienta para la verificación formal de código Rust destinada a comprobar los bloques inseguros a nivel de bits. Ofrece un arnés de pruebas análogo al arnés 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):

- Seguridad de la memoria, por ejemplo, desreferencias de punteros nulos
- Aserciones especificadas por el usuario, es decir, ¡assert!(...)
- La ausencia de pánicos, por ejemplo, unwrap() en valores None
- La ausencia de algunos tipos de comportamiento inesperado, por ejemplo, desbordamientos aritméticos.

Sin embargo, los programas concurrentes están actualmente fuera de alcance[3]. La conclusión es que Kani ofrece una CLI fácil de usar y un arnés 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 puntos muertos 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 sección 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. 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 y 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 en el algoritmo ingenuo de detección de puntos muertos. A través de

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

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 bloqueos 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 literatura.

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 vari- able 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 variables de condición.

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 bloqueo dos veces sobre el mismo mutex en el hilo principal. Lamentablemente, el código disponible en GitHub[4] como parte

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

de la tesis ya no es válida para la nueva versión de *rustc*, puesto que las partes internas del compilador han cambiado significativamente en los últimos tres años.

En un preprint de finales de 2022, [Zhang y Liua, 2022] implementan una traducción del código fuente de Rust a redes de Petri para comprobar los bloqueos muertos. 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 (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 comprobación de código Rust, que podría resultar atractivo para el lector interesado. 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 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.

Lamentablemente, ninguno de ellos cumple los requisitos de la tarea.

Dado que una red de Petri es un grafo, se consideró 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.

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

- 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 una *envoltura* 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, ya que es el responsable de verificar la ausencia de bloqueos. Afortunadamente, se han desarrollado varios comprobadores de modelos para analizar redes de Petri.

El Concurso de Comprobación de Modelos (MCC) [Kordon et al., 2021] organizado en la Universidad de la Sorbona de París es una gran fuente de comprobadores de modelos de última generación. Se trata de un concurso anual en el que los comprobadores de modelos presentados se ejecutan en 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 y el número total de puntos de referencia ha crecido constantemente a medida que se han ido añadiendo nuevos modelos.

Cada año, los puntos de referencia incluyen redes de lugares/transiciones (redes P/T), es decir, redes de Petri, y redes de Petri coloreadas (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 amplia aplicabilidad 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 202213 reproducidas en la Fig. 2.1 ilustra que varios

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

Los verificadores de modelos han demostrado una participación ininterrumpida, con ejemplos notables que incluyen:

- Herramienta para la Verificación de Redes de Petri Temporizadas (TAPAAL) mantenida por la Universidad de Aalborg en [Dinamarca14], ganadora de una medalla de oro en la edición de 2023.
- Analizador de redes de Petri de bajo nivel (LoLA) mantenido por la Universidad de Rostock en [Alemania15], ganador en ediciones anteriores y base para otros modelos.
- ITS-tools [Thierry Mieg, 2015], que también se combinó con LoLA y obtuvo medallas en 202016.

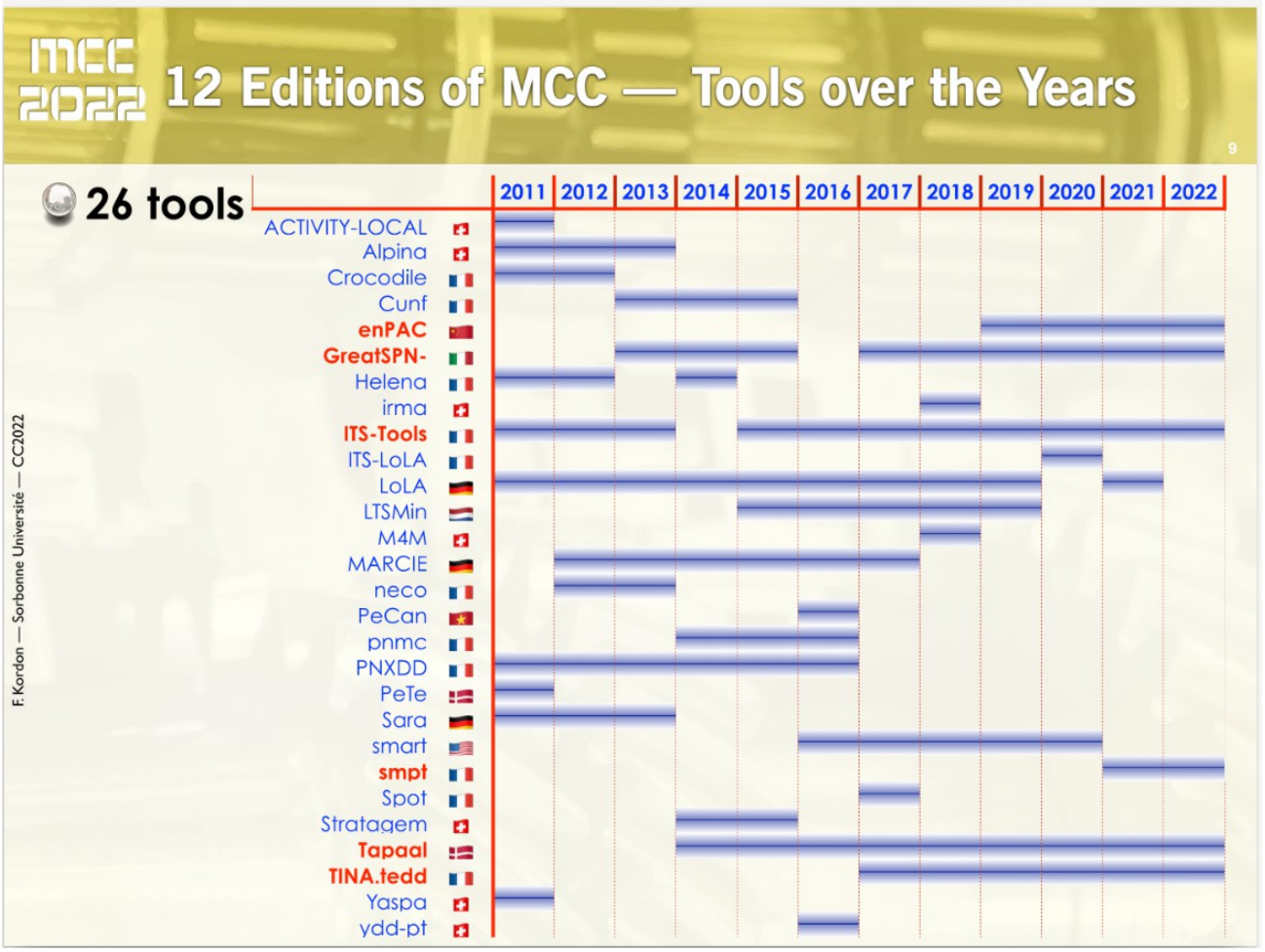


Figura 2.1: Participación de los verificadores de modelos en el MCC a lo largo de los años.

Estas observaciones indican colectivamente la madurez y vitalidad de la comunidad de comprobadores de modelos. El establecimiento de un panorama de herramientas bien desarrollado, fomentado por el col-

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

laboración y la difusión de código abierto de resultados, puntos de referencia y técnicas, presenta una valiosa oportunidad para aprovechar estas herramientas en el ámbito del desarrollo de software. Concretamente, en el contexto de integrarlas como backends para un traductor específico de lenguaje que se encargue de automatizar el proceso de creación de modelos de redes de Petri. Al capitalizar los esfuerzos aca- démicos invertidos en los comprobadores de modelos, se puede lograr una mayor seguridad y fiabilidad en los proyectos de software.

## 2.5 Intercambio de formatos de archivo para 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 temporizadas, 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, dado 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 Lenguaje de marcado de redes Petri

El lenguaje de marcado de redes de Petri (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 la "Reunión sobre formatos de intercambio basados en XML/SGML para redes de Petri" celebrada en Aarhus en junio de 2000 [Jüngel et al., 2000, Weber y Kindler, 2003], con el objetivo de proporcionar un formato estandarizado y ampliamente aceptado para los modelos de redes de Petri. PNML es una norma ISO que consta, a partir de 2023, de tres partes:

- ISO/IEC 15909-1:[200418] (y su última revisión ISO/IEC 15909-1:[201919]) para conceptos, definiciones y notación gráfica.
- ISO/IEC 15909-2:[201120] para la definición de un formato de transferencia basado en XML.
- ISO/IEC 15909-3:[202121] para las extensiones y los mecanismos de estructuración.

[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 21https://www.iso.org/standard/81504.html

Se ha convertido en un estándar de facto para intercambiar modelos de 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 y 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 el lenguaje de marcado extensible (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.

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 del PNML, el intercambio de modelos de redes de Petri era una tarea ardua, ya 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, ya que proporciona un formato estándar que puede ser analizado y procesado fácilmente por distintos sistemas. Por ejemplo, es el formato utilizado en [Zhang y 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.

El 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 fichas por la red.

El formato DOT está basado en texto y es fácil de usar, lo que lo convierte en una opción popular para generar representaciones visuales de gráficos. Esta sencillez también significa que los archivos DOT pueden ser generados fácilmente por programas y pueden ser leídos por una amplia gama de herramientas de software, lo que resulta esencial para la interoperabilidad. Además, el 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 - Analizador de redes de Petri de bajo nivel

Low-Level Petri Net Analyzer (LoLA) [Schmidt, 2000] es un comprobador 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 Lógica Computacional de Árbol* (CTL*). Su punto fuerte es la evaluación de propiedades sencillas como la libertad de bloqueo 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. Por lo tanto, es necesario implementar el formato de archivo requerido por la herramienta. En la sección 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 arquitectónicas 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 la Representación Intermedia de Nivel Medio (MIR).
3. 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.

## 3.1 En busca de un backend

En pocas palabras, 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 genéricos o el propio sistema de tipos enriquecido, 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

todo el equipo para mantener y estar 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 pone a ello. Por 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.

Apoyar 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 las partes relevantes del compilador de Rust.

## 3.2 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.

Otra elección de diseño importante en *rustc* es el interning. Interning es una técnica para almacenar cadenas 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 asignador 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 económica comparando punteros. Esto puede

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

reducir el uso de memoria y acelerar las operaciones que comparan o manipulan cadenas y estructuras de datos.

*rustc* utiliza la infraestructura del compilador LLVM[3] para realizar la generación de código de bajo nivel y la op- timización. LLVM proporciona un marco flexible para compilar código a una variedad de objetivos, 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 representación interme- diante (IR) LLVM del código fuente y luego ordenar a LLVM que lo transforme al objetivo de compilación, 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] de miembros del equipo de Rust.

### 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 un ordenador. 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 de unidades atómicas de código fuente conocidas como tokens.

A continuación se realiza el análisis sintáctico. 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 etapa. La representación intermedia resultante de esta etapa es, por tanto, el AST.

#### Bajada HIR

A continuación, el AST se convierte en Representación Intermedia de Alto Nivel (HIR). Este proceso se conoce como "rebajar". Esta representación se parece al código Rust pero con construcciones complejas

[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

desugarizados a versiones más simples. Por ejemplo, todos los bucles while y for se convierten en versiones más simples
bucles de bucle.

El HIR se utiliza para realizar algunos pasos importantes:

1. *Inferencia de tipo*: La detección automática del tipo de una expresión, por ejemplo, al declarar variables con let.
2. *resolución de traits* : Garantizar que cada bloque de implementación (impl) hace referencia a un trait válido y existente.
3. *Comprobación de tipos* : 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 corrección y la coherencia.

**Bajada del MIR**

En esta etapa, la HIR se rebaja a Representación Intermedia de Nivel Medio (MIR), que se utiliza para la *comprobación de préstamos*. Como parte del proceso, se construye la Representación Intermedia de Alto Nivel Tipificada (THIR), que es una representación más fácil de convertir a MIR que la HIR.

El THIR es una versión aún más desugarizada del HIR. Se utiliza para la concordancia de patrones y la concordancia exhaustiva. 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, ya 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 entre sí para formar el binario.

### 3.2.2 Óxido nocturno

Comprender el modelo de liberación 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", 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, probadas y consideradas lo suficientemente estables para su uso en producción.

Por otro lado, el compilador nightly de Rust es la versión más puntera, 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 denominada *bandera de característica*. Sólo pueden utilizarse al compilar con la cadena de herramientas nightly. Las banderas de características pueden habilitar

- 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,
- banderas adicionales del compilador.

La lista completa de banderas de características se encuentra en [Proyecto Rust, 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 Estrategia de interceptación: Selección de un punto de partida adecuado para la traducción

En esta sección, se elucidan los motivos para seleccionar la Representación Intermedia de Nivel Medio (MIR) como punto de partida para la traducción a una red de Petri. Esta elección de diseño arquitectónico se justifica por varias razones.

### 3.3.1 Beneficios

En primer lugar, el MIR es el IR independiente de la máquina más bajo utilizado en *rustc*. Captura la semántica del código Rust después de que haya sido sometido a una serie de pases de optimización sin depender de los detalles

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

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, 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 la comprobación de préstamos, 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 corrección 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 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 corrección 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 optimizaciones de rendimiento. 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.

Además, como se explicará en la siguiente sección, la MIR se basa en el concepto de grafo de flujo de control (CFG), es decir, 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 susceptible de traducción. Tanto la MIR como las redes de Petri pueden considerarse modelos gráficos que capturan las relaciones e interacciones entre diferentes entidades. El gráfico 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. Por lo tanto, resulta más fácil convertir la MIR en una red de Petri, ya 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 bloqueos.

Por último, 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 tasa/por módulo, lo que permitiría un análisis más rápido y eficiente de grandes bases de código de Rust.

### 3.3.2 Limitaciones

Sin embargo, el enfoque de basar la traducción en la Representación Intermedia de Nivel Medio (MIR) tiene algunas limitaciones.

El más importante es que el MIR está sujeto 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 MIR. 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 numerosas 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 hacer frente a estos cambios en la sección 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[7] exacta 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 del MIR pueden ser superfluas para el análisis de detección de bloqueo y su traducción puede ampliar 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 bloqueos y señales perdidas en el código Rust.

Tanto [Meyer, 2020] como [Zhang y 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 RI de nivel superior.

## 3.4 Representación intermedia de nivel medio (MIR)

En esta sección se ofrece una visión general de la Representación Intermedia de Nivel Medio (MIR). La MIR se introdujo en la RFC 12118 en agosto de 2015. Exploraremos sus diferentes partes, cómo se mapean en ellas diferentes fragmentos de código y la estructura de grafos subyacente.

[7] https://github.com/Skasselbard/Granite/blob/master/rust-toolchain
8https://rust-lang.github.io/rfcs/1211-mir.html

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

Listado 3.1: Programa Rust sencillo para explicar los componentes de la MIR.

```
// ADVERTENCIA: Este formato de salida está destinado únicamente a consumidores humanos
// y está sujeta a cambios sin previo aviso. Dése el gusto
fn main() -> () {
    let mut _0: ();                      // lugar de retorno en el ámbito 0 en 1:11 src/main.rs:1:11:
    let mut _1: usize;                   // en el ámbito 0 en src/main.rs:2:11: 2:33
    let mut _2: &std::env::Args;         // en el ámbito 0 en src/main.rs:2:11: 2:33
    let _3: std::env::Args;              // en el ámbito 0 en src/main.rs:2:11: 2:27

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

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

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

```
31
32      bb3: {
33          volver;                     // ámbito 0 en src/main.rs:7:2: 7:2
34      }
35
36      bb4 (limpieza): {
37          drop(_3) -> [return: bb5, unwind terminate]; // ámbito 0 en src/main.rs:6:6: 6:7
38      }
39
40      bb5 (limpieza): {
41          reanudar;                   // ámbito 0 en src/main.rs:1:1: 7:2
42      }
43  }
```

Listado 3.2: MIR del Listado 3.1 compilado utilizando rustc 1.71.0-nightly en modo depuración.

Considere el código de ejemplo que aparece en el Listado 3.1, el [MIR9] 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 una *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* 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   // ADVERTENCIA: Este formato de salida está destinado únicamente a consumidores humanos
2   // y está sujeta a cambios sin previo aviso. Dése el gusto
3   fn main() -> () {
4       let mut _0: ();                     // lugar de retorno en el ámbito 0 en src/main.rs:1:11: 1:11
5       let mut _1: usize;                  // en el ámbito 0 en src/main.rs:2:11: 2:33
6       let mut _2: &std::env::Args;        // en el ámbito 0 en src/main.rs:2:11: 2:33
7       let _3: std::env::Args;             // en el ámbito 0 en src/main.rs:2:11: 2:27
8
9       bb0: {
10          StorageLive(_1);                // alcan 0 en src/main.rs:2:11: 2:33
                                                ce
11          StorageLive(_2);                // alcan 0 en src/main.rs:2:11: 2:33
                                                ce
12          StorageLive(_3);                // alcan 0 en src/main.rs:2:11: 2:27
                                                ce
```

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

```
13          _3 = args() -> bb1;          // ámbito 0 en src/main.rs:2:11: 2:27
14                                       // mir::Constante
15                                       // + span: src/main.rs:2:11: 2:25
16                                       // + literal: Const { ty: fn() ->
17                                       //     Args {args},
18                                       //     val: Valor(<ZST>) }
19      }
20
21      bb1: {
22          _2 = &_3;                    // scope 0 en src/main.rs:2:11: 2:33
23          _1 = <Args as ExactSizeIterator>::len(move _2) -> [return: bb2, unwind:
                bb4];
24                                       // ámbito 0 en src/main.rs:2:11: 2:33
25                                       // mir::Constante
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: Valor(<ZST>) }
30      }
31
32      bb2: {
33          AlmacenamientoMuerto(_       // alcanc 0 en src/main.rs:2:32: 2:33
            2);                                  e
34          drop(_3) -> bb3;             // alcanc 0 en src/main.rs:6:6: 6:7
                                                 e
35      }
36
37      bb3: {
38          AlmacenamientoMuerto(_       // alcanc 0 en src/main.rs:6:6: 6:7
            3);                                  e
39          AlmacenamientoMuerto(_       // alcanc 0 en src/main.rs:6:6: 6:7
            1);                                  e
40          volver;                      // alcanc 0 en src/main.rs:7:2: 7:2
                                                 e
41      }
42
43      bb4 (limpieza): {
44          drop(_3) -> [return: bb5, unwind terminate]; // ámbito 0 en src/main.rs:6:6: 6:7
45      }
46
47      bb5 (limpieza): {
48          reanudar;                    // ámbito 0 en src/main.rs:1:1: 7:2
49      }
50  }
```

Listado 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 sólo ha cambiado ligeramente con el tiempo. Consulte [Meyer, 2020, Sección 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 (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 MIR

El MIR está formado por funciones. Cada función se representa como una serie de bloques básicos (BB) conectados por aristas dirigidas. Cada BB contiene cero o más *sentencias* (normalmente abreviadas como "STMT") y por último una sentencia *terminadora*, para abreviar *terminator*. El terminador 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 `terminadores`. Los terminadores 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 de bifurcación condicional o incondicional.

