Universidad de la República
Facultad de Ingeniería

Simulación de programas concurrentes en contexto educativo

Memoria de proyecto presentada a la Facultad de Ingeniería de la Universidad de la República por

Alejandro Clara
Leandro Paz

en cumplimiento parcial de los requerimientos
para la obtención del título de
Ingeniero en Computación.

Tutores

Federico Rivero	Universidad de la República
Pablo Ezzatti	Universidad de la República

Tribunal

Jorge Merlino	Universidad de la República
Juan García Garland	Universidad de la República
Ximena Otegui	Universidad de la República

Montevideo
jueves 12 enero, 2023

Simulación de programas concurrentes en contexto educativo, Alejandro Clara Leandro Paz.

Esta tesis fue preparada en LaTeX usando la clase iietesis (v1.1).
Contiene un total de 55 páginas.
Compilada el jueves 12 enero, 2023.
http://iie.fing.edu.uy/

Tabla de contenidos

Tabla de contenidos

Capítulo 1

Introducción

La Unidad Curricular (UC) Sistemas Operativos de la Facultad de Ingeniería (FING) de la Universidad de la República (UDELAR) de Uruguay, es una UC de la carrera de Ingeniería en Computación, cuyo objetivo es introducir a los estudiantes los conceptos básicos de los Sistemas Operativos y de Programación Concurrente. La UC no solo es una asignatura obligatoria para el título de Ingeniero en Computación, sino también para el título intermedio (Analista en Computación). Se sugiere que sea cursada en el quinto semestre. Estas características, además de su conocida dificultad para los estudiantes, hace que sea una asignatura masiva, donde en promedio por semestre cuenta con alrededor de 400 estudiantes, entre un 70 % y 90 % más de estudiantes que otras asignaturas de la misma carrera sugeridas para el mismo semestre[1].

Este proyecto tiene como objetivo identificar y/o desarrollar herramientas que ayuden a los estudiantes en el proceso de aprendizaje del tema Programación Concurrente, tema que es identificado como una de las principales dificultades para los estudiantes a la hora de la aprobación de la asignatura. Específicamente, se busca la implementación o adopción de un software que permita de alguna forma analizar, simular y/o validar un programa concurrente, con especial énfasis en los estilos de problemas abordados en la materia. Esto ayudaría al estudiante en los primeros pasos hacia la comprensión de la programación concurrente: entender dónde y (complementando con conocimientos teóricos) por qué se producen fallos tales como deadlocks, data races, etc.

Dentro del trabajo del proyecto, se realiza un estudio del estado del arte en herramientas educativas, e incluso de uso comercial, para el estudio de la programación concurrente. Las herramientas relevadas son analizadas conforme a un conjunto de requerimientos de usabilidad basados en lo que actualmente se pide para el entendimiento del tema en la UC. Finalmente, se decide trabajar con un subconjunto de ellas. Adicionalmente, se presenta la profundización y adaptación de una de las herramientas exploradas para utilizarse en el curso.

Del conjunto de requerimientos de usabilidad mencionado anteriormente, se destacan:

1. Verificación de existencia de deadlocks.
2. Verificación de existencia de data races.
3. Verificación de existencia de posposición indefinida.
4. Verificación de existencia de livelocks.

[1] Datos obtenidos entre los años 2018-2021

5. Simulación gráfica de procesos concurrentes.
6. Simulación interactiva de procesos concurrentes.
7. Trazabilidad ante la presencia de deadlocks, data races, etc.

A lo largo de la elaboración de este proyecto, fueron surgiendo distintas ideas y posibles implementaciones prácticas, entre las cuales se encuentran:

- Obtener y/o desarrollar una herramienta que permita la **ejecución y verificación** de programas escritos en código por los estudiantes, y que detecte distintos tipos de errores de concurrencia.
- Obtener y/o desarrollar una herramienta interactiva que brinde un **soporte visual** (proceso actualmente en ejecución, contenido de memoria por proceso, etc) al estudiante.
- Obtener y/o desarrollar una herramienta **interactiva** que permita al estudiante avanzar y retroceder en el stack de llamadas para comprender el comportamiento de los procesos concurrentes de un determinado problema.
- Elaborar algún tipo de herramienta que permita la ejecución de problemas ya pre-programados, como pueden ser los problemas clásicos de concurrencia vistos en la asignatura de Sistemas Operativos, con la posibilidad de realizar un análisis de diferentes ejecuciones y dificultades que estos presentan.

1.1. Motivación

La motivación principal de este proyecto es realizar un trabajo que ayude a los estudiantes a comprender la programación concurrente. Se busca brindar un complemento práctico, que no sustituya al conocimiento teórico, sino que promueva la participación del estudiante en elaborar soluciones de diferentes ejercicios.

Hoy en día en la UC se cuenta con la presencia de los profesores que son los encargados de la corrección de las soluciones presentadas y que ayudan a los estudiantes a identificar los patrones de equivocación más comunes. Esto resulta no ser lo más eficiente dada la cantidad de estudiantes que cursan por semestre. Lo que se busca en este proyecto es un complemento a la hora de dar estas correcciones y que el estudiante tenga la posibilidad de interpretar su propia solución con mayor autonomía.

En complemento a la búsqueda de herramientas en artículos académicos, se realiza una investigación informal a través de la web con el objetivo de obtener conocimientos sobre herramientas que no necesariamente se encuentren en el mundo académico, pero sí tal vez en el mundo industrial. En caso de encontrar dichas herramientas, potencialmente podrían ser adaptadas para lograr la parte más práctica del objetivo de este proyecto, que es la de proveer a los estudiantes una herramienta utilizable de forma directa para la verificación de sus propias piezas de código concurrente. Dentro de este marco también se encuentran algunas herramientas puntualmente interesantes, las cuales son presentadas más adelante en este informe.

Cabe destacar que, puede que ante la necesidad de obtener una herramienta que sea efectivamente utilizable, resulte que varias de las herramientas presentadas de corte académico sean rápidamente descartadas, debido justamente a la inexistencia de fuentes de donde obtenerlas y al énfasis en la parte teórica de los métodos utilizados en algunos de ellos. Si bien se podría obtener un marco teórico muy enriquecedor a partir de las distintas técnicas utilizadas en los artículos relevados, resulta imperioso contar con una herramienta que pueda ser probada de forma práctica.

1.2. Estructura del documento

En este primer **capítulo de introducción** se encuentra un resumen general del proyecto y una sección dedicada a motivación del mismo. El marco conceptual es presentado en el **capítulo 2** y, además, pretende dar un repaso general de conceptos teóricos relacionados con la programación concurrente (definiciones básicas, errores más comunes) ya que estos conceptos son utilizados a lo largo del proyecto. También se presentan algunas de las primitivas de sincronización y mensajería utilizadas en la resolución de problemas de este tipo, además de los paradigmas de comunicación entre procesos. Luego, se describen brevemente tres problemas clásicos de la programación concurrente. El enfoque del **capítulo 3** es avanzar en los paradigmas utilizados para testear programas concurrentes. La clasificación se basa en lo presentado en el artículo *Concurrent software testing in practice: A catalog of tools* [19]. Incluye los distintos tipos de testing (funcional, de mutación, basado en modelos, determínistico), así como también la ejecución simbólica, y una clasificación especial para la detección de deadlocks y data races. En el **capítulo 4** se explora un subconjunto de las herramientas presentadas en el artículo mencionado anteriormente, así como también algunas que fueron encontradas a través de la web. De este subconjunto, se realiza especial foco en dos herramientas basadas en model-checking: JPF y SPIN; de las cuales se realiza una comparativa en profundidad, punto a punto, a lo largo de varias características. Dado que, en base al análisis presente en este proyecto y en términos generales, SPIN resulta ser una herramienta con mejores características, gran parte del trabajo se enfoca en dicha herramienta y se dedica el **capítulo 5** a su estudio. En este capítulo se ven en mayor profundidad algunos detalles tales como la interpretación de la salida generada, buenas prácticas (implementación de módulos, escritura eficiente de modelos), así como también las limitaciones que presenta. A su vez, se proveen fragmentos de una serie de soluciones a ejercicios de exámenes de años anteriores de la UC, realizando un análisis de los mismos, así como también algunos casos de error y cómo SPIN los representa a través de su salida. Finalmente, se presenta un breve paneo general de la herramienta gráfica jSPIN, la cual es una interfaz gráfica para SPIN. Las conclusiones y el trabajo a futuro se expresan en el **capítulo 6**, donde se resume lo logrado en el proyecto y se presentan algunas posibles extensiones del trabajo realizado.

Esta página ha sido intencionalmente dejada en blanco.

Capítulo 2

Marco Conceptual

En este capítulo se incluye una breve introducción a los conceptos relacionados con concurrencia, en especial aquellos que son necesarios para comprender el resto del trabajo.

Cabe destacar que a lo largo del informe se hace uso de forma indistinta de los conceptos de programa, proceso e hilo; a menos que sea aclarado de forma específica donde corresponda.

2.1. Programación concurrente

Un programa "ordinario", en el contexto de programación imperativa (paradigma utilizado en la UC), consiste de declaraciones, asignaciones y sentencias de estructuras de control escritas sobre un determinado lenguaje de programación. Hoy en día, incluso tienen la capacidad de ser organizados en procedimientos y módulos de forma tal de organizar el código. Más allá de dichas abstracciones, al final todas estas instrucciones son las que aparecen en el código de máquina que resulta de compilar el programa final. Dichas instrucciones son ejecutadas de forma *secuencial*.

Un **programa concurrente** es un conjunto de programas secuenciales que pueden ser ejecutados de forma paralela.

Tradicionalmente, la palabra **paralelo** se usa para sistemas en los cuales la ejecución de varios programas se solapa en el tiempo, ejecutándolos en procesadores separados. La palabra **concurrente** se usa para el "potencial paralelismo", en el cual las ejecuciones pueden (pero no necesariamente van a) solaparse. Puede incluso que el paralelismo sea solo aparente, y que la implementación de la misma sea a través del uso común de los recursos de unos pocos procesadores (o incluso solo uno de ellos). [3]

Sección crítica y mutua exclusión

El método más sencillo de comunicación entre varios procesos concurrentes es mediante el uso compartido de una variable. Este método, por simple que parezca, puede llevar a errores en el programa, ya que el acceso concurrente puede causar interferencias inesperadas en las acciones de los distintos procesos.

Una posible opción para evitar este tipo de errores es identificar aquellas regiones de los procesos que acceden a variables compartidas y asegurar la condición de acceso en forma independiente, es decir, esas instrucciones (o variables) sean accedidas por la cantidad de procesos (por ejemplo, M) que sea necesario que accedan a la vez, según las características del problema. Se denomina **sección crítica** a la porción de

Tiempo	Proceso 1	Proceso 2
0	l1.lock()	l2.lock()
1	print "l1 obtenido"	-
2	l2.lock()	print "l2 obtenido"
3	print "l2 obtenido"	l1.lock()
4	...	print "l1 obtenido"
5	...	...

Tabla 2.1: Ejemplo Deadlock

código de un programa en la que se accede a un recurso compartido (estructura de datos, dispositivo, etc) que no debe ser accedido por más de M de los N procesos en ejecución [2].

Para alcanzar lo anterior es necesario **mutuo-excluir** las secciones críticas. Para conseguirlo, se deben implementar protocolos de software (en ocasiones con soporte de hardware) que impidan el acceso a una sección crítica mientras está siendo utilizada por los M procesos. En otras palabras, que solo M procesos puedan ejecutar la sección crítica en un momento dado.

2.1.1. Principales problemas en la programación concurrente

Existen una serie de problemas bien conocidos en el contexto de la programación concurrente que se producen ante distintas situaciones. A continuación, se presenta un subconjunto de los mismos, los cuales son estudiados en el libro de referencia de la UC.

Deadlocks

Se dice que un sistema informático se encuentra en estado de **deadlock** si para un conjunto dado de procesos, todo proceso espera por un evento que es provocado por un proceso (incluyendo a sí mismo) de ese mismo conjunto [3]. En general, este evento puede ser el envío de un mensaje, o la liberación de un lock. Un lock es una primitiva de sincronización entre procesos, la cual permite implementar mutuo-exclusión, permitiendo acceso a una determinada sección crítica a una cantidad limitada de procesos (potencialmente uno solo).

Existen diversas formas de afrontar una situación de deadlock, las cuales son: ignorarlo, prevenirlo, evitarlo, o detectarlo y recuperarse. En el marco de este proyecto resulta obvio que ignorarlo no es una buena aproximación al problema, dado que el objetivo principal es que el estudiante tenga la posibilidad de aprender de sus errores. Tampoco es de interés prevenirlo ni evitarlo. Sin embargo, sí es de gran importancia contar con la posibilidad de detectarlo, ya que esa es una buena primera ayuda para lograr dicho aprendizaje (para corregir errores, primero se debe saber que existen, y dónde están).