En la Fig. 3.1 se presenta como ejemplo la representación gráfica de la MIR que aparece en el Listado 3.2. Las sentencias están coloreadas en azul claro y los terminadores en rojo claro. Para que el tipo de declaración terminadora sea más claro, se han añadido anotaciones adicionales como en CALL: o DROP:.

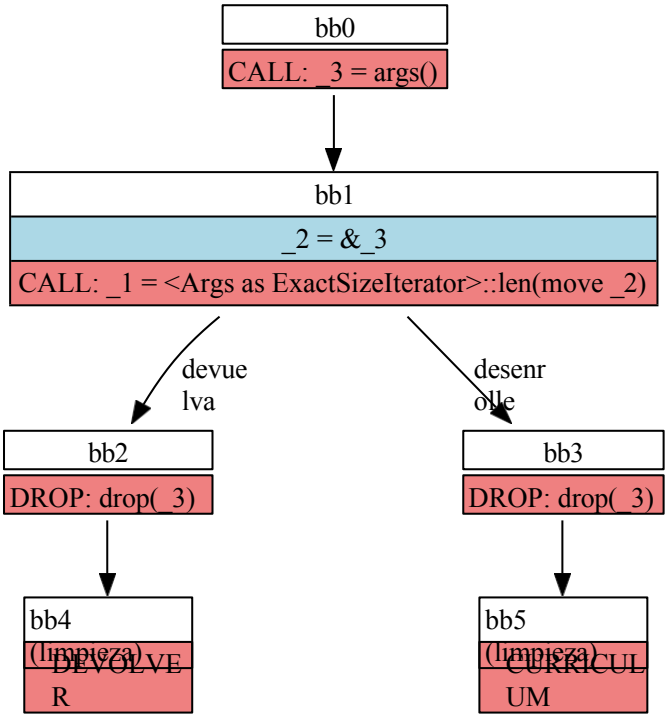


Figura 3.1: Representación gráfica del flujo de control de la MIR mostrada en el listado 3.2.

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, 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.
- RETORNO: retorna de la función. El valor de retorno se almacena siempre en la variable local _0, como veremos en breve.
- RESUME: indica que el proceso debe continuar desenrollándose. 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 nocturna[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: *locales* y *lugares*. Es fundamental observar que estos "lugares" *no* están relacionados con los lugares de las redes de Petri. Los lugares se utilizan para representar todo tipo de ubicaciones de memoria (incluidos los alias), mientras que los locales se limitan a las ubicaciones de memoria basadas en la pila, es decir, las variables locales de una función. En otras palabras, los lugares son más generales y los locales son un caso especial de un lugar, por lo que los lugares no siempre son equivalentes a los locales. 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 "_<índice>". En concreto, 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.

### 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. Además, esto ilustra cómo la salida de la MIR representa construcciones de nivel superior que se encuentran a menudo al programar en Rust.

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

**BB0**

- La función main() comienza en BB0.
- Se llama a una función (std::env::args()) para obtener un iterador sobre los argumentos proporcionados al programa.
- El valor de retorno de la función, el iterador, se asigna al 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 al método std::env::Args::len() por desplazamiento 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 pánico.

**BB2**

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

**BB3**

- La función retorna. El valor de retorno (local _0) es de tipo "unidad "[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* puesto que ya no es necesaria.
- Si la caída tiene éxito, la ejecución continúa en BB5, de lo contrario, finaliza el programa inmediatamente.

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

**BB5**

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

## 3.5 Inlineado 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 inlineado 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 puntos muertos 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 bloqueo 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 "inline" 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 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 el 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 el 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, ya que estos estados extraños pueden violar las garantías de seguridad ofrecidas por el compilador.

Por otro lado, alinear el 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. Por lo tanto, 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.

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

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.

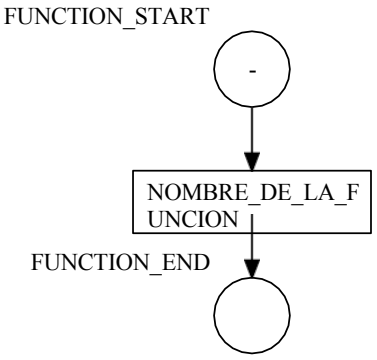


Figura 3.2: El modelo de red de Petri más simple para una llamada de función.

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 mostrar las dificultades de tratar con funciones llamadas en varios lugares en el código.

```
1 fn función_sencilla() {}
2
3 pub fn main() {
4     para n en 0..5 {
5         función_sencilla();
6     }
7 }
```

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

### 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 función_simple() aparece dos veces o más. El listado 3.5 satisface esta condición y está diseñado para mostrar el comportamiento extra- neo descrito al principio de la sección.

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 ¡pánico! y siempre termina con éxito, dado que la variable segunda_llamada nunca es verdadera antes de la línea 9.

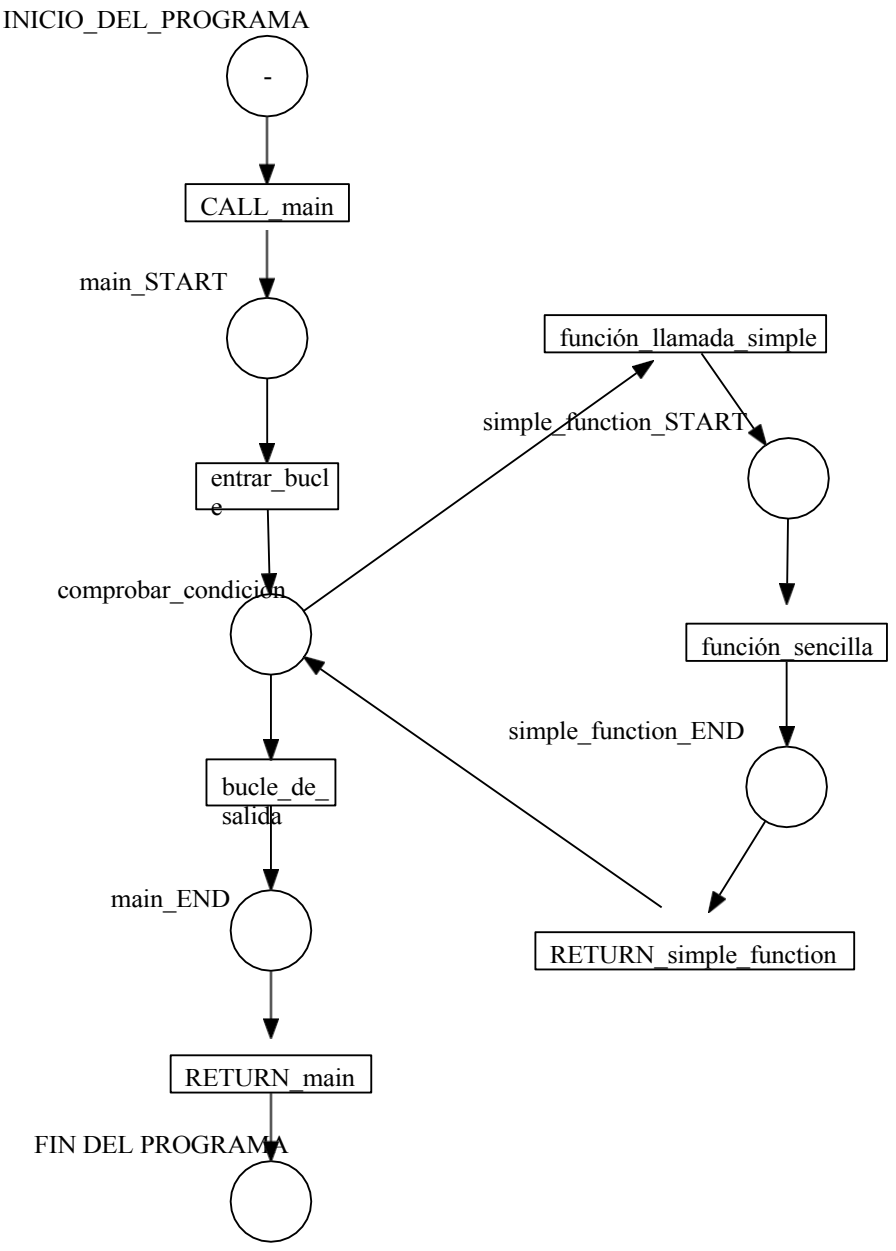


Figura 3.3: Una posible red de Petri para el código del listado 3.4 aplicando el modelo de la figura 3.2.

Sin embargo, el PN representado en la Fig. 3.4. es llamativamente defectuoso, lo que lo hace inadecuado 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 m a i n _ e n d _ p l a c e . 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 relativos al segundo token. El token en check_flag

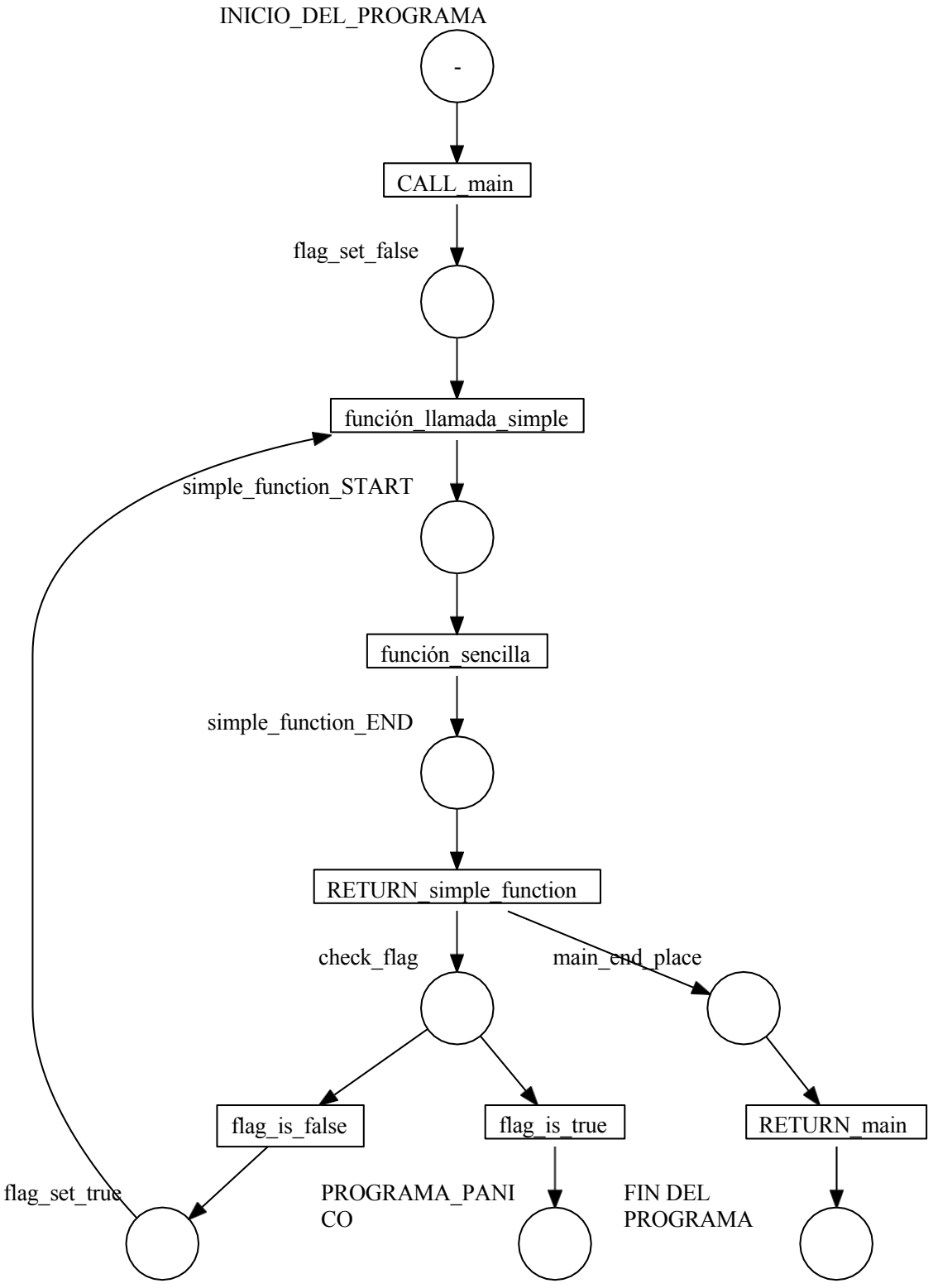


Figura 3.4: Una primera red de Petri (incorrecta) para el código del listado 3.5.

```
1  fn función_sencilla() {}
2
3  pub fn main() {
4      let mut segunda_llamada = false
5      función_sencilla();
6      si segunda_llamada {
7          ¡pánico!
8      }
9      segunda_llamada = true;
10     función_sencilla();
11 }
```

Listado 3.5: Un sencillo programa Rust que llama a una función en dos lugares diferentes.

puede ser consumida por la transición flag_is_false o flag_is_true. Si es consumida por 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 la circunstancia 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.

Lamentablemente, 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 sección 1.1.3, el disparo de las transiciones no es determinista.

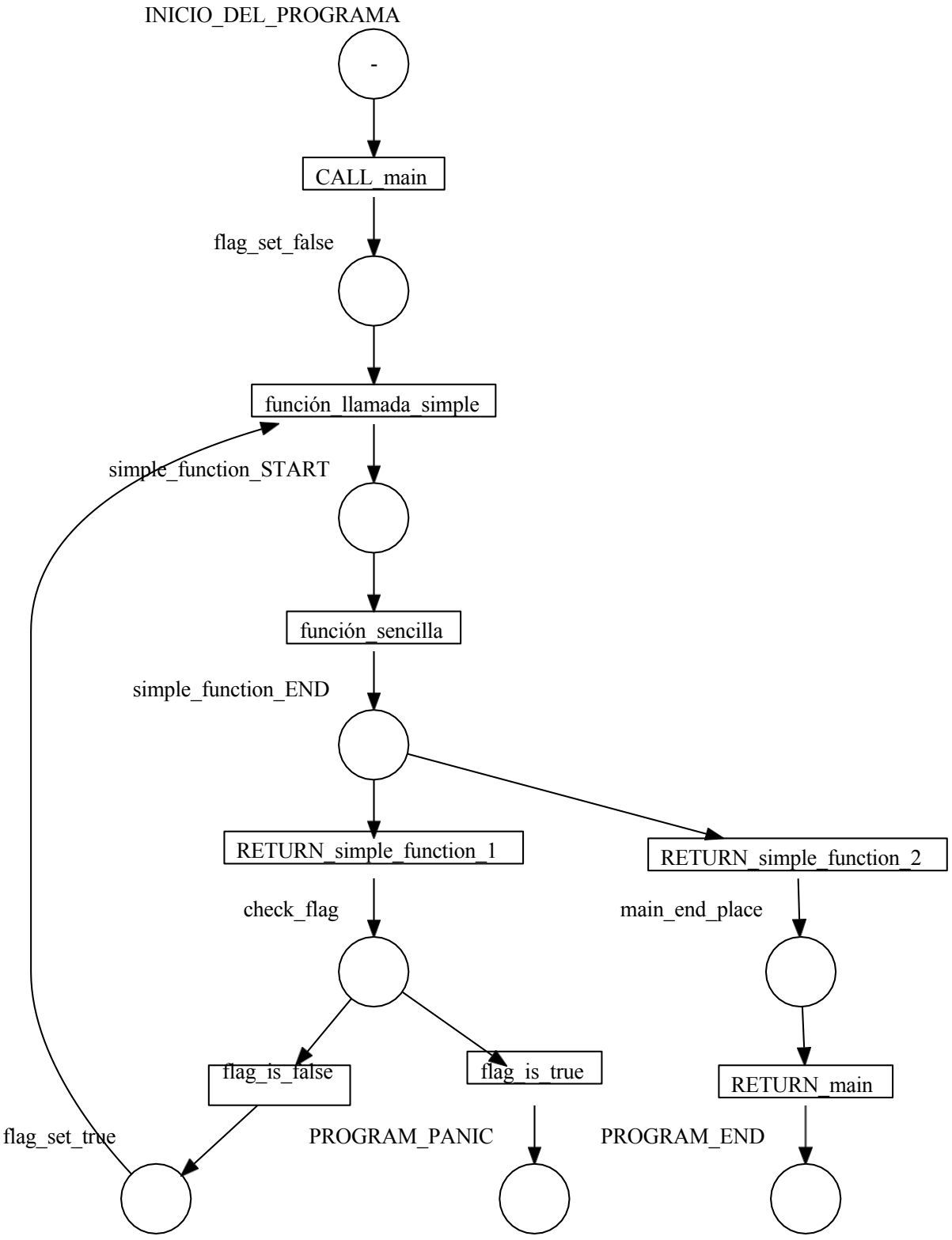


Figura 3.5: Una segunda red de Petri (también incorrecta) para el código del listado 3.5.

### 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 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 *seguridad* de la PN.
- Como el token simboliza el contador del programa, sólo debe haber un token en la PN en un 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 figuras 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 comprobador 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 Secciones 6.1 y 6.2.

Por último, un lector atento puede notar que el análisis de la NP de la Fig. 3.6 lleva a la conclusión de que el programa puede llamar a ¡pánico! 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. La sección 6.6 explora las consecuencias de esta restricción y propone posibles soluciones.