A modo de ejemplo de un caso en estado de deadlock, en la Tabla 2.1 se puede ver la ejecución de dos procesos a lo largo del tiempo. Se supone que ambos procesos son ejecutados en dos procesadores diferentes, y los locks son de acceso global. Se puede ver que en caso de seguir los ordenes de ejecución planteados en la tabla, el sistema termina en deadlock y nunca finaliza. Lo que termina sucediendo es que el *Proceso 1* se queda esperando por el lock *l2*, el cual ya fue obtenido por el *Proceso 2*. De forma similar, el *Proceso 2* está esperando por el lock *l1*, el cual ya fue obtenido por el *Proceso 1*.

Data races

Se dice que ocurre un data race (condición de carrera) cuando dos o más procesos acceden a la misma dirección de memoria de forma concurrente, y al menos uno de los accesos es de escritura, y además los procesos no están usando ningún lock exclusivo para controlar sus accesos a dicha memoria.

Cuando estas tres condiciones se cumplen, el orden de los accesos es no-determinista, y la computación puede dar distintos resultados entre ejecuciones dependiendo de dicho orden.

A modo de ejemplo, se presenta el Algoritmo 2.1 que contiene un data race. Si ambos métodos se ejecutan en hilos diferentes, se cumplen las condiciones de existencia de data races. Los hilos donde son ejecutados los códigos *f1* y *f2* acceden de forma potencialmente concurrente a la dirección de memoria correspondiente a la variable *n*, el proceso *f1* realiza una escritura sobre la variable *n*, y ningún proceso aplica una estrategia de bloqueo para sincronizar los accesos. En este ejemplo, la salida no es determinista y podría resultar como valor final impreso de la variable *n* tanto *n=1* como *n=2*:

Algoritmo 2.1: Data Race - Programa 1.

```
1  begin
2    global n = 1
3      proc f1() {
4        n++
5      }
6      proc f2() {
7        print("n=", n)
8      }
9  end
```

Esto sucede debido a que, dependiendo de qué proceso se ejecute primero, por un lado podría ejecutarse primero la línea 7 (proceso *f2*), imprimiendo como salida *n=1*. Sin embargo, eventualmente podría ejecutarse la línea 4 (proceso *f1*) en primer lugar, obteniendo *n=2* como salida.

Posposición indefinida

La posposición indefinida se produce cuando a un proceso o hilo se le niega el acceso a un recurso compartido por un período de tiempo no definido. No es sinónimo de deadlock, aunque la existencia de deadlocks implica la posposición indefinida de los procesos involucrados.

Puede, aunque no tiene por qué, acabar. En el caso de los deadlocks, la única forma de salir del bloqueo es mediante una acción externa.

Livelocks

Esta situación es comparable a la que se da en la vida real, cuando dos personas A y B se encuentran enfrentadas en un corredor, y A se mueve a su izquierda para dejar pasar a B, pero B se movió a su derecha, por lo que ninguno puede pasar. Luego, A se mueve a su derecha y B a su izquierda, por lo que nuevamente se bloquean el paso mutuamente, y así de forma indefinida.

Como con los deadlocks, los procesos en estado de livelock no pueden avanzar. Sin embargo, los hilos no son bloqueados, sino que simplemente se encuentran ejecutando sentencias que no son útiles.

2.1.2. Primitivas para la solución de problemas de concurrencia

Para lograr mitigar los problemas anteriormente mencionados, en la asignatura se presentan las siguientes primitivas:

Semáforos

Un semáforo [3] es un tipo de dato inventado por Edsger Dijkstra, donde una instancia de semáforo s es una variable de tipo entero que puede tomar valores positivos diferentes de cero. Una vez s sea inicializado solo dos operaciones son permitidas sobre s: *wait(s)* (*P(s)*) o *signal(s)* (*V(s)*).

Al ejecutar *P(s)*, si $s > 0$, entonces $s = s$ - *1*. Si no, la ejecución del proceso que ejecutó *P(s)* es suspendida.

Por otro lado, al ejecutar *V(s)*, si algún proceso había sido suspendido por una previa ejecución de *P(s)*, entonces ese proceso es despertado. Si no, $s = s + 1$.

La ejecución tanto de *P(s)* como de *V(s)* se realiza de forma atómica. Esto último quiere decir que, dada una operación X, ninguna otra operación Y puede ocurrir entre el comienzo y el final de X. En otras palabras, la ejecución de X nunca es interrumpida.

Monitores

En la búsqueda de tener una herramienta de sincronización más estructurada que un semáforo, se presentan los monitores [3].

Un monitor es una estructura de datos abstracta diseñada para permitir concurrencia pero manteniendo una clase estructurada. Cada monitor tiene su propio conjunto de funciones y de variables (globales dentro del monitor y solo accesibles por las funciones del monitor), además de un constructor donde se inicializa el valor de variables y se deja accesible al monitor a procesos externos. La comunicación de los procesos externos con un monitor es a través del pasaje de parámetros a las funciones internas del mismo y su principal característica es que sus funciones están mutuo excluidas, es decir, en un momento dado solo un proceso como máximo puede estar ejecutando sus funciones. Aquellos procesos que intenten acceder al monitor mientras está ocupado son bloqueados y colocados en una cola de espera para entrar.

La sincronización anteriormente mencionada se logra gracias a las variables de condición (o *conditions*). Para una variable c de tipo condition solo se pueden aplicar dos tipos de comandos:

- *wait(c)*: donde el proceso que lo ejecuta es bloqueado y entra en una estructura FIFO (cola, primero en entrar, primero en salir) asociada al condition c.
- *signal(c)*: donde si la cola de c no está vacía, despierta al primer proceso encolado.

Mailboxes (colas de mensajes)

Otra primitiva utilizada para la comunicación entre procesos son los mailboxes [1]. En este caso el paradigma de comunicación utilizado es el de pasaje de mensajes (definido en profundidad en la Sección 2.2.2).

Esta primitiva tiene dos **métodos fundamentales**, los cuales son *enviar(mensaje)* y *recibir(mensaje)*. Los comportamientos de ambos son autoexplicativos en base a su nombre.