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

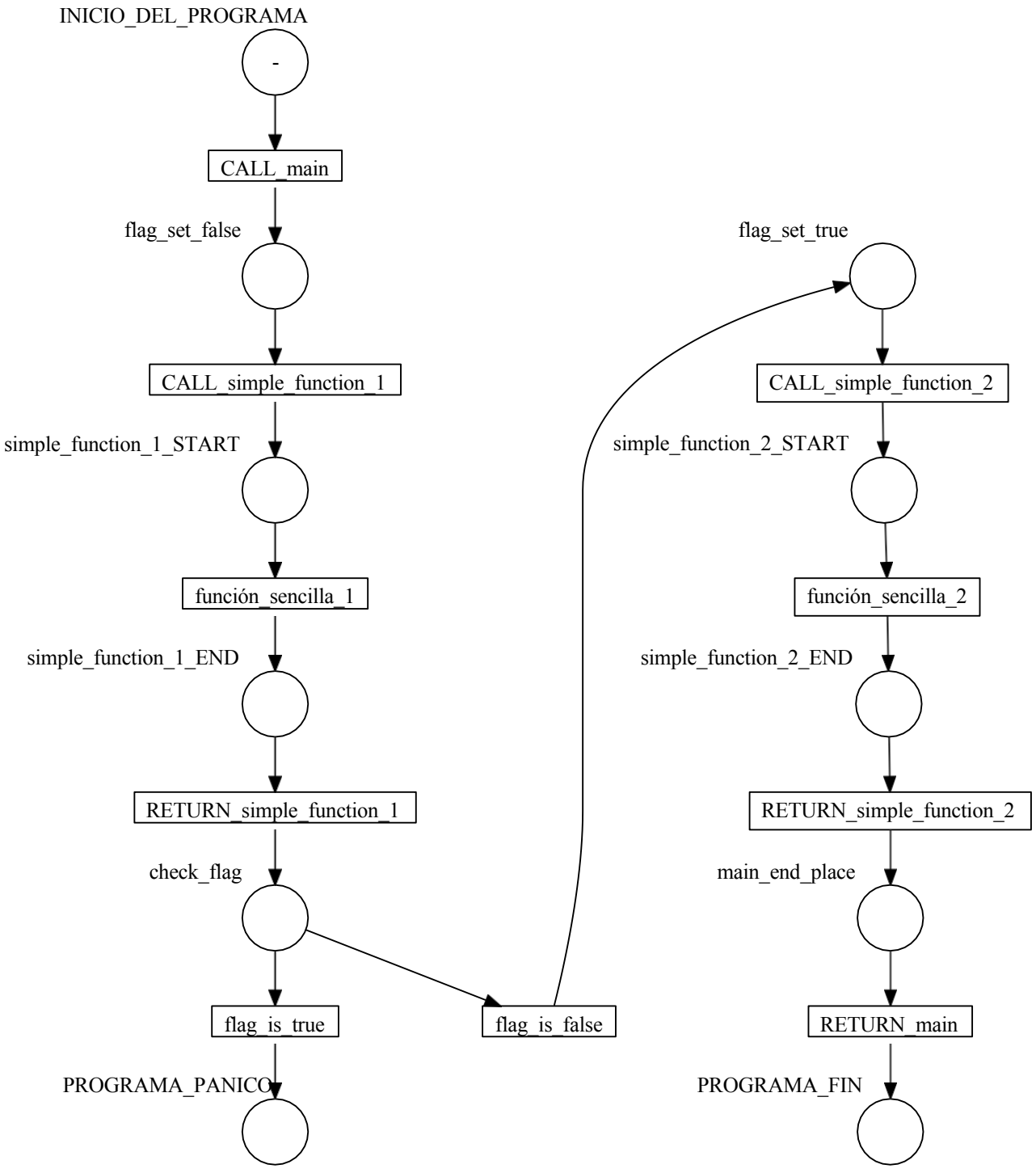


Figura 3.6: Una red de Petri correcta para el código del Listado 3.5 utilizando inlining.

# Capítulo 4

# Realización de la traducción

Este capítulo está dedicado a explorar los detalles de implementación de la herramienta de detección de bloqueos. 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 siguientes, describiremos los componentes centrales de la herramienta de detección de puntos muertos, 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 corrección 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]),
- variables de condición ( std::sync::Condvar[2]),
- contadores de referencia atómicos ( std::sync::Arc[3]).

Aunque se cubren los detalles principales, este capítulo no pretende sustituir a la documentación del 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[45].

[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 5https://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 principal, 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. Éstas 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!

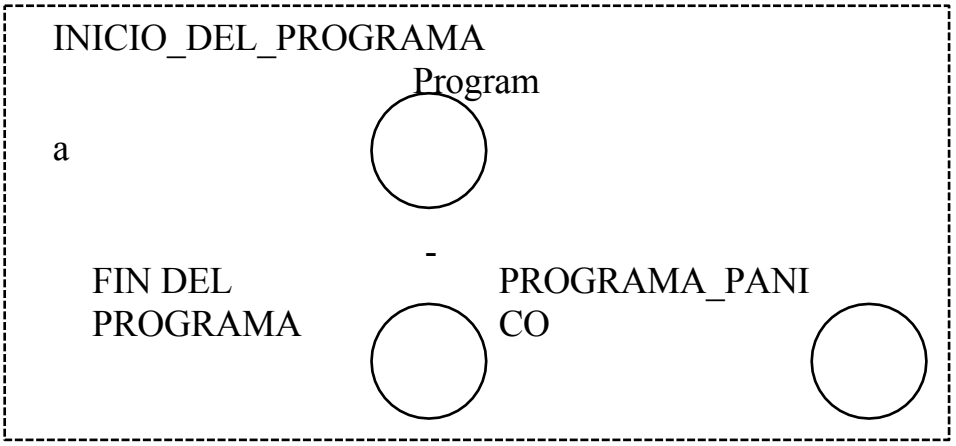


Figura 4.1: Lugares básicos en todo programa Rust.

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 lo tanto, estos programas no deben marcarse en principio como erróneos o defectuosos, pero es aconsejable registrar el estado final a efectos de solución de problemas y depuración. En segundo lugar, incluso si el código del usuario no recurre a ¡pánico! 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 (OOM) o a una

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

errores de hardware, o cuando el SO 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 se percibe típicamente como la primera función que se ejecuta, en realidad no es así. En su lugar, los programas Rust tienen un tiempo de ejecución 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 tiempo de ejecución. Simplemente es más delgado y menos sofisticado. Para los lectores interesados, hace poco se presentó en una conferencia sobre Rust una visita guiada del viaje antes de main [Levick, 2022].

Teniendo esto en cuenta, nos enfrentamos a la cuestión de si debemos incluir este tiempo de ejecución en la traducción PN. 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 tiempo de ejecución es ligeramente diferente para cada SO) 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 principal.

### 4.1.2 Paso de argumentos e introducción de la consulta

La herramienta está diseñada en torno a una sencilla interfaz de línea de comandos (CLI). Tras analizar los argumentos de la línea de comandos utilizando la conocida biblioteca clap [library][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 se describió brevemente en el apartado 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, ya que proporcionan un excelente y breve ejemplo de trabajo 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.

[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

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

```
// Esta puerta de función es necesaria para acceder a las cajas internas del compilador.
// Existe desde hace mucho tiempo y como los internos del compilador nunca serán
↪ estabilizado,
// es probable que la situación siga así.
// <https://doc.rust-lang.org/unstable-book/language-features/rustc-private.html>
#![feature(rustc_private)]

// Los crates del compilador deben importarse de esta forma porque no se publican en
↪ crates.io.
// Estas cajas sólo están disponibles cuando se utiliza la cadena de herramientas nocturna.
// Basta con declararlos una vez para utilizar sus tipos y métodos en toda la caja.
extern crate rustc_ast_pretty;
extern crate rustc_const_eval;
extern crate rustc_driver;
extern crate rustc_error_codes;
extern crate rustc_errors;
extern crate rustc_hash;
extern crate rustc_hir;
extern crate rustc_interface;
extern crate rustc_middle;
extern crate rustc_session;
extern crate rustc_span;
```

Listado 4.1: Extracto del archivo *lib.rs* que muestra cómo utilizar las funciones internas de *rustc*.

El rustc_private es una bandera de función 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 lo tanto, es necesario instalar los componentes adicionales *rustc-dev*, *rust-src*, y *llvm-tools-preview*. El propósito de cada componente se detalla en [Proyecto Rust, 2023d]. En el README[13] del repositorio.

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

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

## 4.2 Llamadas a funciones

### 4.2.1 La pila de llamadas

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 principal, 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*. Cuando se llama a una función en Rust, ésta se introduce en la pila de llamadas, creando un nuevo marco de pila. Un marco de pila 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 (LIFO). A medida que se llaman funciones, cada nuevo marco de la pila 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 lo tanto, la pila de llamadas desempeña un papel esencial en la gestión de las llamadas y retornos de funciones, ya que 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, el reflejo de la pila de llamadas en el traductor es el enfoque más adecuado para el seguimiento de las llamadas a funciones que deben traducirse, ya que se alinea con el flujo lógico de la ejecución del programa. A medida que se traducen las funciones, se empujan y se sacan de la pila de llamadas del traductor, reflejando 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 Traductor tiene una pila 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, ya 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 el cuerpo MIR de una sola función

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

a la vez, lo que puede hacerse utilizando el método optimized_mir[19] método. 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 llega a ellas 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 comience la traducción:

1. Consulta el id del punto de entrada del programa (la función principal).
2. Cree una MirFunction con la información necesaria.
3. Empújelo a la pila.
4. Si es necesario, modifique el contenido de la función MIR mediante peek.
5. Traduce la parte superior de la pila de llamadas.
6. Cuando main termine, elimínelo (pop) de la pila de

llamadas. Cuando se encuentre un terminador de tipo "llamada"

(véase la Sec. 3.4.1):

1. Consulta el id de la función llamada.
2. Cree una MirFunction con la información necesaria.
3. Empújelo a la pila.
4. Si es necesario, modifique el contenido de la función MIR mediante peek.
5. Traduce la parte superior de la pila de llamadas.
6. Cuando la función termine, elimínela (pop) de la pila de llamadas.

Como se ha visto, el enfoque es coherente para cada función MIR y, por tanto, es más fácil de aplicar.

El uso de una pila de llamadas en el proceso de traducción permite el cambio de contexto entre funciones MIR y facilita la posibilidad de volver al bloque básico específico desde el que se llamó a una función. Esto permite 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, emplear el enfoque de la pila de llamadas conlleva ciertas limitaciones. En primer lugar, si la misma función se llama varias veces dentro del programa, también se traducirá varias veces. Esto está relacionado con la estrategia de inlining elaborada en la sección 3.5. Aunque esto puede

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

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

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 encuentra con una función recursiva dentro del proceso de 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 marcos de pila, lo que provoca un desbordamiento de la pila y el consiguiente bloqueo 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 no pueden traducirse utilizando este marco.

### 4.2.3 Funciones extranjeras y funciones de la biblioteca estándar

En Rust, el compilador incluye por defecto la biblioteca estándar en todos los binarios compilados, enlazándola efi cazmente de forma estática. Para anular este comportamiento, se utiliza el atributo a nivel de crate *#![no_std]* para indicar que el crate enlazará con la core-crate en lugar de con la 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 punto muerto. 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 resultante sigue siendo más manejable, lo que facilita el análisis y la verificación de posibles bloqueos 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 una cuidadosa atención. Numerosas funciones de la biblioteca estándar contienen no sólo un lugar final ("bloque de destino", en la jerga *de rustc*) sino también un lugar de limpieza ("bloque de limpieza"). 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 bloqueo más adelante en el proceso de traducción, es imperativo 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.

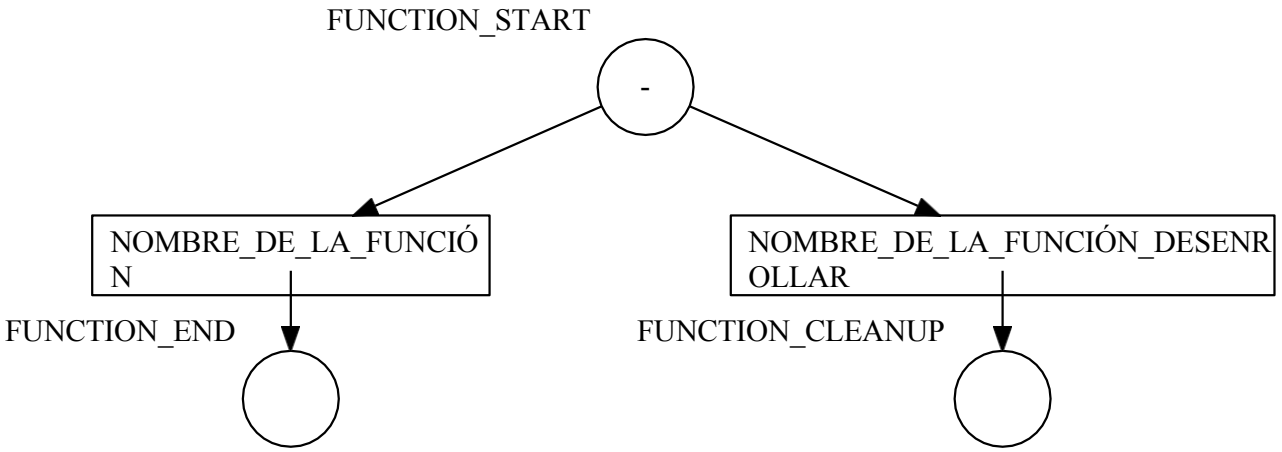


Figura 4.2: El modelo de red de Petri para una función con un bloque de limpieza.

### 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.

- Las funciones forman parte de la biblioteca estándar (la std-crate[21]), salvo la función std::sync::Condvar::wait que se detalla en la sección 4.8.3.
- Funciones que forman parte de la biblioteca central (la core-crate[22]).
- Funciones en la alloc-crate: el núcleo de la biblioteca de asignaciones y colecciones[23].
- Funciones sin representación MIR Esto puede comprobarse con la función is_mir_available método[24].

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

- Funciones que son un elemento externo, es decir, vinculadas 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 están* definidas por el usuario es tratarlas como una función ajena 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 devuelve al llamante. Ejemplos de ello son una envoltura alrededor de un bucle while infinito, una función que sale del proceso o una función que inicia un SO. 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.

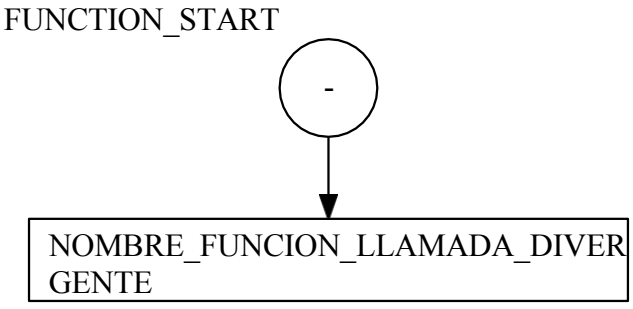


Figura 4.3: El modelo de red de Petri para una función divergente (una función que no retorna).

Tenga en cuenta que este caso especial no constituye un punto muerto y *no* debe tratarse como tal. Un bucle infinito, es decir, una "espera ocupada", es en su naturaleza inherente distinto de la espera infinita que carac- teriza un punto muerto 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 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 al pánico

La macro ¡pánico! 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 al pánico, que es una de las siguientes funciones:

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

- core::pánico::assert_failed
- core::pánico::pánico
- core::pánico::panic_fmt
- std::rt::comenzar_pánico
- 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.

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.

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

Listado 4.2: Un sencillo programa Rust que llama ¡pánico!.

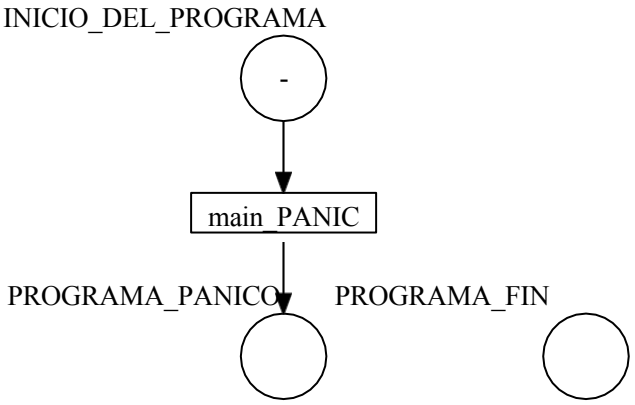


Figura 4.4: El modelo de red de Petri para el Listado 4.2.

## 4.3 Visitante MIR

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 directa por la MIR del código fuente de Rust. En otras palabras, actúa como el pegamento que une a la perfección los distintos c o m p o n e n t e s del traductor.

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

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

No era necesario implementar todos los métodos posibles. Si no se definen, los métodos de MIR Visitor simplemente llaman al supermétodo 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, mu- tex guards, join handles y variables de condición).
- visit_terminator para procesar la sentencia terminadora de cada bloque básico, es decir, conectar los bloques básicos.

Para empezar a visitar el MIR, debe utilizarse el método visit_body. El listado 4.3 muestra la función corre- spondiente en el traductor.

En conclusión, el rasgo MIR Visitor simplifica notablemente la traducción, ya que no es necesario implementar un mecanismo de travesía gracias a las interfaces de compilador proporcionadas. 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 imprima al usuario.

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

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

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

```rust
/// Bucle principal de traducción.
/// Traduce la función desde la parte superior de la pila de llamadas.
/// Dentro del Visitante MIR, cuando se produzca una llamada a otra función, este método será llamado de nuevo
/// para saltar a la nueva f u n c i ó n . Eventualmente se llegará a una "función hoja", las funciones saldrá y el
/// los elementos de la pila se saltarán en orden.
fn translate_top_call_stack(&mut self) {
    let function = self.call_stack.peek();
    // Obtener la representación MIR de la función.
    let body = self.tcx.optimized_mir(function.def_id);
    // Visite el cuerpo MIR de la función utilizando los métodos de `rustc_middle::mir::visit::Visitor`.
    // <https://doc.rust-lang.org/stable/nightly-rustc/rustc_middle/mir/visit/trait.Visitor.html>
    self.visit_body(cuerpo);
    // Finalizado el procesamiento de esta función.
    self.call_stack.pop();
}
```

Listado 4.3: El método del Traductor que inicia el recorrido del MIR.

## 4.4 Función MIR

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 bloques básicos (BB), las sentencias y los terminadores. Estos compo- nentes se introdujeron anteriormente en la Sec. 3.4.1.

La implementación en el repositorio se denomina en consecuencia MirFunction[28]. Este tipo almacena el lugar de inicio y el lugar final de la función. Éstos 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. Consulte la Fig. 3.2 para ver una ilustración.

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 f u n c i ó n . Además, la función requiere un nombre que es diferente para cada

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

llamada a la función, por lo que recibe un nombre con un índice añadido, lo que la hace única en toda la red de Petri.

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, 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, ya 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 las BB y trabajar con un único lugar. La razón de ello es que las conexiones con otras BB dependen únicamente de los terminadores y las declaraciones no se modelan en absoluto, como veremos en breve. Además, la aplicación[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 Declaraciones

Las declaraciones MIR *no se* incorporan intencionadamente al modelo de red de Petri. Teniendo en cuenta que las razones para ello y los beneficios pueden no ser evidentes de inmediato, proporcionaremos una explicación detallada de esta decisión de implementación.

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

El enfoque que se aplicó anteriormente sí incluía el modelado de enunciados. 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 puntos muertos o señales perdidas. Además, inflaba innecesariamente el tamaño de la representación de la 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 que habíamos probado hasta ahora, las declaraciones no realizaban ninguna acción que pudiera justificar su adición a la red de Petri. Al contrario, las redes que incluían las sentencias 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 arreglar las pruebas en consecuencia para que se ajustaran al nuevo resultado.