En cuanto a la **implementación de la comunicación**, existen dos variantes. En la *comunicación directa* el emisor debe nombrar explícitamente al receptor en la invocación de los métodos. Por otro lado, en la *comunicación indirecta*, existe una estructura independiente de los procesos llamada mailbox (o cola de mensajes). En

este caso los mensajes son enviados al (y recibidos del) mailbox, además de que pueden existir múltiples emisores y múltiples receptores.

Por otra parte, existen dos **tipos de sincronización** para ambos métodos de comunicación. Tanto el envío como la recepción de mensajes pueden ser bloqueantes o no bloqueantes. Se dice que el método es *bloqueante* si el emisor (receptor) se bloquea hasta que el mensaje es enviado (recibido) al (por el) mailbox, mientras que es *no bloqueante* si el proceso emisor (receptor) continúa su ejecución si el mailbox no está lleno (si hay un mensaje disponible).

Finalmente, el buffer utilizado en la comunicación indirecta puede tener una **capacidad** *finita*, donde puede almacenar N mensajes como máximo. Si se alcanza dicha capacidad máxima, el emisor, si es bloqueante, se bloquea hasta que haya espacio. A su vez, pueden tener una capacidad *infinita*, donde el emisor no se bloquea.

2.2. Paradigmas de comunicación

Se presentan los dos paradigmas principales que se utilizan para la comunicación de procesos, utilizados por las herramientas presentadas más adelante. [18]

2.2.1. Memoria compartida

En el paradigma de **Memoria compartida** existen ciertas direcciones de memorias que son compartidas y pueden ser accedidas simultáneamente por más de un proceso, con el fin de comunicarse entre sí. Es un modelo que se caracteriza por tener una buena velocidad de comunicación, pero al basarse en Memoria compartida se debe sincronizar el acceso y proteger la memoria de forma adecuada (ver Figura 2.1).

Dentro de este paradigma, se encuentra el uso de primitivas tales como los **semáforos** y los **monitores**.

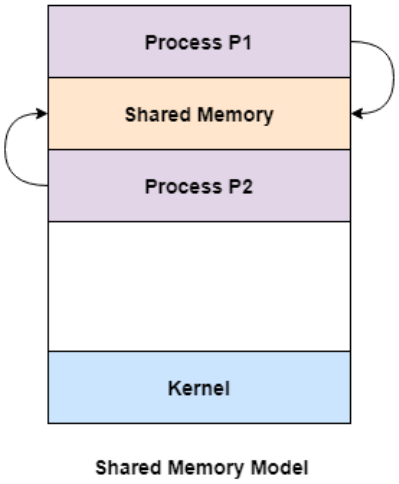

Figura 2.1: Diagrama de Memoria compartida de dos procesos P1 y P2. Extraído de [18]

.

2.2.2. Memoria distribuida

En contextos distribuidos la técnica más difundida para la comunicación es el **Pasaje de mensajes**. Este paradigma se basa en la existencia de una cola de *mensajes* donde múltiples procesos que no tienen un método de comunicación preexistente, pueden acceder a esa cola y escribir o leer mensajes. Dichos mensajes son estructuras de datos que son guardados en la cola hasta que su receptor los obtiene (ver Figura 2.2).

Al no requerir una alta velocidad de comunicación (como puede suceder, dependiendo de su implementación, en el caso del paradigma de Memoria compartida), este modelo tiene la ventaja de necesitar un hardware menos potente. Como contraparte, justamente tiene la desventaja de tener una velocidad de comunicación más lenta que la de Memoria compartida.

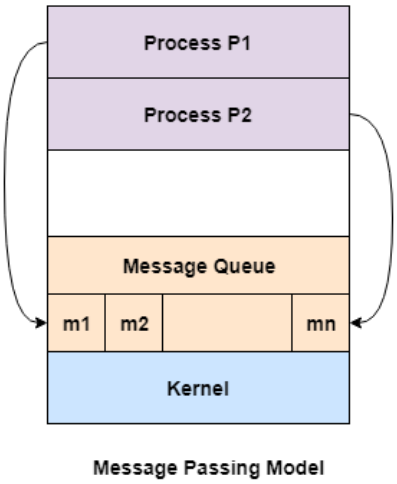

Figura 2.2: Diagrama de Pasaje de mensajes de dos procesos P1 y P2. Extraído de [18]

2.3. Problemas clásicos de concurrencia

En este capítulo se presentan algunos problemas clásicos que se estudian en el contexto de programación concurrente.

2.3.1. Productores-Consumidores

El primer problema presentado es el de Productores-Consumidores, donde existe al menos un proceso Productor y también al menos un Consumidor. Además, existe un buffer en común donde el Productor genera datos que deben ser recibidos y procesados por los Consumidores.

La problemática en este caso es mutuo excluir el buffer, teniendo en cuenta que si no existe un dato almacenado el Consumidor debe esperar a que un Productor almacene uno. En caso de ser un buffer finito, si este está lleno, es el Productor el que debe esperar hasta que el Consumidor libere un espacio.

2.3.2. Lectores-Escritores

Este problema presenta un escenario similar al de Productores-Consumidores, donde existe un conjunto de procesos Lectores que acceden en modo lectura a un recurso común (como una base de datos) y, por otro lado, procesos Escritores que acceden de modo escritura. Los Escritores solo pueden acceder de uno a la vez al recurso compartido, mientras que los Lectores, al ser acceso de solo lectura, pueden hacerlo varios al mismo tiempo.

Para este caso se debe mutuo excluir la base de datos para los procesos Lectores y Escritores, ya que al ser la lectura y la escritura operaciones no atómicas, dependiendo su orden de ejecución pueden llevar a que se produzca un data race.

También puede existir un problema de posposición indefinida. Como múltiples Lectores pueden acceder a la sección crítica, esto puede llevar a que el Escritor nunca pueda acceder. Para este caso se puede hacer una implementación donde se priorice el acceso del escritor a la base de datos.

2.3.3. Filósofos comensales

Aquí se presenta un problema un poco mas complejo, en esta situación hay cinco filósofos que se sientan alrededor de una mesa y pasan su vida cenando y pensando. Cada filósofo tiene un plato y un tenedor a la izquierda de su plato. Para comer los fideos son necesarios dos tenedores y cada filósofo sólo puede tomar los que están a su izquierda y derecha.