La alternativa era desactivar las sentencias con una bandera de compilación, 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 resultante.

### 4.4.3 Terminators

Como se vio en la Sec. 3.4.1, las sentencias terminator tienen diferentes formas. La documentación del enum TerminatorKind[31] enumera, en el momento de redactar este documento, 14 variantes diferentes. Se requiere que la implementación admita la mayoría de ellas, ya que aparecen 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 del MIR Visitor[32] incluye el método visit_terminator como se ha visto antes. Aquí es donde se crean los bordes que conectan un BB con otro. En los párrafos siguientes se discuten los detalles de alto nivel de cada manejador. Se omiten algunos detalles de implementación ya que no afectan a la red de Petri.

#### Ir a

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.

[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

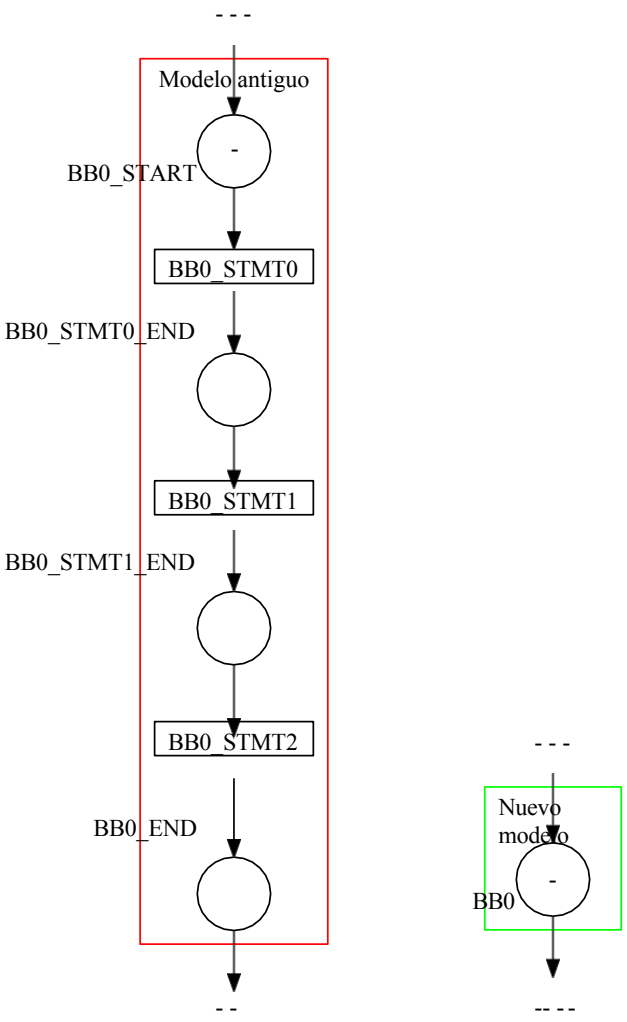


Figura 4.5: Comparación lado a lado de dos posibilidades para modelar las declaraciones MIR.

### 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 mediante 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 condición previa para traducir correctamente varios bloques básicos con un SwitchInt que salte al mismo bloque.

### Reanudar o finalizar

Se trata de terminadores que modelan respectivamente un desenrollado de la pila 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.

### Devuelva

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.

### Inaccesible

Se trata de un caso límite que aparece en algunas coincidencias, bucles while u otras estructuras de control. La documentación lo indica: *Indica un terminador que nunca puede ser alcanzado*. 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. Consulte los comentarios en el repositorio para obtener más detalles.

### Drop

El rasgo std::ops::Drop se utiliza para especificar el código que debe ejecutarse cuando el tipo sale de ámbito [Klabnik y Nichols, 2023, Cap. 15.3]. Es equivalente al concepto de destructores que se encuentra en otros lenguajes de programación.

El terminador de caída 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 lanzando una guardia mutex. 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.

### Llame a

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

Tenga en cuenta 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.

[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

**Afirme**

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

La implementación no modela la condición para la afirmación. 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 de la caída.

## 4.5 Memoria de 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 puntos muertos y señales perdidas. En equipos más sencillos, nos vemos obligados a modelar la memoria únicamente porque es necesario realizar un seguimiento de las variables de sincronización admitidas durante el proceso de traducción.

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

- Mutexes (std::sync::Mutex).
- Guardas Mutex ( std::sync::MutexGuard).
- Unir asas (std::thread::JoinHandle).
- Variables de condición ( std::sync::Condvar).
- Agregados, es decir, envoltorios 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 estar fácilmente disponible para que el traductor opere en el modelo de red de Petri de la variable de sincronización específica.

### 4.5.1 Un ejemplo guiado para introducir los retos

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

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

método está siendo invocado dos veces sobre el mismo mutex. Para detectar este punto muerto, el traductor debe ser capaz, como mínimo, de identificar que la invocación de bloqueo tiene lugar sobre el mismo mutex.

```
fn main() {
  let data = std::sync::Mutex::new(0);
  let _d1 = datos.bloquear();
  let _d2 = data.lock(); // no se puede bloquear, ya que d1 sigue activo
}
```

Listado 4.4: Un punto muerto causado por llamar a lock dos veces en 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 terminador 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.

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, los guardianes del mutex en _2 y _4 serán eventualmente eliminados, lo que indirectamente desbloquea el mutex. Tiene que haber un enlace desde la guardia mutex en _2 y _4 al mutex en
_1. Más concisamente, el traductor debe monitorizar qué mutex está detrás de cada guardia mutex.

Para complicar 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.

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

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

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

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

Listado 4.5: Una excepción de la MIR del programa del Listado 4.4.

### 4.5.2 Una asignación de `rustc_middle::mir::Place` a refe- rencias contadas compartidas

La aplicación se denomina adecuadamente Memoria[36]. Como se anticipó en la sección anterior, hay una instancia de Memoria por cada MirFunción. La memoria está estrechamente conectada al contexto de

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

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". Esto implica asociar un rustc_middle::mir::Lugar 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 y Nichols, 2023, Cap. 15.4].

La memoria utiliza 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 agregado contiene en cambio un vector de valor. 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 mapa hash permite recuperar y gestionar eficazmente los valores asociados durante el proceso de traducción. La memoria también se encarga de proporcionar los typedefs para las diferentes refe- rencias a las variables de sincronización. El listado 4.6 muestra un extracto del archivo 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 Interceptar asignaciones

La pieza que falta en el puzzle 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 Secciones 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 (RHS) se gestionan mediante el mismo mecanismo: El

```rust
#[derive(Default)]
pub struct Memoria<'tcx> {
    map: HashMap<Lugar<'tcx>, Valor>,
}

/// ...

/// Posibles valores que se pueden almacenar en la `Memoria`.
/// Un lugar se asignará a uno de estos.
#[derivar(EcParcial, Clonar)]
pub enum Valor {
    Mutex(MutexRef),
    MutexGuard(MutexGuardRef),
    JoinHandle(ThreadRef),
    Condvar(CondvarRef),
    Agregado(Vec<Valor>),
}

/// ...

/// Una referencia mutex no es más que un puntero compartido al mutex.
pub tipo MutexRef = std::rc::Rc<Mutex>;

/// Una referencia al guardamutex es sólo un puntero compartido al guardamutex.
pub tipo MutexGuardRef = std::rc::Rc<MutexGuard>;

/// Una referencia condvar no es más que un puntero compartido a la variable de condición.
pub tipo CondvarRef = std::rc::Rc<Condvar>;

/// Una referencia al hilo no es más que un puntero compartido al hilo.
pub tipo ThreadRef = std::rc::Rc<Hilo>;
```

Listado 4.6: Resumen de las definiciones de tipos de la implementación de Memoria.

El lado izquierdo (LHS) está vinculado al lado derecho (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, cierres o structs. Requiere un manejo especial ya que el valor

```
fn visita_asignar(
    &mut auto,
    lugar: &rustc_middle::mir::Lugar<'tcx>,
    rvalue: &rustc_middle::mir::Rvalue<'tcx>,
    ubicación: rustc_middle::mir::Ubicación,
) {
    match rvalue {
        rustc_middle::mir::Rvalue::Use(
            rustc_middle::mir::Operand::Copy(rhs) | rustc_middle::mir::Operando::Mover(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::Agregado(_, operandos) => {
            let function = self.call_stack.peek_mut();
            handle_aggregate_assignment(
                lugar,
                &operands.raw,
                &mut función.memoria,
                function.def_id,
                auto.tcx,
            );
        }
        // No es necesario hacer nada para los otros casos por ahora.
        _ => {}
    }

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

Listado 4.7: La implementación personalizada de visit_assign para rastrear variables de sincronización.

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 memoria 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.

## 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

El tiempo de vida de un hilo comienza cuando se inicia invocando la función std::thread::spawn[37] . Ésta recibe como argumento un cierre o función, que representa el código que el nuevo hilo ejecutará concurrentemente con los demás hilos del programa. El hilo engendrado puede empezar a ejecutarse inmediatamente después de ser engendrado, 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 no- ción 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 han- dle 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 la finalización de otro hilo. Al llamar a join sobre un join handle, el hilo llamante se bloquea hasta que el hilo generado 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 del sistema correspondientes. De lo contrario, los hilos que no se unieron correctamente pueden potencialmente perder recursos.

Si se abandona el asa de unión, el hilo ya no puede unirse y se convierte implícitamente en

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

*desacoplado*. 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 éste 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 salido.

### 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. El cierre proporcionado 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 imita los lugares básicos del programa detallados en la Sec. 4.1.1.

Los hilos carecen de un lugar de pánico separado, ya que invocar ¡pánico! 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, es decir, 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 desove donde comienza el hilo.
- La transición de unión (opcional) en la que se utiliza el asa de unión.

- 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 PN 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 generación descrita en la subsección anterior. La rama izquierda es el `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 || {
      // algo de trabajo aquí
  });
  // algo de trabajo aquí
  let _res = thread_join_handle.join();
}
```

Listado 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. Ini- cialmente, 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. Traduzca la llamada a la función utilizando el modelo visto en la Fig. 4.2.
2. Recupera el primer argumento pasado a la `función`: Un valor agregado que contiene las variables capturadas por el cierre y la función que ejecutará el hilo.
3. Extrae el ID de la función que debe ejecutar el hilo.

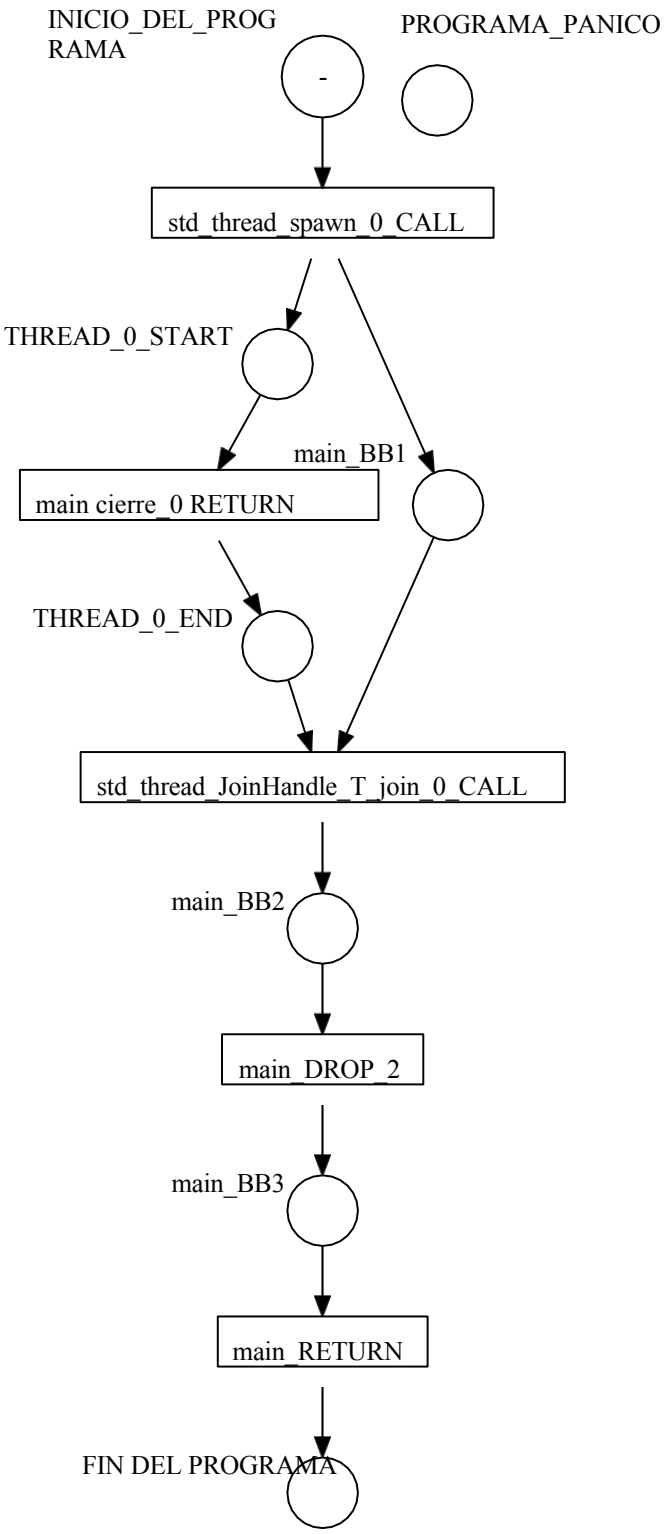


Figura 4.6: El modelo de red de Petri para el programa del listado 4.8.

4. Extraiga los valores capturados por el cierre.
5. Cree un nuevo hilo[39] para almacenar la información necesaria para la traducción retardada.
6. Vincula el valor de retorno de std::thread::spawn, el nuevo asa de unión, al Thread.
7. Empuja el hilo a una cola de hilos detectados en el Traductor.

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

1. Traduzca la llamada a la función utilizando el modelo visto en la Fig. 3.2. Ignore 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. Recupera el primer argumento pasado a la función: El asa de unión. La posición de memoria está vinculada al hilo correspondiente gracias a la interceptación de asignaciones explicada en la Sec. 4.5.3.
3. Establece la transición de unión del hilo subyacente tras el asa de unión.

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

1. Cree un nuevo lugar de inicio y final para el hilo.
2. Conecte la transición de desove al lugar de inicio.
3. Si se ha encontrado una transición de unión, conecte el lugar final a ella.
4. Sustituya el lugar PROGRAM_PANIC por el lugar THREAD_<index>_END para traducir correctamente termi- nantes como Unwind (Sec. 4.4.3).
5. Empuja la función del hilo a la pila de llamadas.
6. Mueva las variables de sincronización a la memoria de la función de hilo, es decir, asigne el valor agregado y sus campos a la memoria de la función de hilo.
7. Traduce la parte 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.

## 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

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

recurso de forma mutuamente excluyente, asegurando que sólo un hilo pueda acceder al recurso a la vez.

En esta sección, se explica el modelo PN 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 bloqueo y puede proceder a acceder al recurso. Si el mutex ya está bloqueado por otro hilo, el hilo que intente adquirir el bloqueo se bloqueará hasta que el bloqueo esté disponible. El método de bloqueo devuelve un guardamutex (std::sync::MutexGuard) que garantiza el acceso exclusivo al recurso hasta que se libere.

A diferencia de la semántica de desbloqueo 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 (RAII) y libera el bloqueo automáticamente cuando sale de ámbito, evitando los bloqueos muertos. Alternativamente, soltar una variable local de tipo std::sync::MutexGuard equivale a desbloquear el mutex correspondiente.

Un mutex puede modelarse en PN 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 testigo 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 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 de caída (Sec. 4.4.3) cuando el lugar de caída 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 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 PN 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 y Liua, 2022].

### 4.7.2 Un ejemplo práctico

Considere el modelo PN mostrado en la Fig. 4.7 correspondiente al programa del Listado 4.4. El MIR se representa 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 al bloqueo en el código fuente. El sistema de indexación refleja el orden de su aparición en el programa, lo que justifica 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 mutex MUTEX_0.

Como ya se ha explicado, los terminadores de caída 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.

Debe tenerse en cuenta que hay más arcos entrantes al lugar mutex que salientes, lo que pone de relieve la importancia de seguir los guardias mutex a lo largo de la MIR utilizando la estrategia explicada en la sección 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. Traduzca la llamada a la función utilizando el modelo visto en la Fig. 4.2.
2. Cree un nuevo Mutex[40] con un índice para identificarlo inequívocamente en toda la PN.
3. Vincula 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. Traduzca la llamada a la función utilizando el modelo visto en la Fig. 3.2. Ignore el lugar de limpieza ya que debemos obligar a la PN a "esperar" a que se marque el lugar mutex. Esto equivale a suponer que la función de bloqueo nunca falla.
2. Recupera la auto-referencia al mutex sobre el que se llama a la función.
3. Añade un arco desde el lugar del mutex subyacente a la transición que representa la llamada a la función.
4. Crea un nuevo MutexGuard con una referencia al Mutex.

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

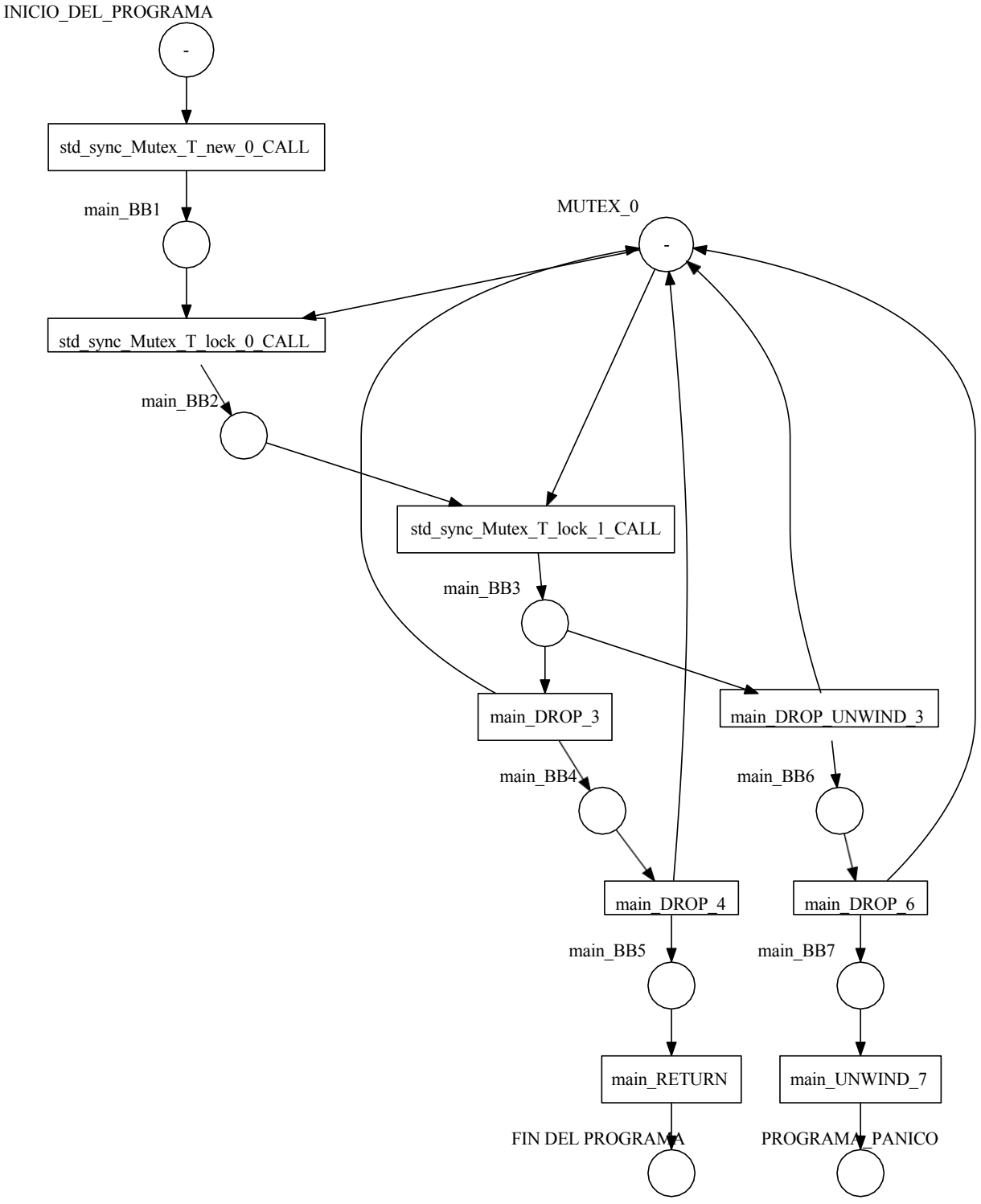


Figura 4.7: El modelo de red de Petri para el programa del listado 4.4.

5. Vincular el valor de retorno de std::sync::Mutex::lock, el nuevo guardamutex, al guardamutex estructura.

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

1. Traduzca la llamada a la función utilizando el modelo visto en la Fig. 4.2.
2. Extrae la variable pasada a la función.
3. Si la variable está vinculada a un guardamutex, añada un arco desde la transición de la llamada a la función hasta el lugar del guardamutex.
4. Si se proporcionó un lugar de limpieza, añada también un arco de desbloqueo desde la transición de limpieza al lugar mutex.

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

1. Si la variable que se va a soltar está vinculada a un guardamutex, añada un arco desde la transición de la llamada a la función hasta el lugar del guardamutex.
2. Si se ha suministrado un lugar de limpieza, añada también un arco de desbloqueo de la transición de desenrollado de la caída al lugar del 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 Variable de condición (`std::sync::Condvar`)

Una variable de condición es una primitiva de sincronización utilizada en la programación concurrente para permitir que los hilos esperen una determinada condición antes de proceder a 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 variable de condición, 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 sección 1.5. La comprensión del comportamiento preciso de las variables de condición en todas las circunstancias es un requisito previo para esta sección.

Esta sección ofrece una explicación detallada del modelo PN utilizado para representar las variables de condición 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 PN debe examinarse detenidamente, ya 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. Por ejemplo, la misma variable de condición podría utilizarse para señalar un número arbitrario de condiciones distintas. En consecuencia, pueden pasarse diferentes mutexes como argumento a la llamada de espera. Además, un número arbitrario de hilos puede bloquearse en una variable de condición y Rust admite la operación de difusión para despertar a todos los hilos en espera a la vez mediante el método notify_all[41] (véase la sección 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 admitidos de las variables de condición para reducir la complejidad de la tarea. Abarcar y tratar todas las posibilidades queda fuera del alcance de esta tesis.

#### Supuestos

1. *Llamada única* : Sólo hay una llamada a esperar por variable de condición, es decir, 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 variable de condición.
3. *Condición booleana*: La condición es una bandera booleana. Se establece o no se establece. 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 la establece a un valor diferente. En términos más sencillos, los hilos que miran el valor, no lo cambian e inmediatamente notifican a la variable de condición *no* son compatibles.
5. *Exclusión de difusión*: El método std::sync::Condvar::notify_all está fuera de alcance.

Podría implementarse el soporte para múltiples llamadas a la espera 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 se establece 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 de palanca superior. Véase la sección 6.6 para algunas reflexiones en este sentido.

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

### Análisis del modelo propuesto

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

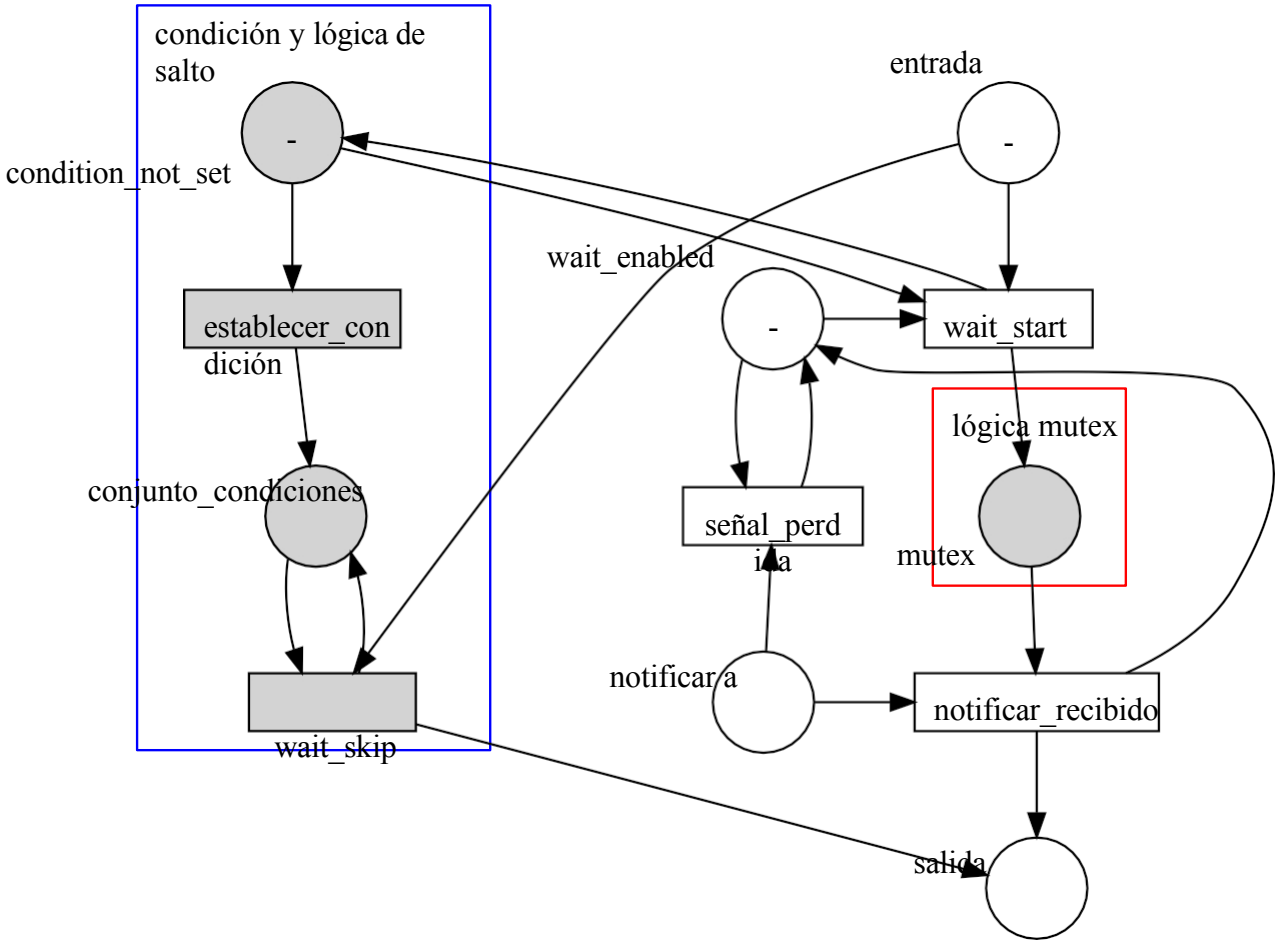


Figura 4.8: El modelo de red de Petri para las variables de condición.

Los lugares de entrada son:

- entrada: 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 se marca cuando la condición es falsa.
- condición_marcada: El lugar se marca cuando la condición es verdadera.
- notificar: El lugar donde el hilo notificador coloca un testigo para despertar al hilo en espera.

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

Existen dos formas posibles de pasar de la entrada a la salida, 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 establecido. El hilo no espera, en su lugar, se salta la espera y alcanza la salida 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.

Observe los arcos entre condition_not_set y wait_start. El testigo 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 las hipótesis 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 importante. 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 variable de condición "acepte" otros hilos (Suposición 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 variable de condición, regenera el testigo en wait_enabled.

### Requisitos de traducción global del modelo de red de Petri

Un reto 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 general cuando se procesa la llamada a la espera. 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 guardia mutex.

Considere el Listado 4.9, otro programa de prueba del repositorio. En la línea 9, la guarda mutex es dereferenciada para escribirle el valor true, estableciendo la condición para la variable condition. 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 en BB), se aproxima lo suficiente y satisface nuestras necesidades.

```
1  fn main() {
2      deje par = std::sync::Arc::new((std::sync::Mutex::new(false),
       ↪  std::sync::Condvar::new()));
3      deje par2 = std::sync::Arc::clone(&par);
4
5      // Dentro de nuestro bloqueo, genere un nuevo hilo y espere a que se inicie.
6      std::thread::spawn(mover || {
7          let (lock, cvar) = &*pair2;
8          let mut iniciado = cerradura.cerradura().desenvolver();
9          *iniciado = verdadero;
10         // Notificamos al condvar que el valor ha cambiado.
11         cvar.notify_one();
12     });
13
14     // Espere a que se inicie el hilo.
15     let (lock, cvar) = &*pair;
16     let mut iniciado = cerradura.cerradura().desenvolver();
17     while !*started {
18         iniciado = cvar.wait(iniciado).unwrap();
19     }
20 }
```

Listado 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 variable de condición ya se haya visto. Todavía podría estar por delante en la ruta de 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 desagradable. 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 PN, es decir, necesitamos información *global* para traducir eficazmente la primitiva de sincronización.

En consecuencia, es inevitable incorporar algún tipo de paso de postprocesamiento a la traducción. Las tareas también deben realizarse en un orden específico. Primero debe descubrirse el mutex. Después puede vincularse a una variable de condición si se encuentra tal variable de condición (el

código fuente es mejor no utilizar ninguna). De ahí que sea aconsejable introducir una noción de "prioridad" en las tareas de postprocesamiento.

El Traductor 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 traductor aborda las tareas de postprocesamiento. Al completarlas según su prioridad, garantizamos que la información esté disponible en el orden requerido.

### Tabla de posibles entradas y salidas previstas

Como complemento a la explicación de los subapartados 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.

| Fila # | Entrada | | | Salida |
|---|---|---|---|---|
| | conjunto_condiciones | wait_enabled | notifique a | *donde el token inicial en la entrada termina* |
| R1 | Falso | Falso | Falso | esperando (esperando una notificación) |
| R2 | Falso | Falso | Verdadero | salida (condición final de espera correcta) |
| R3 | Falso | Verdadero | Falso | entrada (estado inicial) |
| R4 | Falso | Verdadero | Verdadero | *señal perdida (estado transitorio, pasa a R1)* |
| R5 | Verdadero | Falso | Falso | espera (condición establecida, necesita notificación) |
| R6 | Verdadero | Falso | Verdadero | espera (condición final de espera correcta) |
| R7 | Verdadero | Verdadero | Falso | salida (omita la espera) |
| R8 | Verdadero | Verdadero | Verdadero | salida (omita la espera, con señal perdida) |

Tabla 4.1: Resumen de los posibles estados del modelo de red de Petri para las variables de condición.

## 4.8.2 Un ejemplo práctico

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

A pesar de las limitaciones de espacio, el ejemplo del listado 4.10 comprende los elementos centrales del modelo presentado anteriormente. El PN completo 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.

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

2. std_sync_Condvar_notify_one_0_CALL establece un token en CONDVAR_0_NOTIFY.
3. El flujo de tokens continúa hacia main_BB5 justo antes de CONDVAR_0_WAIT_START.

Cabe destacar que no se produce ningún punto muerto si la transición CONDVAR_0_WAIT_START se dispara *antes que* CONDVAR_0_LOST_SIGNAL. En resumen, existe un conflicto entre CONDVAR_0_WAIT_START y CONDVAR_0_LOST_SIGNAL para el testigo en CONDVAR_0_NOTIFY. No obstante, el verificador del modelo 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 las vías de limpieza en la detección de señales perdidas. Si hubiera una segunda transición en el 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 punto muerto seguiría sin detectarse.

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 del modelo para llegar a la conclusión de que la PN nunca se bloquea.

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

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

### 4.8.3 Algoritmos para la traducción de variables de condición

Para concluir esta sección, presentaremos un resumen conciso de los algoritmos utilizados en la traducción de variables de condición. A continuación se incluyen también las adiciones necesarias a los algoritmos mutex.

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

1. Traduzca la llamada a la función utilizando el modelo visto en la Fig. 4.2.
2. Cree una nueva estructura Condvar[43] con un índice para identificarla inequívocamente en toda la PN.
3. Vincula el valor de retorno de std::sync::Condvar::new, la nueva variable de condición, a la variable
   Estructura Condvar.

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

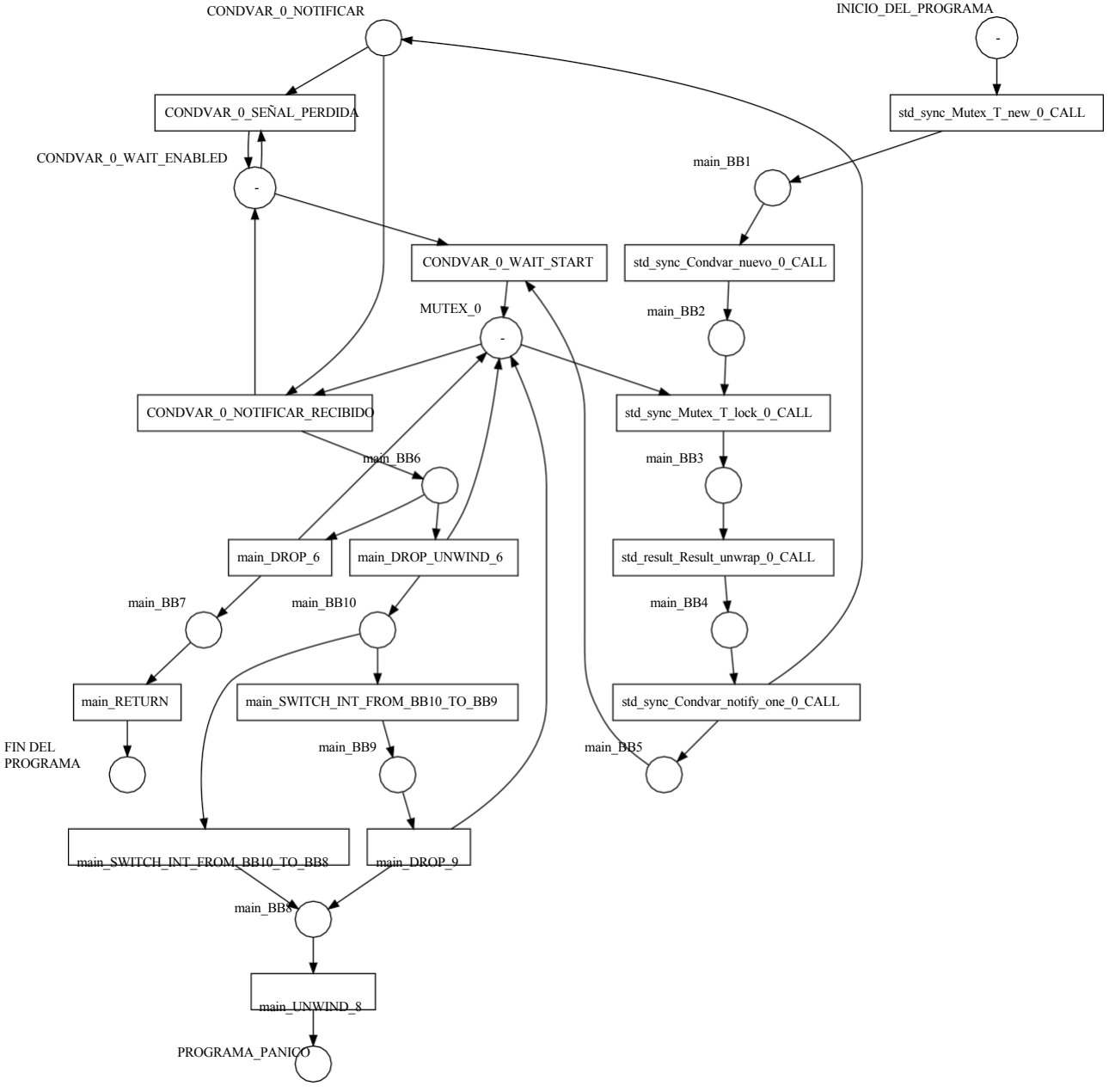


Figura 4.9: El modelo de red de Petri para el programa del listado 4.10.

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

1. Traduzca la llamada a la función utilizando el modelo visto en la Fig. 3.2. Ignore 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 de pérdida. Esto equivale a suponer que la función notify_one nunca falla.
2. Recupera la autorreferencia a la variable de condición sobre la que se llama a la función.
3. Añada un arco desde la transición que representa la llamada a la función hasta el lugar de notificación de la variable de condición subyacente.

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

1. Ignore 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 PN a "esperar" a que se envíe la señal de notificación. Esto equivale a suponer que la función wait o wait_while nunca falla.
2. Recupera la autorreferencia a la variable de condición sobre la que se llama a la función.
3. Extrae el guardamutex pasado a la función.
4. Si la variable de condición ya estaba conectada a una llamada de función, la traducción falla. Esto hace cumplir los supuestos 1 y 2 vistos al principio de la sección.
5. En caso contrario, conecte los lugares de inicio y fin a las tran- siciones wait_start y notify_received respectivamente.
6. Vincula el valor de retorno, el mismo guardamutex que se pasó como argumento, al Estructura MutexGuard.
7. Notifica al traductor que el mutex recibido debe vincularse a este Condvar. Para ello, utilice 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 traductor entra en un bucle para completar las tareas de postprocesamiento por orden de prioridad:

1. Cree al principio del bucle un vector vacío de referencias mutex.
2. Saca del std::collections::BinaryHeap la tarea con la prioridad más baja. Ésta será por diseño una PostprocessingTask::NewMutex. Añada la referencia del mutex al vector.
3. Después de procesar todas las tareas de menor prioridad, el traductor tiene referencias a todos los mutex del código. Continúa sacando tareas de la cola de prioridad.
4. Finalmente, se extrae un PostprocessingTask::LinkMutexToCondvar. Vincula cada mutex a la variable de condición, lo que crea los lugares condition_set y condition_not_set para la condición. También conecta las transiciones deref_mut a estos lugares para establecer el

condición. Por último, conecta los lugares de la condición a las transiciones de la variable de condición para desactivar la espera.