Si cualquier filósofo toma un tenedor y el otro tenedor está siendo utilizado, el filósofo se quedará esperando, con el tenedor en la mano, hasta que pueda tomar el otro tenedor, para luego empezar a comer. Si dos filósofos adyacentes intentan tomar el mismo tenedor a la vez, se produce un data race: ambos compiten por tomar el mismo tenedor, y uno de ellos se queda sin comer. Si todos los filósofos toman el tenedor que está a su derecha al mismo tiempo, entonces todos se quedarán esperando eternamente, porque alguno debe liberar el tenedor que les falta, y ninguno lo hará porque todos se encuentran en la misma situación (esperando que otro termine de usar un tenedor). Sucede entonces un deadlock.

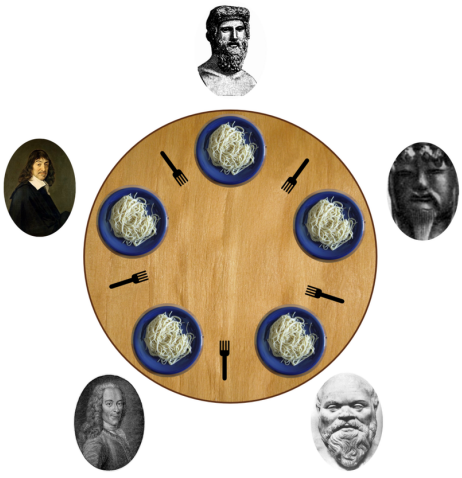

Figura 2.3: Representación gráfica del problema Filósofos comensales

Este problema se puede resolver de diferentes formas utilizando sincronización de

procesos:

- Permitir comer a cuatro filósofos a la vez: esto evita el problema de que cada filósofo quede con un tenedor en su mano, ya que siempre habrá un tenedor libre. Esto puede llevar que el filósofo que no coma quede esperando demasiado tiempo a que se libere un lugar.
- Agregar un proceso externo que se asegure que un filósofo agarra los dos tenedores o, si no puede agarrar los dos, no agarre ninguno.
- Definir un orden a los tenedores y que se tomen en ese orden. Por ejemplo, numerando los tenedores y que solo se puedan tomar en orden ascendente (o descendente).

Capítulo 3

Paradigmas de testing de concurrencia

En este capítulo se profundiza sobre la verificación de los programas concurrentes y los distintos tipos de verificación que existen. Una de las características de los programas concurrentes es que requieren de un costo alto a nivel computacional a la hora de ser testeados, ya que una prueba concurrente ejecuta dos o más subprocesos y el orden también puede cambiar, por lo que una única ejecución no es suficiente. En otras palabras, a medida que la cantidad de procesos crece se vuelve casi imposible cubrir todos los posibles caminos.

Como se menciona anteriormente, gran parte del presente estudio se basa en el trabajo realizado en el artículo de Melo, Silvana M., et al. *Concurrent software testing in practice: A catalog of tools* [19]. Dicho trabajo provee una interesante clasificación de las distintas técnicas que se pueden utilizar para realizar la verificación de programas concurrentes. Las técnicas presentes en dicha clasificación son las siguientes:

- *Testing funcional.*
- *Testing de mutación.*
- *Testing basado en modelos (model-checking).*
- *Detección de deadlocks y data races.*
- *Testing determinístico.*
- *Ejecución simbólica.*

Cada una de las cuales a su vez se puede clasificar como una herramienta cuyo paradigma sea de memoria compartida o de pasaje de mensajes.

Dentro de los lenguajes utilizados por las herramientas se encuentran algunos muy populares como Java, C, C++, así como también algunos desarrollados específicamente para la herramienta en sí, como es el caso del model-checker SPIN y su lenguaje Promela.

A continuación, se describen las técnicas mencionadas y se concluye cuáles son las que más se relacionan con el objetivo del proyecto, realizando posteriormente una exploración en mayor detalle de ellos.

3.1. Testing funcional

El testing funcional se basa en la validación del software probando si cumple con los requerimientos funcionales. Se cuenta con la dificultad extra de contar con la ca-

racterística del no determinismo en los programas concurrentes.

3.2. Testing de mutación

El tipo de testing de mutación [12] es una forma de testing de caja blanca, donde se realizan cambios en el código en lugares previamente elegidos de forma sistemática añadiendo casos de prueba de un conjunto. Dado un programa a ser testeado, el testing de mutación aplica lo que le llaman *operadores de mutación* para encontrar el conjunto de lugares mutables y luego por cada lugar va agregando las pruebas creadas en el código, intentando causar errores en el programa. La idea es que el programa detecte un cambio en el código, por ejemplo eliminar o duplicar una sentencia o cambiar valores de variables. Este proceso en programas concurrentes es bastante costoso ya que lleva tiempo realizar las pruebas cada vez que se mute una sección.

3.3. Detección de deadlocks y data races

Este tipo de herramientas utilizan una variedad de enfoques para lograr esta detección, por ejemplo mediante muestreos estadísticos para detección de data races, la inyección de demoras en la ejecución de algunos threads de forma de forzar la ejecución de ciertas secuencias de sincronización y luego detectar deadlocks, o el monitoreo de la ejecución mediante un planificador de procesos.

Estas herramientas se encuentran dentro de diferentes sub-clasificaciones. Una de estas que resulta interesante de resaltar, es la de análisis estático/análisis dinámico.

Dado que una de las prioridades principales de la parte de concurrencia de la unidad curricular Sistemas Operativos es la de proveer al estudiante con la capacidad de programar de forma concurrente, evitando la presencia de deadlocks y data races, se hace especial énfasis en el estudio de las herramientas presentadas que utilizan esta técnica.

3.3.1. Análisis estático

Este tipo de análisis se realiza de forma estática, en tiempo de compilación, lo que lleva a un análisis más veloz pero no tan efectivo. En general ayuda mucho a la hora de detectar errores como variables que se usan pero no están previamente definidas o a las que le son asignadas variables de diferente tipo. En el contexto de los programas concurrentes las herramientas suelen intentar detectar la ausencia de mutuo-exclusión de una variable o de locks que son tomados pero no devueltos. Uno de los problemas que sufre este tipo de análisis es que puede presentar falsos positivos (potencialmente muchos, dependiendo de la complejidad del código), es decir, casos en los cuales reporta, por ejemplo, deadlocks que en realidad no existen [25].