### Modificaciones en los algoritmos mutex

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

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

1. Notifica al traductor que se ha creado un nuevo mutex. Para ello, utilice 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. Compruebe que la auto-referencia es un mutex o un mutex guard.
2. Traduzca la llamada a la función utilizando el modelo visto en la Fig. 3.2. Ignore el lugar de limpieza porque, de lo contrario, cualquier llamada puede fallar, como si la operación de bloqueo de mutex no estuviera presente en el programa, lo que daría lugar a una falsa señal de pérdida. Esto equivale a suponer que la función unwrap nunca falla cuando se aplica a una variable vinculada a un mutex o a una guarda mutex.

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

1. Compruebe que la auto-referencia es un mutex o un mutex guard.
2. Traduzca la llamada a la función utilizando el modelo visto en la Fig. 3.2. Ignore el lugar de limpieza porque, de lo contrario, cualquier llamada puede fallar, como si la operación de bloqueo de mutex no estuviera presente en el programa, dando lugar a una falsa señal de pérdida. 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), extraiga el primer argumento pasado a la función: El mutex o guardia mutex. Añada la transición deref_mut al mutex para establecer la condición de una variable de condición en el paso 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 funda- mentalmente de mayor complejidad y deben manejar casos más límite que los de detección de simples bloqueos causados por un uso incorrecto de los mutexes o por llamar a join en hilos que nunca terminan.

Cabe mencionar que algunos casos límite 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 serían más sencillos. La sección 6.2 se ocupa de la cuestión de no modelar las rutas de limpieza.

# Capítulo 5

# Comprobación de la aplicación

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 corrección de la implementación del software. Se ha desarrollado un completo conjunto de pruebas para cubrir la amplia funcionalidad y comportamiento del traductor y la biblioteca PN.

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 PN. Estas pruebas se dirigen a componentes individuales, examinando a fondo su funcionalidad de forma aislada.

Además de las pruebas unitarias, se ha construido de forma incremental un conjunto de pruebas de integración para evaluar la adherencia del traductor al comportamiento esperado. Estas pruebas consisten en programas de prueba que simulan escenarios sencillos en los que se compara la salida del archivo resultante 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 admitidos.

Además, incorporamos una descripción de cómo generar el MIR y visualizar el resultado de la traducción para ayudar 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 problema de la detección del bloqueo. Por lo tanto, es beneficioso explorar qué características proporciona el verificador de modelos para depurar la traducción PN.

Por último, 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 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 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. También 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 y LoLA) viene con pruebas unitarias para comprobar que la salida se genera correctamente para los siguientes casos sencillos:

- Red vacía.
- Un PN con 5 plazas y 0 transiciones.
- Un PN con 0 plazas y 5 transiciones.
- Un PN con 5 plazas marcadas con diferentes números de fichas.
- Un PN con topología en cadena.
- Un PN 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 commit[2].

### 5.1.2 Pila

La sencilla pila Stack[3] se emplea para implementar la pila de llamadas en el traductor, como se vio en la sección 4.2. Las pruebas unitarias demuestran los métodos admitidos y comprueban algunos casos de uso sencillos.

### 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 Traductor

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

que lleva la cuenta de cuántas veces se ha llamado a cada función para generar un índice incremental único para las etiquetas de transición.

## 5.2 Pruebas de integración

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 un paso firme hacia adelante. Las capacidades de prueba que ofrece Rust, incluido su soporte para pruebas unitarias y de inte- gración, han sido fundamentales 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. A continuación, 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 utilizando las herramientas presentadas en la Sec. 5.3. Se consignó 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].

### 5.2.2 Pruebas de detección de bloqueo

Las pruebas de detección de punto muerto se acercan más a una prueba de extremo a extremo del traductor. Generan el archivo en formato LoLA para el programa de prueba y ordenan al traductor que realice el análisis de bloqueo. El resultado se contrasta entonces con el comportamiento conocido del programa de prueba, es decir, 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.

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

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

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, que está basado en XML.

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

```
1  LUGAR
2      MUTEX_0,
3      FIN_DEL_PROGRA
       MA,
4    PROGRAMA_PANICO,
5  INICIO_DEL_PROGRAM
                  A,
6      main_BB1,
7      main_BB2,
8      main_BB3,
9      main_BB4,
10     main_BB5,
11     main_BB6,
12     main_BB7;
13
14 MARCADO
15     MUTEX_0 : 1,
16     PROGRAM_END : 0,
17     PROGRAMA_PANIC: 0,
       O
18     INICIO_DEL_PRO: 1,
       GRAMA
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
27 TRANSICIÓN main_DROP_3
28   CONSUMIR
29     main_BB3 : 1;
30   PRODUCE
31     MUTEX_0 : 1,
32     main_BB4 : 1;
33 TRANSICIÓN main_DROP_4
34   CONSUMIR
35     main_BB4 : 1;
```

```
36    PRODUCE
37      MUTEX_0 : 1,
38      main_BB5 : 1;
39  TRANSICIÓN main_DROP_6
40    CONSUMIR
41      main_BB6 : 1;
42    PRODUCE
43      MUTEX_0 : 1,
44      main_BB7 : 1;
45  TRANSITION main_DROP_UNWIND_3
46    CONSUMIR
47      main_BB3 : 1;
48    PRODUCE
49      MUTEX_0 : 1,
50      main_BB6 : 1;
51  TRANSITION main_RETURN
52    CONSUMIR
53      main_BB5 : 1;
54    PRODUCE
55      PROGRAM_END : 1;
56  TRANSITION main_UNWIND_7
57    CONSUMIR
58      main_BB7 : 1;
59    PRODUCE
60      PROGRAMA_PANICO : 1;
61  TRANSITION std_sync_Mutex_T_lock_0_CALL
62    CONSUMIR
63      MUTEX_0 : 1,
64      main_BB1 : 1;
65    PRODUCE
66      main_BB2 : 1;
67  TRANSITION std_sync_Mutex_T_lock_1_CALL
68    CONSUMIR
69      MUTEX_0 : 1,
70      main_BB2 : 1;
71    PRODUCE
72      main_BB3 : 1;
73  TRANSITION std_sync_Mutex_T_new_0_CALL
74    CONSUMIR
75      INICIO_PROGRAMA : 1;
76    PRODUCE
77      main_BB1 : 1;
```

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

### 5.2.3 Estructura de la prueba

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

Las pruebas se agrupan en categorías:

- Basic: Para programas básicos como "¡Hola, mundo!" y una simple calculadora aritmética.
- Condvar: Para programas relativos a variables de condición.
- Llamada a función: Para los programas que prueban los diferentes tipos de llamadas de función vistos en la Sec. 4.2.
- Mutex: Para programas que utilizan mutexes.
- Declaración: Para programas que comprueban construcciones específicas como una coincidencia, un bucle infinito, una opción, una llamada a panic! o std::process::abort.
- Hilo: 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 Realización de 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, Cap. 1.6]. Fue un recurso útil para experimentar también con clap para analizar argumentos.

Además, las pruebas de integración utilizan un submódulo compartido [Klabnik y Nichols, 2023, cap. 11.3] que contiene dos cómodas macros que nos ahorran escribir casi todo el código boilerplate. Estas macros se definieron utilizando [Wirth y 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 exhaustividad, el Listado
5.4 representa la función utilizada para las pruebas de traducción.

## 5.3 Visualización del resultado

La visualización del resultado es esencial para comprender el resultado de la detección del bloqueo. Por ello, 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.

```
macro_reglas! generar_pruebas_para_programa_ejemplo {
    ($ruta_programa :literal, $ruta_carpeta_resultado :literal) => {
        #[test]
        fn genera_archivos_de_salida_correctos() {
            super::utils::assert_output_files($ruta_del_programa , $ruta_de_la_carpeta_de_resultados
);
        }
    };
}
```

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

```
mod utilidades;

mod calculadora {
    super::utils::¡generar_pruebas_para_programa_ejemplo!(
        "./ejemplos/programas/básicos/calculadora.rs",
        "./ejemplos/resultados/básico/calculadora/"
    );
}

mod saludar {
    super::utils::¡generar_pruebas_para_programa_ejemplo!(
        "./ejemplos/programas/basic/greet.rs",
        "./ejemplos/resultados/básico/greet/"
    );
}

mod hello_world {
    super::utils::¡generar_pruebas_para_programa_ejemplo!(
        "./ejemplos/programas/basic/hola_mundo.rs",
        "./ejemplos/resultados/básico/hola_mundo/"
    );
}
```

Listado 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(fichero_codigo_origen: &str, carpeta_salida: &str) {
    let mut cmd = Comando::cargo_bin("cargo-check-deadlock").expect("Comando no encontrado");

    // El directorio de trabajo actual es siempre la carpeta raíz del proyecto
    cmd.arg("comprobar-bloqueo")
        .arg(archivo_código_fuente)
        .arg(format!("--carpeta_salida={carpeta_salida}"))
        .arg("--punto")
        .arg("--pnml")
        .arg("--filename=prueba")
        .arg("--skip-analysis");

    cmd.assert().success();

    for extensión in ["lola", "punto", "pnml"] {
        let ruta_salida = PathBuf::from(format!("{carpeta_salida}prueba.{extension}"));
        let ruta_salida_esperada = PathBuf::from(format!("{carpeta_salida}net.{extension}"));

        let contenido_archivo =
            std::fs::read_to_string(&output_path).expect("No se pudo leer el archivo de salida en
            ↪ cadena");

        let contenido_archivo_esperado = std::fs::read_to_string(&ruta_de_salida_esperada)
            .expect("No se ha podido leer el archivo con el contenido esperado a cadena");

        if contenido_archivo != contenido_archivo_esperado {
            ¡pánico!
                "El contenido de {} no coincide con el contenido de {}",
                ruta_salida.to_string_lossy(),
                ruta_salida_esperada.to_string_lossy()
            );
        }

        std::fs::eliminar_archivo(ruta_salida).expect("No se pudo eliminar el archivo de salida");
    }
}
```

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

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 bandera correspondiente: rustc --emit=mir <ruta_al_código_fuente>.

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 sección 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].

Ejecute dot -Tpng net.dot -o outfile.png para generar una imagen PNG a partir del archivo .dot resultante. Ejecute dot -Tsvg net.dot -o outfile.svg para generar una imagen SVG a partir del archivo .dot resultante. 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 "Ejecutar" 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

El programa soporta las banderas de verbosidad definidas en el crate clap_verbosity_flag[14]. Por ejemplo, ejecutar el programa con la bandera -vvv imprime mensajes de depuración que pueden ser útiles para localizar qué línea de la MIR no se está traduciendo correctamente.

A continuación, deberá invocar la herramienta del siguiente modo:

cargo check-deadlock <ruta_al_programa>/programa_oxido.rs -vvv

---

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

El comprobador de modelos LoLA admite la impresión de una "ruta testigo" que muestra una secuencia de disparos de transición que conducen a un punto muerto. Esto es extremadamente útil cuando se amplía el traductor y el PN 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.

```
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 LoLA para el programa del Listado 4.4.

## 5.4 Integración de LoLA en 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 bloqueos. Integrarlo no fue, por desgracia, trivial.

### 5.4.1 Compilación

En primer lugar, la compilación a partir del código fuente no funcionaba en el hardware de que disponíamos. 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 de trabajo del comprobador de modelos, indispensable para realizar el análisis de punto muerto, 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 de nuestro 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 canalización CI/CD mediante GitHub Actions. Un usuario también puede copiar este ejecutable para instalar LoLA en lugar de compilarlo desde cero.

Al final, 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:

1. Utilizando scripts de compilación (build.rs) como se describe en el Libro de Cargo [Proyecto Rust, 2023a, Cap. 3.8].
2. Modificación de LoLA para convertirla en una biblioteca.
3. Mueva un ejecutable precompilado a la carpeta de instalación cuando ejecute cargo install.
4. Defina LoLA como un binario en el Cargo.toml [Proyecto Rust, 2023a, Cap 3.2.1] y, con toda esperanza, se moverá al directorio Cargo bin.
5. Defina LoLA como ejemplo en el Cargo.toml [Proyecto Rust, 2023a, Cap. 3.2.1] y, con suerte, se moverá al directorio Cargo.toml.
6. Utilice una herramienta de construcción de uso general como make.

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 activos arbitrarios. En resumen, *no* está pensada para ser una herramienta de compilación de uso general como make.

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

### 5.4.3 Expresar la propiedad a comprobar

El tercer reto es encontrar una fórmula de lógica computacional de árbol* (CTL*) para ordenar a LoLA que busque los puntos muertos. Por suerte, podemos reutilizar la fórmula encontrada en [Meyer, 2020]:

EF (BLOQUEO Y (FIN_PROGRAMA = 0 Y PÁNICO_PROGRAMA = 0))

La fórmula representa la propiedad a comprobar. Cabe destacar que no todos los bloqueos son interesantes para nuestro análisis. Nuestro objetivo es identificar los casos de bloqueos muertos en los que la ejecución del programa se bloquea *inesperadamente*. Este escenario se alinea con una PN muerta como se ve en la definición 14, en la que no se habilita ninguna transición y la PN alcanza un estado final. Sin embargo, debemos actuar con cautela, ya que hay casos en los que *se espera que* la NP esté *m u e r t a* , como cuando el programa termina o entra en pánico. Estos son estados en los que la ejecución normalmente se detiene. Por lo tanto, si llegamos al lugar PROGRAM_END o PROGRAM_PANIC, la ejecución ha sido satisfactoria, no se trata de un punto muerto 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 bloqueo. 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 una trayectoria 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 Programas de pruebas notables

Para concluir este capítulo, presentamos dos programas de prueba notables que ilustran las capacidades actuales de la herramienta desarrollada en esta tesis. Nuestra intención es inspirar a otros para que contribuyan a este `proyecto` o, `como` mínimo, generar interés en el campo de la comprobación de modelos.

En primer lugar, el Listado 5.5 muestra una versión sencilla del famoso Problema de los Filósofos Comensales pro- puesto por Dijkstra. Esta versión, cariñosamente apodada "Cenar Filósofos", tiene sólo dos filósofos y dos tenedores en la mesa. Es necesario bloquear un mutex para acceder a cada tenedor. Cuando los filósofos intentan coger ambos tenedores para comer, el programa se bloquea, lo que es fácil de comprobar por inspección. Este punto muerto es detectado con éxito por la herramienta. Además, en el repositorio[18] se incluye una versión más compleja con 5 filósofos, para la que también se detecta el punto muerto. Se ha omitido aquí debido a las limitaciones de espacio.

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

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

fn main() {
    let bifurcación0 = Arco::nuevo(Mutex::nuevo(0));
    let bifurcación1 = Arco::nuevo(Mutex::nuevo(1));

    let filósofo0 = {
        let horquilla_izquierda = horquilla0.clonar();
        let horquilla_derecha = horquilla1.clonar();
        thread::spawn(mover || {
            let _left = horquilla_izquierda.lock().unwrap();
            let _right = horquilla_derecha.lock().unwrap();
        })
    };

    let filósofo1 = {
        let horquilla_izquierda = horquilla1.clonar();
        let horquilla_derecha = horquilla0.clonar();
        thread::spawn(mover || {
            let _left = horquilla_izquierda.lock().unwrap();
            let _right = horquilla_derecha.lock().unwrap();
        })
    };