3.3.2. Análisis dinámico

Por otro lado, existe el análisis en tiempo de ejecución, llamado análisis dinámico. El problema principal que presenta esta forma de análisis, es que la detección del deadlock puede suceder demasiado tarde como para lograr una recuperación. Realizar rollbacks y re-ejecuciones puede ayudar, pero algunos contextos tales como la utilización de entrada/salida pueden complicar la implementación de dichos rollbacks. De cualquier manera, en el contexto del desarrollo de este proyecto, esto no representa un

gran problema. Notar que el objetivo principal es el de la detección de los deadlocks/data races, aunque la posibilidad de realizar un rollback podría ser interesante en el caso de que se quisiera implementar o utilizar una ejecución interactiva [25].

3.4. Testing basado en modelos (model-checking)

Model-checking [14] se basa en la idea de verificar un programa mediante el uso de software que analiza los diferentes estados que un sistema puede tener cuando ejecuta un programa concurrente, intentando aumentar la eficiencia disminuyendo este espacio de estados. El no determinismo de los programas concurrentes conlleva que en su ejecución, para el mismo input, puedan producirse salidas diferentes y, a la vez, correctas. Esto lleva al problema de la existencia de un número enorme de cálculos computacionales, por lo que una comprobación exhaustiva no resulta, a primera vista, práctica como método de testeo de correctitud de un programa.

Se ha demostrado que se puede encarar este problema modelando la ejecución de un programa como un autómata finito no determinista (AFND). Dado el AFND que corresponde a la ejecución correcta del programa y otro que corresponda a la negación de la correctitud del programa, si se puede encontrar una entrada que sea aceptada por ambos autómatas entonces el programa no es correcto y esa entrada es un error.

El AFND no es necesariamente tan grande como podría parecer dado que no todos los estados son alcanzables. Lo más complicado es realizar modelos abstractos, donde se enfoque más en lo que hace el programa que en detalles. Es decir, que tenga los suficientes detalles para que se adapte a la realidad pero reduciendo lo más posible con el fin de que sea más fácil de entender y verificar.

Una de las tareas más difíciles de abordar en el desarrollo de herramientas de model-checking es la de subsanar su mayor desventaja: el crecimiento del espacio de estados. Como se menciona en su articulo Willem Visser, et al. [24]: Dado que el diseño en general contiene menos detalles que la implementación, el uso de model-checking es a menudo pensado como una técnica que es mejor aplicada al diseño que a la implementación. El articulo también destaca que puede ser conveniente utilizar model-checking junto con alguna otra técnica de reducción de estados ya que estiman que aplicarlo por sí solo a programas con un número mucho mayor que 10000 líneas, no es escalable.

3.5. Ejecución simbólica

La idea de verificación de programas basada en ejecución simbólica [9] reside en ejecutar el programa objetivo de forma simbólica, es decir, los valores de entrada y las variables se representan como valores simbólicos en lugar de valores concretos. Estos valores se usan para generar condiciones de ruta, que son fórmulas lógicas que representan el estado del programa y las transformaciones entre los mismos. Por ejemplo, para la siguiente pieza de código:

```
1  input: int entrada
2  output: -
3  begin
4   entrada = read();
5   z = entrada * 2;
6   if (z == 12) {
7     fail();
8   } else {
9     printf("OK");
10  }
11 end
```

La aplicación de la ejecución simbólica utiliza un valor simbólico (λ) y asignándole a la variable *entrada* ese valor, generando dos caminos simbólicos (si λ*2 es igual a 12 o si no lo es), logrando así deducir qué entradas llevan por qué camino. La ejecución simbólica mantiene un estado simbólico, que mapea variables a expresiones simbólicas y a restricciones de ruta simbólica. Cada ruta simbólica y su restricción (por ejemplo la ruta que lleva a que se cumpla la sentencia "If" y la que no) pueden ser tomadas. Cuando se termina una ejecución simbólica sobre una ruta, ya sea con un resultado exitoso o con error, el resultado es guardado y si el programa es ejecutado con un valor concreto que cumpla con alguna de las rutas, el resultado ya será conocido.

La ejecución simbólica puede ser útil en los programas concurrentes, ya que ayuda a modelar los estados que estos programas pueden tener al momento de su ejecución, mejorando la eficiencia de la verificación.

Esta técnica puede tener algunos problemas como la generación de caminos que no tengan una solución y, por ende, sean infinitos. En la práctica se define un número finito de caminos a explorar o la profundidad de exploración.

3.6. Conclusión

Entre los tipos de enfoques estudiados para la detección de problemas de concurrencia, se concluye que la verificación basada en modelos (model-checking) resulta ser el más eficaz y con una utilización de fundamentos teóricos que respaldan dicha eficacia. Se considera mejor a otras herramientas, como las de testing más tradicional o las de detecciones estática y dinámica sin model-checking, que tienden a fallar al momento de detectar la totalidad de los errores o reportan falsos positivos. Aún teniendo en cuenta el inconveniente que presenta model-checking, que es que puede llegar a presentar un gran consumo de recursos debido al aumento del espacio de estado de manera rápida al aumentar la cantidad de líneas de código a analizar de un programa.

Por otro lado, si bien puede llegar a ser computacionalmente costoso, model-checking se adapta muy bien al objetivo de aprendizaje del curso, ya que los ejemplos que se presentan (Productores-Consumidores, Lectores-Escritores, etc) y los ejercicios de parciales o exámenes son de una solución pequeña con relativamente pocos posibles caminos de ejecución diferentes. Una herramienta que lleva esta técnica a la práctica es SPIN, presentada más adelante.

Adicionalmente, para el marco de este trabajo, las herramientas en las que se hará mayor énfasis son las que logren identificar la existencia de deadlocks y/o data races, ya que son los que obtienen mayor atención en el contexto del actual curso de Sistemas Operativos. Idealmente, encontrar un complemento a una herramienta que base su análisis en model-checking, que permita generar una simulación gráfica o interactiva de procesos concurrentes, permitiría a los estudiantes obtener una visión más profunda del comportamiento de los programas concurrentes. Finalmente, algo que resulta de vital importancia es la utilización de alguna herramienta que permita la trazabilidad ante un problema de sincronización, ya que es el método principal mediante el cual el estudiante podría dar uno de los primeros pasos hacia la comprensión de la programación concurrente: entender **dónde** y (complementando con conocimientos teóricos) **por qué** se producen fallos tales como deadlocks, data races, etc.

Capítulo 4

Herramientas exploradas

En este capítulo se presentan algunas herramientas que implementan uno o más tipos de testing presentados en el Capítulo 3.

El resultado siguiente se obtiene de la base que presenta el esfuerzo de Silvana M. Melo et al. [19], el cual, como se mencionó anteriormente, es un compilado/catálogo de 116 herramientas distintas utilizadas para la verificación de programas concurrentes. El relevamiento cubre el período que comprende desde el año 1992 al 2014. Dentro del mismo, se realiza una clasificación según el tipo de herramienta (de testing estructural, de model-checking, etc.), la cual provee una categorización bastante útil al momento de seleccionarlas y filtrarlas para su análisis, y es la forma de clasificación principal que se utiliza a lo largo de este proyecto. Por otro lado, cabe destacar que las herramientas presentadas a continuación, son un pequeño subconjunto de todas las contenidas en el texto original. Una gran cantidad de las mismas son descartadas debido a que son de código propietario, pagas, o directamente el código no es accesible por algún otro motivo.

4.1. OSHAJava

OSHAJava [26] implementa una estrategia de **testing funcional** que utiliza análisis dinámico para probar la especificación de programas concurrentes escritos en Java. Su metodología es indicar cuáles métodos de un determinado programa pueden comunicarse a través de memoria compartida cuando son ejecutados en hilos separados. En tiempo de ejecución se verifica que la especificación sea conformada, devolviendo una excepción en caso de que no lo sea.

Una aspecto positivo de OSHAJava es que facilita el control de los programadores sobre la comunicación entre hilos, la cual suele producirse de forma implícita. Sin embargo, se decide no profundizar en esta herramienta ya que, además de contar con cierto nivel de dificultad de aprendizaje para su uso, no se basa en la detección de los principales problemas que se buscan cubrir en este trabajo como lo son la detección de posposición indefinida, deadlocks, etc.

4.2. MutMut

MutMut [11] es una herramienta de **testing de mutación**. Utiliza una técnica cuya idea base es elegir ciertas partes del código como candidatas a ser mutadas, y

realizar varias ejecuciones donde esas partes del código son cambiadas con cierta probabilidad. Para dicha selección se utilizan operadores de mutación, los cuales definen los criterios a utilizar para cambiar el código. Dentro de estos se encuentran algunos como la *Modificación de Parámetros de Métodos de Concurrencia*, donde un ejemplo de uso sería cambiar el valor inicial de un semáforo presente en el programa; así como también otros tales como el *Intercambio de Objetos de Concurrencia* donde, por ejemplo, se reemplazan primitivas de sincronización por otras (semáforos por locks, etc). El enfoque también permite al tester seleccionar un hilo específico para ejecutar, obligando a la mutación introducida a ser ejecutada.

Esta herramienta si bien se presenta como interesante a nivel teórico, su aplicación práctica es compleja. No funciona para la detección de los problemas de deadlocks y además, basado en su documentación, crece rápidamente el orden computacional necesario mientras más threads se utilizan. Esto lleva a descartar el uso de esta herramienta.

4.3. Carisma

Carisma [27] es una herramienta que se basa en el muestreo de múltiples ejecuciones del mismo código. El fuerte de esta herramienta en comparación a otras de su misma clase, está en necesitar un tamaño relativamente pequeño de elementos de muestreo, y aún así mantener un buen nivel de precisión de detección de data races.

Explota el principio de localidad, el cual sugiere que un programa puede acceder múltiples veces a una misma posición de memoria, con motivos similares en cada acceso. Luego, utilizando técnicas estadísticas, logra obtener estimados de cuales van a ser los accesos a memoria en ejecuciones siguientes, asistiendo de esta forma a la herramienta para **detectar data races**.

Si bien Carisma arroja buenos resultados en la detección de data races y además utiliza C, el cual es un lenguaje conocido, no puede ser utilizado en la detección de deadlocks o posposición indefinida.

4.4. LCT

LCT [17] es una herramienta que utiliza **concolic testing**, la cual es una ténica resultado de la combinación de ejecuciones concretas (*"conc-"*) y simbólicas (*"-olic"*). La parte de la ejecución "concreta" refiere a la ejecución normal del programa. En las ejecuciones "simbólicas" se recolectan las restricciones simbólicas que se deben utilizar sobre los valores de entrada simbólicos.

Por las características de los ejercicios que se plantean en la asignatura de Sistemas Operativos, LCT y la ejecución simbólica no cumplen con las características buscadas al momento de validar dichos ejercicios, ya que no se busca identificar las diferentes soluciones que pueda dar un programa concurrente sino que el programa no caiga en los principales problemas que puede tener y además el estudiante entienda la comunicación entre procesos.

4.5. RacerD

RacerD [21] [6] es una herramienta open source desarrollada por Facebook cuyo objetivo es la **detección de condiciones de data race** en programas escritos principalmente en Java, pero también con algo de soporte para C/C++, ObjectiveC y

C#/.Net, utilizando **análisis estático**. Utiliza razonamiento simbólico formal para cubrir varios caminos a lo largo del programa sin ejecutarlo.

El análisis que hace esta herramienta no pretende ser exhaustivo y probar la ausencia absoluta de errores de concurrencia, sino que se busca una clase bastante común de data races. Su enfoque es el de funcionar correctamente en ambientes de producción. Tiene los objetivos de ser performante y detectar los errores rápidamente, ser escalable, y presentar baja fricción al momento de realizar la configuración de la herramienta.

Estas últimas, si bien son un conjunto de características deseables en toda herramienta de software, como ya se mencionó anteriormente, no necesariamente resultan vitales en el contexto del desarrollo de este proyecto.

Como contrapartida a estas características, sucede que la misma no es exhaustiva, es decir, no tiene la capacidad de detectar la totalidad de los errores producidos. Esto sucede, entre otras cosas, debido a que buscando evitar uno de los principales problemas de las herramientas de este tipo (exceso de falsos positivos), RacerD prioriza la reducción de los falsos positivos por sobre la detección de los verdaderos negativos. En otras palabras, intenta que hayan menos alertas falsas de data races, aunque eso conlleve que algunos data races no sean detectados.

Por todo lo mencionado anteriormente es que RacerD resulta una herramienta interesante a ser aplicada en el proyecto pero que necesitaría ser complementada por otras, tanto con la capacidad de detección de deadlocks, como de reducción de falsos positivos.

4.6. Clang Thread Safety Analysis

Clang Thread Safety Analysis [23], es una extensión de C++ que permite **detectar y alertar sobre potenciales data races** en el código, el análisis realizado es de forma **estática** (en tiempo de compilación). Es desarrollado por Google y CERT/CEI y es utilizado activamente en código interno de Google.