    // Esperar a que terminen todos los hilos
    filosof0.join().unwrap();
    filósofo1.join().unwrap();
}
```

Listado 5.5: Una versión reducida del problema del filósofo comedor que se bloquea.

En segundo lugar, observe el programa del listado 5.6. Modela el problema clásico de productor-consumidor. Utiliza una variable de condición y un búfer con capacidad para un solo elemento. El acceso al búfer está protegido por un mutex. El productor genera 10 elementos secuencialmente y el consumidor los procesa a medida que están disponibles. La herramienta verifica con éxito la ausencia de bloqueo en el programa.

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

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

    let productor_buffer = buffer.clone();
    let consumidor_buffer = buffer.clone();

    let _producer = thread::spawn(move || {
        para i en 1..10 {
            let (bloqueo, cvar_productor, cvar_consumidor) = &*productor_buffer;
            let mut buffer = lock.lock().unwrap();

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

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

            cvar_consumer.notify_one();
        }
    });

    let _consumer = thread::spawn(move || loop {
        let (bloqueo, cvar_productor, cvar_consumidor) = &*consumidor_buffer;
        let mut buffer = lock.lock().unwrap();

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

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

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

Listado 5.6: Una solución al problema productor-consumidor.

# Capítulo 6

# Trabajo

# futuro

## 6.1 Reducción del tamaño de la red de Petri en el postprocesado

[Murata, 1989] describe en una sección titulada "Reglas de reducción sencillas para el análisis" seis operaciones que preservan las propiedades de seguridad, liveness y boundedness de PN. Consulte las definiciones 13, 14 y 12 respectivamente para refrescar el significado de estas propiedades.

Las seis operaciones implican simplificaciones que reducen el número de lugares o transiciones en la red de Petri. A continuación reproducimos los nombres utilizados para las reglas de reducción en el documento y la Fig. 6.1 representa la transformación que tiene lugar en cada caso.

a) Fusión de lugares en serie.

b) Fusión de transiciones en serie.

c) Fusión de lugares paralelos.

d) Fusión de transiciones paralelas.

e) Eliminación de lugares de bucle propio.

f) Eliminación de transiciones de bucle propio.

Como estas operaciones no afectan a la propiedad de liveness, el resultado de la detección del bloqueo permanece inalterado. En consecuencia, puede resultar ventajoso reducir el tamaño de la PN tras el proceso de traducción utilizando métodos específicos disponibles en la biblioteca netcrab. Este paso debe realizarse después de traducir todos los hilos pero antes de invocar al verificador de modelos.

Incorporar esta funcionalidad a la propia biblioteca PN sería más adecuado, ya que permitiría que otras aplicaciones se beneficiaran de esta característica. Sería interesante investigar si este enfoque resulta útil cuando se traducen programas más grandes que contienen cientos o miles

de lugares y transiciones.

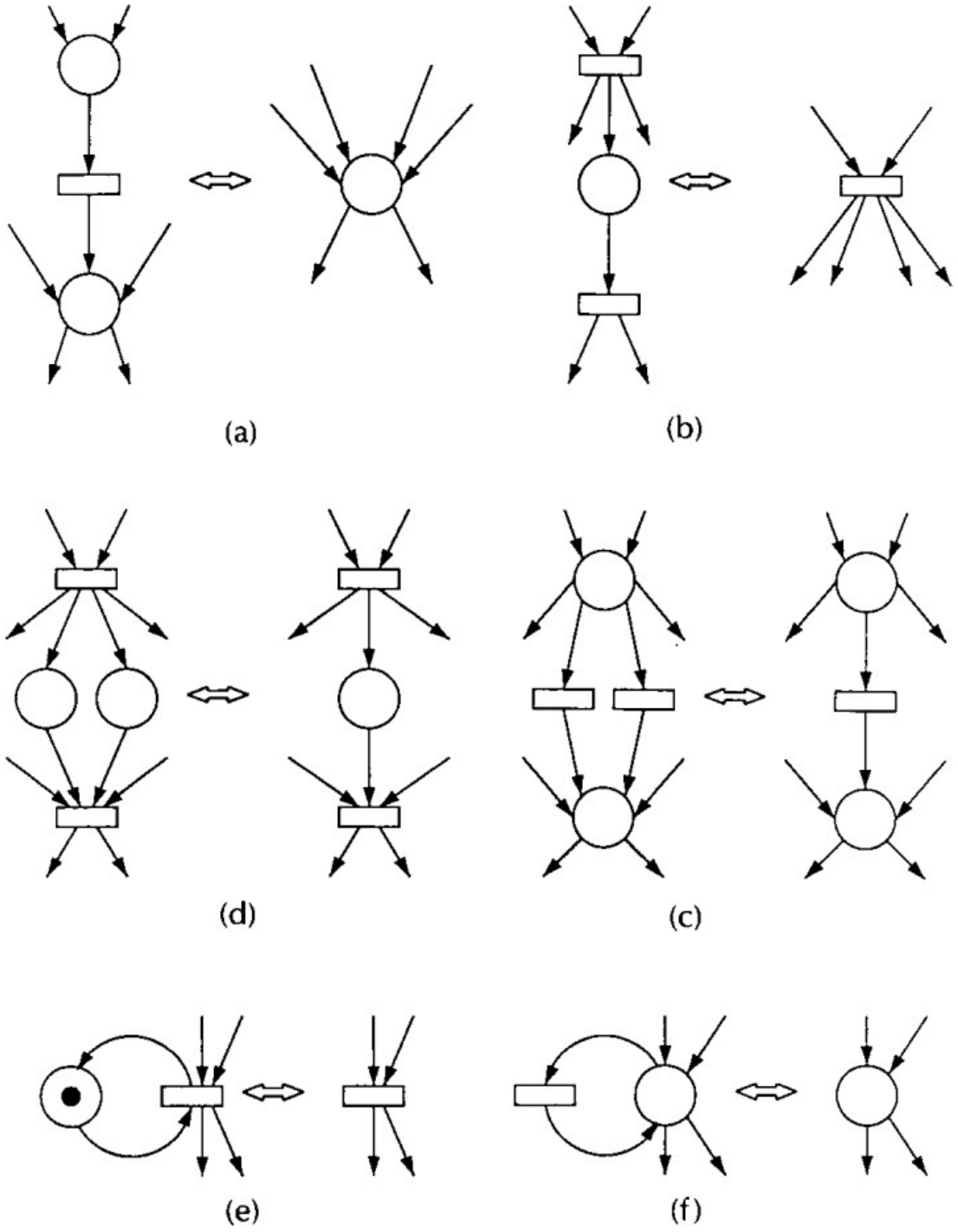


Figura 6.1: Las reglas de reducción presentadas en el documento de Murata.

Un inconveniente notable de la aplicación de estas operaciones es que podría ocultar el origen del bloqueo. Es valioso para el usuario disponer de información precisa sobre la línea del código fuente

donde se produce el punto muerto. Si las transiciones o lugares correspondientes que representan esta línea se fusionan, esta información se pierde. Sin embargo, esta desventaja puede considerarse aceptable cuando se trata de modelos extensos, y la función podría activarse o desactivarse a discreción del usuario.

## 6.2 Eliminación de las rutas de limpieza de la traducción

El mecanismo de gestión de errores de la MIR debe tener en cuenta todos los escenarios posibles de fallo durante el tiempo de ejecución. El objetivo del compilador *rustc* es garantizar que el código compilado falle con gracia, incluso en las circunstancias más extremas, por ejemplo, cuando el programa se queda sin memoria o las llamadas al sistema fallan inesperadamente debido a límites duros en los recursos disponibles u otras causas. Sin embargo, la mayor parte de este código de limpieza de salvaguarda nunca se ejecuta en la práctica. Los errores OOM y los fallos del sistema operativo son poco frecuentes y, si llegan a producirse, un bloqueo en el código de usuario es el menor de nuestros problemas.

[Meyer, 2020] argumenta que el programa siempre terminará en un estado final de pánico una vez que falle una sola llamada a función o aserción. En lugar de traducir el camino alternativo que sigue la ejecución en la MIR, propone establecer un token en el lugar PROGRAM_PANIC directamente. Esto equivale a ignorar el BB de destino de limpieza específico durante el proceso de traducción y conectar el BB al lugar PROGRAM_PANIC como si fuera un terminador Unwind (Sec. 4.4.3).

Esto reduce sustancialmente el tamaño del modelo de red de Petri. Tiene el inconveniente de que se visitan BB de limpieza pero nunca se conectan a otras BB. Éstas deben eliminarse en un paso de postprocesamiento para no desordenar el modelo final. La implementación de Meyer no parece haber realizado este paso crucial. No está claro si la implementación se ajusta a lo propuesto en la tesis porque el código fuente ya no puede compilarse y no hay ejemplos de salida en el repositorio[1].

La afirmación de que el estado de pánico es irrecuperable requiere un examen exhaustivo, ya que hemos observado anteriormente en la introducción a Rust que el programador tiene la opción de utilizar std::panic::catch_unwind. Además, este razonamiento intuitivo podría pasar por alto situaciones en las que se produce un bloqueo tras un pánico. Esto no tiene por qué ser un fallo catastrófico. Considere, por ejemplo, un hilo que se bloquea mientras espera un mensaje de otro hilo que entró en pánico debido a una entrada incorrecta del usuario.

En conclusión, esta modificación de la lógica de traducción parece prometedora para reducir significativamente el número de lugares y transiciones en la PN, especialmente en los modelos más grandes, pero es necesario seguir investigando.

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

## 6.3 Caché de funciones traducidas

Una caché que almacene funciones después de traducirlas es una optimización interesante que explorar. El objetivo es evitar traducciones redundantes de la misma función cuando se la llama varias veces dentro del programa. Esta idea ya se mencionó brevemente (pero no se implementó) en [Meyer, 2020]. La implementación actual no incorpora tales mecanismos de almacenamiento en caché.

Esta caché tendría que almacenar un PN distinto para cada función. Podría realizarse como un HashMap<rustc_hir::def_id::DefId, PetriNet>, análogo al contador de funciones ya presente en la implementación. Además, el proceso de traducción necesitaría fusionar/conectar las redes Petri resultantes de cada función traducida. Este paso de fusión requiere el apoyo de la biblioteca de PN netcrab para combinar las múltiples subredes en un todo cohesivo.

Sin embargo, conectar las redes de Petri individuales no es una tarea trivial, ya que una función puede llamar a un número arbitrario de otras funciones. En consecuencia, determinar los "puntos de contacto" apropiados en los que debe conectarse la subred se convierte en una ardua tarea. La existencia potencial de numerosos puntos de contacto, derivados de los distintos patrones de llamada a funciones, complica así el proceso de fusión.

Además, las redes de Petri para cada función deben tener etiquetas que sean inequívocas en todo el programa o al menos al exportarlas al formato para el comprobador de modelos. Esto requiere generar versiones ligeramente diferentes de la misma función para cada llamada, lo que en parte desprecia las ventajas de tener una caché en primer lugar.

Por último, algunas funciones pueden no almacenarse en caché en absoluto en caso de que existan efectos secundarios especiales. Esto ocurre, por ejemplo, con todas las primitivas de sincronización soportadas actualmente. Su traducción debe gestionarse individualmente.

## 6.4 Recursión

La recursividad en las llamadas a funciones plantea un reto en las NPs cuando se definen como en la definición 1 debido a la incapacidad de mapear adecuadamente los valores de los datos al modelo. La PN carece del poder expresivo necesario para representar esto de forma compacta.

El número de veces que se llama a una función recursiva depende en última instancia de los datos con los que se llama y no puede determinarse en tiempo de compilación. En la ejecución normal de un programa, una función recursiva es empujada a un nuevo marco de pila repetidamente hasta que se alcanza el caso base o la pila se desborda. Sin embargo, en PN, la llamada a la función en la que se alcanza el caso base no puede distinguirse de las demás, a menos que de algún modo los tokens que representan los niveles de recursividad sean distintos.

[Meyer, 2020, Sec. 3.4.2] analiza este problema y propone utilizar redes de Petri de alto nivel, es decir, redes de Petri coloreadas (CPN) para resolverlo. Las redes de Petri de alto nivel ofrecen una posible solución al permitir la distinción entre tokens y la anotación de tokens con los niveles de recursión correspondientes. Sin embargo, esto requiere una reconsideración seria de toda la lógica de traducción

debido al diferente formalismo. Al utilizar CPN cada transición se convierte en una función generalizada de fichas de entrada de un tipo específico que genera fichas del mismo tipo o de un tipo diferente. La red de Petri resultante es sustancialmente más compleja y no todos los verificadores de modelos admiten CPN.

Las estrategias de mitigación tampoco proporcionan consuelo en este caso. Por un lado, se podría intentar detectar la recursividad y detener la traducción, pero la recursividad puede mostrar patrones inusuales que no son triviales de detectar. Por ejemplo, considere una función A que llama a una función B que llama a una función C que finalmente vuelve a llamar a A. Este ciclo de recursión puede ser arbitrariamente largo y añadir esta capacidad al traductor no añade mucho valor en comparación con simplemente ignorar el problema y llegar a un desbordamiento de pila.

Por otro lado, [Meyer, 2020] sugiere modelar cada nivel de recursión hasta una profundidad máxima fija, pero esto afectaría a los resultados de la verificación, ya que las propiedades de los programas podrían variar con diferentes profundidades máximas de recursión. Para cada profundidad de recursión máxima $N$ , se puede construir un programa contraejemplo que muestre un comportamiento diferente, por ejemplo, un bloqueo, en la profundidad de recursión $N + 1$, evitando así la detección.

## 6.5 Mejoras en el modelo de memoria

A pesar de la aparentemente sencilla implementación, idear un modelo de memoria que funcione en todos los casos es una tarea difícil. Dicho esto, el modelo actual es ante todo una buena primera aproximación y la solución también tiene sus inconvenientes.

Aún no se admite el paso de variables entre funciones MIR. Se trata de un inconveniente importante, ya que debe resolverse para admitir la llamada a métodos en bloques impl que reciban variables de sincronización. Para esta tesis, bastó con escribir los programas de forma simplificada para evitar esta limitación, pero en un caso real, esto no es factible.

Existe un acoplamiento significativo entre las funciones que gestionan las llamadas a funciones del módulo std::sync de la biblioteca estándar y la Memoria. Una interfaz más generalizada podría ser útil para añadir compatibilidad con bibliotecas externas.

La idea de la "vinculación" funciona bien, pero no se ajusta a la semántica de los programas Rust. A la larga, sería preferible borrar el mapeo si la variable se traslada a una función diferente. Tomar referencias también debería tratarse como un caso distinto al de simplemente copiar o utilizar la variable.

El tamaño inicial del std::collections::HashMap podría optimizarse para el número medio de variables locales en una función MIR típica. Esto podría ser un parámetro de configuración de la herramienta.

## 6.6 Modelos de nivel superior

El campo de los modelos de redes de Petri de nivel superior es vasto y abarca numerosas ramas y metodologías potenciales. Explorar este dominio ofrece un amplio abanico de posibilidades para avanzar en las capacidades de modelado.

Un avance notable reside en la utilización de redes de Petri coloreadas (CPN). De este modo, los valores de los datos podrían modelarse como fichas de distintos tipos, mejorando así la expresividad y la precisión de la representación de la red de Petri. En el capítulo siguiente se presenta un artículo relacionado con este tema. [Meyer, 2020] también mencionó los modelos de nivel superior al hablar de las mejoras de su semántica de redes de Petri para Rust. Para una introducción a las redes de Petri de nivel superior, véase [Murata, 1989].

Otra intrigante adición al actual modelo de red de Petri implica la incorporación de arcos inhibidores. Estos arcos proporcionan un medio para modelar condiciones en el código fuente en las que se comprueba la presencia de un valor cero. Al introducir arcos inhibidores, las redes de Petri pueden capturar eficazmente situaciones en las que se requiere la ausencia de un token específico para que se produzca una transición. Por ejemplo, al comprobar una bandera booleana utilizada como condición para una variable de condición. Los arcos inhibidores elevan el poder expresivo de las redes de Petri al nivel de las máquinas de Turing [Peterson, 1981].

# Capítulo 7

# Trabajos relacionados

En [Rawson y Rawson, 2022], los autores proponen un modelo generalizado basado en redes de Petri coloreadas e implementan un marco de trabajo middleware de código abierto en Rust[1] para construir, diseñar, simular y analizar las redes de Petri resultantes.

Las redes de Petri coloreadas (CPN) son un tipo de red de Petri que puede representar sistemas más complejos que las redes de Petri tradicionales. En una CPN, las fichas tienen asociado un valor específico, que puede representar diversos atributos o propiedades del sistema que se está modelando. Esto permite un modelado más detallado y preciso de los sistemas del mundo real, incluidos aquellos con estructuras de datos y comportamientos complejos. En la representación visual, cada token tiene un color (análogo a un tipo en los lenguajes de programación) y las transiciones esperan tokens de un determinado color (tipo) y pueden generar tokens del mismo color o tokens de un color diferente. Como ejemplo breve, considere una transición con dos lugares de entrada y uno de salida que representa la mezcla de colores primarios. Si los colores de las fichas de entrada son rojo y azul, entonces el color de la ficha de salida es morado. Si los colores de las fichas de entrada son amarillo y azul, entonces el color de la ficha de salida es verde.

El modelo propuesto por los autores es un tipo de red de Petri aún más general, denominado Redes de Petri de transición no terminística (NT-PN), que permite que las transiciones se disparen sin que todos sus lugares de entrada estén marcados con fichas, al tiempo que permite que cada transición defina qué lugares de salida deben marcarse en función de la entrada. En otras palabras, cada transición de- fine reglas arbitrarias para que se produzca su disparo. Explican brevemente cómo podría analizarse la red de Petri para resolver el número máximo de hilos útiles para ejecutar la tarea modelada en ella. También mencionan el paso de modelado como herramienta para comprobar si hay estados erróneos antes de desplegar un sistema electrónico o informático.

En [De Boer et al., 2013] se presenta una traducción de un lenguaje formal a redes de Petri para la de- tección de bloqueos en el contexto de objetos activos y futuros. El lenguaje formal elegido es Concurrent Reflective Object-oriented Language (Creol). Se trata de un modelado orientado a objetos

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

lenguaje diseñado para especificar sistemas distribuidos. En este trabajo, el programa está formado por objetos activos que se comunican asíncronamente y en el que se utilizan futuros para manejar los valores de retorno, que pueden recuperarse mediante una primitiva get que retiene el bloqueo (bloqueante) o una primitiva claim que libera el bloqueo (no bloqueante). Tras traducir el programa a una red de Petri, se aplica el análisis de alcanzabilidad para detectar los bloqueos. Este artículo muestra que también es posible una traducción de las estrategias de comunicación asíncrona a redes de Petri con el objetivo de detectar los bloqueos.

# Capítulo 8

# Conclusione

# s

Esta tesis ha explorado la traducción de programas Rust a modelos de redes de Petri con el fin de detectar puntos muertos y señales perdidas. A lo largo del estudio, se han examinado diversos aspectos del proceso de transla- ción, incluido el manejo de llamadas a funciones, hilos, mutexes y variables de condición. El traductor que hemos desarrollado ha demostrado su capacidad para capturar con precisión el comportamiento de concurrencia y sincronización de programas Rust bastante sencillos.

El enfoque de traducción presentado en esta tesis ha mostrado resultados prometedores, modelando y detectando con éxito bloqueos en una serie de programas de prueba, que comprenden incluso dos problemas clásicos de la programación concurrente. Al aprovechar el poder expresivo de las redes de Petri, el traductor proporciona una representación visual del comportamiento del programa, facilitando la identificación de posibles problemas de sincronización. Y lo que es más importante, la traducción produce un modelo que puede ser analizado por una miríada de herramientas de comprobación de modelos, aprovechando el trabajo académico existente para aportar soluciones a los problemas de la industria. La incorporación de un modelo sucinto para las variables de condición mejora las capacidades de modelado y permite la detección de señales omitidas, que son una clase más intrincada de bloqueo en los sistemas concurrentes.

De cara al futuro, existen varias vías de investigación y mejora. Una dirección potencial es la exploración de programas más complejos y aplicaciones del mundo real para evaluar la escalabilidad y eficacia del enfoque de traducción. Además, un mayor refinamiento y optimización de los algoritmos de traducción podría mejorar la eficacia del análisis, especialmente modelos de más alto nivel que permitirían modelar la memoria con mayor eficacia.

En general, esta tesis ha supuesto una contribución significativa al desarrollar un traductor que tiende un puente entre los programas Rust y las redes de Petri. Los conocimientos obtenidos en esta investigación han arrojado luz sobre los retos y las oportunidades de modelar y analizar sistemas concurrentes en tiempo de compilación. Idealmente, un lenguaje de programación cuyo compilador detectara los prob- lemas de concurrencia sería una bendición para muchas

aplicaciones. Aprovechando los puntos fuertes de las redes de Petri, esta posibilidad podría avanzar aún más en el lenguaje de programación Rust.

En otro orden de cosas, la contribución de esta tesis va más allá de los beneficios inmediatos de

el traductor propuesto y sus capacidades. Al proporcionar una base sólida y bien documentada para la traducción de programas Rust a redes de Petri, este trabajo pretende hacer una contribución significativa a la comunidad Rust en su conjunto. Sirve de trampolín para futuros empeños, ofreciendo una base fiable sobre la que pueden construirse otras herramientas y proyectos de investigación. Abre nuevas posibilidades para explorar el análisis y la verificación de programas Rust concurrentes utilizando redes de Petri. Esto, a su vez, tiene el potencial de impulsar nuevos avances en el campo, estimulando la innovación y promoviendo una comprensión más profunda de la programación concurrente en Rust. Con su completa documentación y su clara implementación, el traductor no sólo facilita su uso inmediato, sino que también sirve como valioso recurso para quienes estén interesados en estudiar o ampliar las técnicas de traducción empleadas. En última instancia, este trabajo aspira a encender la curiosidad e inspirar nuevas contribuciones al ecosistema Rust, fomentando la colaboración y el crecimiento de la comunidad.

# Bibliografía


[Aho et al., 2014] Aho, A. V., Lam, M. S., Sethi, R., y Ullman, J. D. (2014). *Compiladores: Principles, Techniques, and Tools*. Pearson Education, 2 edición.

[Albini, 2019] Albini, P. (2019). RustFest Barcelona - Envío de un compilador estable cada seis semanas. https://www.youtube.com/watch?v=As1gXp5kX1M. Consultado el 2023-02-24.

[Arpaci-Dusseau y Arpaci-Dusseau, 2018] Arpaci-Dusseau, R. H. y Arpaci-Dusseau, A. C. (2018). *Sistemas operativos: Tres piezas fáciles*. Arpaci-Dusseau Books, 1.00 edición. https://pages.cs.wisc.edu/~remzi/OSTEP/.

[Ben-Ari, 2006] Ben-Ari, M. (2006). *Principios de programación concurrente y distribuida*. Pearson Education, 2ª edición.

[Bernstein et al., 1987] Bernstein, P. A., Hadzilacos, V., Goodman, N., et al. (1987). *Concurrency control and recovery in database systems*, volumen 370. Addison-Wesley Reading.

[Carreño y Muñoz, 2005] Carreño, V. y Muñoz, C. (2005). Verificación de la seguridad del concepto de operaciones del sistema de transporte de aeronaves pequeñas. En *AIAA 5th ATIO and 16th Lighter- Than-Air Sys Tech. and Balloon Systems Conferences*, página 7423.

[Chifflier y Couprie, 2017] Chifflier, P. y Couprie, G. (2017). Escribiendo analizadores sintácticos como si fuera 2017. En *2017 IEEE Security and Privacy Workshops (SPW)*, páginas 80-92. IEEE.

[Coffman et al., 1971] Coffman, E. G., Elphick, M., y Shoshani, A. (1971). Bloqueos de sistemas. *ACM Computing Surveys (CSUR)*, 3(2):67-78.

[Corbet, J. (2022). Ya está disponible el núcleo 6.1. https://lwn.net/Articles/917504/. Consultado el 2023-02-24.

[Coulouris et al., 2012] Coulouris, G., Dollimore, J., Kindberg, T., y Blair, G. (2012). *Sistemas Dis- tribuidos, Conceptos y Diseño*. Pearson Education, 5ª edición.

[Czerwiński et al., 2020] Czerwiński, W., Lasota, S., Lazić, R., Leroux, J., y Mazowiecki, F. (2020). El problema de alcanzabilidad de las redes de Petri no es elemental. *Journal of the ACM (JACM)*, 68(1):1-28. https://arxiv.org/abs/1809.07115.

[Davidoff, 2018] Davidoff, S. (2018). Cómo la biblioteca estándar de Rust fue vulnerable durante años y nadie se dio cuenta. https://shnatsel.medium.com/how-rusts-standard-library-was-v ulnerable-for-years-and-nobody-noticed-aebf0503c3d6. Consultado el 2023-02-20.

[De Boer et al., 2013] De Boer, F. S., Bravetti, M., Grabe, I., Lee, M., Steffen, M., y Zavattaro, G. (2013). A petri net based analysis of deadlocks for active objects and futures. En *Formal Aspects of Component Software: 9th International Symposium, FACS 2012, Mountain View, CA, EE.UU., 12-14 de septiembre de 2012. Revised Selected Papers 9*, páginas 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*, páginas 65-138. Springer Nueva York, Nueva York, NY.

[Esparza y Nielsen, 1994] Esparza, J. y Nielsen, M. (1994). Cuestiones de decidibilidad para redes de Petri. *BRICS Report Series*, 1(8). https://tidsskrift.dk/brics/article/download/216 62/19099/49254.

[Fernandez, S. (2019). Un enfoque proactivo para un código más seguro. https: //msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/. Consultado el 2023-02-24.

[Gansner et al., 2015] Gansner, E. R., Koutsofios, E., y North, S. C. (2015). *Dibujar gráficos con puntos*.

[García, E. (2022). Lenguajes de programación avalados para su uso del lado del servidor en Meta. https://engineering.fb.com/2022/07/27/developer-tools/programming-languages -aprobado-para-uso-servidor-en-meta/. Consultado el 2023-02-24.

[Gaynor, 2020] Gaynor, A. (2020). Lo que la ciencia puede decirnos sobre la seguridad de C y C++. https://alexgaynor.net/2020/may/27/science-on-memory-unsafety-and-security/. Consultado el 2023-02-24.

[Habermann, 1969] Habermann, A. N. (1969). Prevención de los bloqueos de sistemas. *Communica- tions of the ACM*, 12(7):373-ff.

[Hansen, 1972] Hansen, P. B. (1972). Multiprogramación estructurada. *Comunicaciones de la ACM*, 15(7):574-578.

[Hansen, 1973] Hansen, P. B. (1973). *Principios de sistemas operativos*. Prentice-Hall, Inc.

[Heiner, 1992] Heiner, M. (1992). Validación de software basada en redes de Petri. *International Computer Science Institute ICSI TR-92-022, Berkeley, California*.

[Hillah y Petrucci, 2010] Hillah, L. M. y 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). Monitores: Un concepto de estructuración del sistema operativo.
*Comunicaciones de la ACM*, 17(10):549-557.

[Holt, 1972] Holt, R. C. (1972). Some deadlock properties of computer systems. *ACM Com- puting Surveys (CSUR)*, 4(3):179-196.

[Hosfelt, 2019] Hosfelt, D. (2019). Implicaciones de reescribir un componente del navegador en Rust. https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/. Consultado el 2023-02-24.

[Howarth, 2020] Howarth, J. (2020). Por qué Discord está cambiando de Go a Rust. https://discord.com/blog/why-discord-is-switching-from-go-to-rust. Consultado el 2023-03-20.

[Huss, 2020] Huss, E. (2020). Espacio en disco y mejoras en LTO. https://blog.rust-lang. org/inside-rust/2020/06/29/lto-improvements.html. Consultado el 2023-04-06.

[Jaeger y Levillain, 2014] Jaeger, E. y Levillain, O. (2014). Mind your language(s): Un debate sobre los idiomas y la seguridad. En *2014 IEEE Security and Privacy Workshops*, páginas 140-151. IEEE.

[Jannesari et al., 2009] Jannesari, A., Bao, K., Pankratius, V., y Tichy, W. F. (2009). Helgrind+: Un eficiente detector dinámico de carreras. En *2009 IEEE International Symposium on Parallel & Distributed Processing*, páginas 1-13. IEEE.

[Jüngel et al., 2000] Jüngel, M., Kindler, E., y Weber, M. (2000). El lenguaje de marcado de redes Petri. *Petri Net Newsletter*, 59(24-29):103-104.

[Proyecto Kani, 2023] Proyecto Kani (2023). El verificador de óxido Kani. https://model-checkin g.github.io/kani/. Consultado el 2023-05-30.

[Karatkevich y Grobelna, 2014] Karatkevich, A. y Grobelna, I. (2014). Detección de puntos muertos en redes de Petri: ¿una traza para un punto muerto? En *2014 7th International Conference on Human System Interactions (HSI)*, páginas 227-231. IEEE.

[Kavi et al., 2002] Kavi, K. M., Moshtaghi, A., y Chen, D.-J. (2002). Modelado de aplicaciones multihilo mediante redes de Petri. *Revista internacional de programación paralela*, 30:353-371.

[Kavi et al., 1996] Kavi, K. M., Sheldon, F. T., y Reed, S. (1996). Especificación y análisis de sistemas en tiempo real mediante csp y redes de Petri. *Revista internacional de ingeniería del software e ingeniería del conocimiento*, 6(02):229-248.

[Kehrer, 2019] Kehrer, P. (2019). Inseguridad de la memoria en los sistemas operativos de A p p l e . https://langui.sh/2019/07/23/apple-memory-safety/. Consultado el 2023-02-24.

[Klabnik y Nichols, 2023] Klabnik, S. y Nichols, C. (2023). *The Rust programming language*. No Starch P r e s s . https://doc.rust-lang.org/stable/book/.

[Klock, F. S. (2022). Contribuyendo a Rust: Bootstrapping the Rust Compiler (rustc). https://www.youtube.com/watch?v=oG-JshUmkuA. Consultado el 2023-04-08.

[Knapp, 1987] Knapp, E. (1987). Detección de puntos muertos en bases de datos distribuidas. *ACM Computing Surveys (CSUR)*, 19(4):303-328.

[Kordon et al., 2022] Kordon, F., Bouvier, P., Garavel, H., Hulin-Hubard, F., Amat., N., Am- parore, E., Berthomieu, B., Donatelli, D., Dal Zilio, S., Jensen, P., Jezequel, L., He, C., Li, S., Paviot-Adet, E., Srba, J., y Thierry-Mieg, Y. (2022). Resultados completos de la edición 2022 del concurso de comprobación de modelos. http://mcc.lip6.fr/2022/results.php.

[Kordon et al., 2021] Kordon, F., Hillah, L. M., Hulin-Hubard, F., Jezequel, L., y Paviot- Adet, E. (2021). Estudio de la eficacia de las técnicas de comprobación de modelos utilizando los resultados del mcc de 2015 a 2019. *Revista internacional sobre herramientas de software para la transferencia de tecnología*.

[Küngas, 2005] Küngas, P. (2005). La comprobación de alcanzabilidad de redes de Petri es polinómica con jerarquías de abstracción óptimas. En *Abstracción, Reformulación y Aproximación: 6th International Symposium, SARA 2005, Airth Castle, Escocia, Reino Unido, 26-29 de julio de 2005. Actas 6*, páginas 149-164. Springer. PDF disponible en el perfil público de ResearchGate.

[Levick, 2022] Levick, R. (2022). Rust Before Main - Rust Linz. https://www.youtube.com/watch?v=q8irLfXwaFM. Consultado el 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. Consultado el 2023-04-14.

[Mayr, 1981] Mayr, E. W. (1981). An algorithm for the general petri net reachability problem. En *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, página 238-246, Nueva York, NY, EE.UU.. Asociación para la Maquinaria Informática.

[Meyer, 2020] Meyer, T. (2020). A Petri Net semantics for Rust. Tesis de máster, 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. Consultado el 2023-02-24.

[Monzón y Fernández-Sánchez, 2009] Monzón, A. y Fernández-Sánchez, J. L. (2009). Evaluación del riesgo de bloqueo en modelos arquitectónicos de sistemas en tiempo real. En *2009 IEEE In- ternational Symposium on Industrial Embedded Systems*, páginas 181-190. IEEE.

[Moshtaghi, 2001] Moshtaghi, A. (2001). Modeling Multithreaded Applications Using Petri Nets. Tesis de máster, Universidad de Alabama en Huntsville.

[Mozilla Wiki, 2015] Mozilla Wiki (2015). Proyecto Oxidación. https://wiki.mozilla.org/O xidation. Consultado el 2023-03-20.

[Murata, 1989] Murata, T. (1989). Redes de Petri: Propiedades, análisis y aplicaciones. *Proceed- ings of the IEEE*, 77(4):541-580. http://www2.ing.unipi.it/~a009435/issw/extra/mu rata.pdf.

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

[Nichols et al., 1996] Nichols, B., Buttlar, D., y Farrell, J. (1996). *Programación Pthreads: Un estándar POSIX para un mejor multiprocesamiento*. O'Reilly Media, Inc.

[Oaten, T. (2023). La brujería de Rust. https://www.youtube.com/watch?v=MW RPYBoCEaY. Consultado el 2023-04-08.

[Perronnet et al., 2019] Perronnet, F., Buisson, J., Lombard, A., Abbas-Turki, A., Ahmane, M., y El Moudni, A. (2019). Prevención de atascos de vehículos autoconducidos en una red de intersecciones. *IEEE Transactions on Intelligent Transportation Systems*, 20(11):4219-4233.

[Peterson, 1981] Peterson, J. L. (1981). La *teoría de redes de Petri y el modelado de sistemas*. 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 y Rawson, 2022] Rawson, M. y Rawson, M. (2022). Petri nets for concurrent programming. *arXiv preprint arXiv:2208.02900*.

[Reid, 2021] Reid, A. (2021). Herramientas de verificación automática de Rust (2021). https://alastairre id.github.io/automatic-rust-verification-tools-2021/. Consultado el 2023-02-20.

[Reid et al., 2020] Reid, A., Church, L., Flur, S., de Haas, S., Johnson, M., y Laurie, B. (2020). Towards making formal methods normal: meeting developers where they are. Ac- ceptado en HATRA 2020.

[Reisig, 2013] Reisig, W. (2013). *Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies*. Springer-Verlag Berlin Heidelberg, 1ª edición.

[Rust CLI WG, 2023] Rust CLI WG (2023). Aplicaciones de línea de comandos en Rust. https://rust-cli.github.io/book/. Consultado el 2023-06-08.

[Grupo de trabajo sobre Rust en dispositivos embebidos, 2023] Grupo de trabajo sobre Rust en dispositivos embebidos (2023). The Embedded Rust Book. https://docs.rust-embedded.org/book/. Consultado el 2023-06-02.

[Proyecto Óxido, 2023a] Proyecto Óxido (2023a). The Cargo Book. https://doc.rust-lang.org/ cargo/. Consultado el 2023-06-08.

[Proyecto Rust, 2023b] Proyecto Rust (2023b). The rustc Book. https://doc.rust-lang.org/ rustc/. Consultado el 2023-02-20.

[Proyecto Rust, 2023c] Proyecto Rust (2023c). El Rustonomicón. https://doc.rust-lang.org/nomicon/. Consultado el 2023-04-19.

[Proyecto Rust, 2023d] Proyecto Rust (2023d). El libro rustup. https://rust-lang.github.io/rustup/index.html. Consultado el 2023-05-02.

[Proyecto Rust, 2023e] Proyecto Rust (2023e). The Unstable Book. https://doc.rust-lang. org/unstable-book/the-unstable-book.html. Consultado el 2023-04-14.

[Savage et al., 1997] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., y 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 un analizador de bajo nivel. En *Application and Theory of Petri Nets 2000: 21st International Conference, ICATPN 2000 Aarhus, Denmark, June 26-30, 2000 Proceedings 21*, páginas 465-474. Springer.

[Shibu, 2016] Shibu, K. V. (2016). *Introducción a los sistemas empotrados*. McGraw Hill Education (India), 2ª edición.

[Silva y Dos Santos, 2004] Silva, J. R. y Dos Santos, E. A. (2004). Aplicación de las redes de Petri a la validación de requisitos. *IFAC Proceedings Volumes*, 37(4):659-666.

[Simone, 2022] Simone, S. D. (2022). Linux 6.1 añade oficialmente soporte para Rust en el núcleo. https://www.infoq.com/news/2022/12/linux-6-1-rust/. Consultado el 2023-02-24.

[Singhal, 1989] Singhal, M. (1989). Detección de puntos muertos en sistemas distribuidos. *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. Consultado el 2023-02-22.

[Stepanov, 2020] Stepanov, E. (2020). Detecting Memory Corruption Bugs With HWASan. https://android-developers.g o o g l e b l o g . c o m / 2 0 2 0 / 0 2 / d e t e c t i n g - m e m o r y - c o r r u p t i o n-bugs-with-hwasan.html. Consultado el 2023-02-24.

[Stoep y Hines, 2021] Stoep, J. V. y Hines, S. (2021). Rust en la plataforma Android. https://security.googleblog.com/2021/04/rust-in-android-platform.html. Consultado el 2023-02-22.

[Stoep y Zhang, 2020] Stoep, J. V. y Zhang, C. (2020). Queue the Hardening Enhancements. https://android-developers.googleblog.com/2020/02/detecting-memory-corruption-bugs-with-hwasan.html. Consultado el 2023-02-24.

[Szekeres et al., 2013] Szekeres, L., Payer, M., Wei, T., y Song, D. (2013). Sok: Guerra eterna en la memoria. En *2013 IEEE Symposium on Security and Privacy*, páginas 48-62. IEEE.

[Los proyectos Chromium, 2015] Los proyectos Chromium (2015). Memory safety. https://www.chromium.org/Home/chromium-security/memory-safety/. Consultado el 2023-02-24.

[Los desarrolladores del proyecto Rust, 2019] Los desarrolladores del proyecto Rust (2019). Caso práctico de Rust: La comunidad hace de rust una elección fácil para npm. https://www.rust-lang.org/static/pdf s/Rust-npm-Whitepaper.pdf.

[Thierry Mieg, 2015] Thierry Mieg, Y. (2015). Symbolic Model-Checking using ITS-tools. En *Tools and Algorithms for the Construction and Analysis of Systems*, volumen 9035 de *Lecture Notes in Computer Science*, páginas 231-237, Londres, Reino Unido. Springer Berlin Heidelberg.

[Thompson, C. (2023). How Rust went from a side project to the world's most-loved programming language. https://www.technologyreview.com/2023/02/14/10 67869/rust-worlds-fastest-growing-programming-language/.

[Toman et al., 2015] Toman, J., Pernsteiner, S., y Torlak, E. (2015). Crust: a bounded veri- fier for rust (n). En *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, páginas 75-80. IEEE.

[Van der Aalst, 1994] Van der Aalst, W. (1994). Putting high-level petri nets to work industry. *Computers in industry*, 25(1):45-54.

[van Steen y Tanenbaum, 2017] van Steen, M. y Tanenbaum, A. S. (2017). *Sistemas distribuidos*. Pearson Education, 3ª edición.

[Weber y Kindler, 2003] Weber, M. y Kindler, E. (2003). The Petri Net Markup Language. *Tecnología de redes de Petri para sistemas basados en la comunicación: Advances in Petri Nets*, páginas 124- 144.

[Wirth y Keep, 2023] Wirth, L. y Keep, D. (2023). El pequeño libro de las macros de Rust. https://veykril.github.io/tlborm/introduction.html. Consultado el 2023-06-08.

[Wu y Hauck, 2022] Wu, Y. y Hauck, A. (2022). Cómo construimos Pingora, el proxy que conecta Cloudflare a Internet. https://blog.cloudflare.com/how-we-built-pingo ra-the-proxy-that-connects-cloudflare-to-the-internet/. Consultado el 2023-03-20.

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