Se basa en la declaración de tipos, donde el programador puede declarar cómo se accede a los datos en un entorno de múltiples hilos de ejecución. Por ejemplo, si la variable *var* se etiqueta como *guarded* por un mutex *mu*, al momento del análisis estático la herramienta arrojará un mensaje de aviso si en alguna parte del código se escribe o lee en var sin un previo bloqueo de *mu*.

Se decide no seguir con las pruebas de esta herramienta ya que, al igual que racerD, es sólo de análisis estático y puede pasarse por alto algunos problemas que son de interés hallar como deadlocks o posposición indefinida.

4.7. ThreadSanitizer

Otra de las herramientas exploradas es ThreadSanitizer [22], su objetivo es la **detección de data races** utilizando **análisis dinámico**. Es desarrollada por Google para la validación de muchos de sus algoritmos.

ThreadSanitizer observa la ejecución del programa como una secuencia de eventos. Los eventos más importantes son los eventos de acceso a la memoria (Read y Write) y eventos de sincronización (como Waits y Signals).

Es una herramienta muy eficaz pero difícil de utilizar. Además, solo detecta data races, y no a los demás problemas de concurrencia como deadlocks o posposición indefinida.

4.8. SPIN

SPIN [13] es un sistema de verificación basado en **model-checking** que tiene como objetivo analizar la correctitud de programas concurrentes con respecto a ciertas propiedades declaradas y soporta tanto el paradigma de **pasaje de mensajes** como el de **memoria compartida**. Los programas concurrentes deben ser escritos en el lenguaje Promela, este es un lenguaje de programación inspirado en C. Los procesos se comunican a través de variables compartidas, o mediante el pasaje de mensajes a través de canales (channels) nativos de Promela. Las propiedades a ser verificadas se escriben en lógica temporal lineal (LTL), con la cual se introducen conceptos de tiempo a la lógica tradicional, tales como "una cierta condición *eventualmente* va a ser cierta", o "una cierta condición va a ser falsa *hasta* que otra condición se vuelva verdadera". SPIN utiliza la técnica de model-checking para verificar la correctitud, lanzando un mensaje de error en caso de no cumplir alguna.

SPIN también proporciona la posibilidad de ejecutar una simulación interactiva, con el que los programas de Promela se pueden ejecutar paso a paso. Esta se puede utilizar en particular para volver a ejecutar los rastreos de errores generados por el model-checker sobre las propiedades no satisfechas.

Promela provee de forma nativa la implementación de semáforos no binarios, además de los ya mencionados channels, donde se logra implementar fácilmente la primitiva mailbox. Los monitores deben ser implementados utilizando alguna de las dos herramientas anteriores.

A efectos de lo planteado en este proyecto, SPIN provee grandes fortalezas a la hora de analizar una solución planteada para problemas de concurrencia. Se destaca por su rapidez y efectividad en encontrar **deadlocks** o **data races** utilizando model-checking y análisis dinámico. De forma práctica, dado el código de una solución a un problema de concurrencia, se debe ejecutar el analizador que genera una salida donde se marcan, en caso de existir, los errores.

Las mayores desventajas de esta herramienta se encuentran a la hora de aprender a utilizarla, donde por ejemplo, la salida generada al momento que se producen errores, es poco clara. Además, se agrega la dificultad de que utiliza su propio lenguaje de programación (Promela) que es poco práctico de aprender durante el curso, añadiendo así una dificultad extra para la comprensión del tema de concurrencia.

Estas desventajas pueden ser disipadas si se logra generar un lenguaje y una herramienta que logren que dicho lenguaje sea transpilable a Promela, de forma tal de intentar obtener una forma más amigable de escribir el código.

4.9. JavaPathFinder (JPF)

JPF [24] una herramienta desarrollada por la NASA, utilizada para la detección de **varios tipos de errores de concurrencia**, la cual se basa en un sistema de model-checking y en el paradigma de **memoria compartida**.

La primera versión de JPF se desarrolló en 1999 [7] y originalmente realizaba una traducción de Java a Promela para que este último se encargara de la ejecución del model-checking. Sin embargo, esta traducción sufría de algunos problemas, por lo cual se desarrolló una nueva versión con la capacidad de ejecutar la totalidad de las instrucciones a nivel de bytecode del programa a ser verificado sobre la propia Java Virtual Machine. De esta forma, resulta interesante destacar que JPF es un model-checker de estado explícito, al igual que SPIN. Es decir, tanto la descripción del estado, como las transiciones entre los mismos, es mantenido de forma explícita [15].

Una de las características que resulta interesante mencionar de JPF es que su diseño es modular y fácil de comprender. Notar que SPIN es al menos un orden de

magnitud más rápido que JPF [24].

Sus desarrolladores también justifican que la herramienta está escrita en Java (y no en C, lo cual brindaría un mayor grado de performance) porque les permitió una velocidad de desarrollo ampliamente superior.

Para afrontar la mayor desventaja de model-checking (explosión de espacio de estados) resulta deseable complementar las herramientas de model-checking con técnicas de reducción del espacio de estados. A mayor cantidad de líneas en el programa, mayor es la cantidad de estados a evaluar, y este crecimiento se da de forma exponencial. JPF es una herramienta que evalúa código que no está estructurado en forma de modelos, por lo que naturalmente sufre de este problema aún en mayor medida que otras herramientas tales como SPIN. JPF hace uso de técnicas para reducir los problemas que acarrea la explosión de estados. A continuación, se describen algunas de dichas técnicas [8].

La primer técnica es la *reducción de simetría*, que utiliza clases de equivalencia para descartar estados que se asemejan lo suficiente a otros que ya fueron visitados.

Por otro lado, utiliza tres técnicas de análisis estático: *slicing estático* y *evaluación parcial*, donde ambos generan un programa más pequeño, funcionalmente equivalente al original, y *computación de orden parcial*, que identifica sentencias que pueden ser ejecutadas de forma concurrente con cualquier otra sentencia sin generar errores.

Finalmente, haciendo uso de *análisis dinámico*, extrae un conjunto de propiedades de caminos que ya fueron ejecutados para evaluar elementos en estados de caminos que aún no fueron visitados. Incluso si no se encontraron errores en el camino ejecutado, la información extraída puede proveer contexto para encontrar potenciales errores en otros caminos.

Una de las características más destacables de esta herramienta es que realiza el model-checking sobre código escrito en Java. Esto tiene como consecuencia la existencia de un conjunto de ventajas de esta herramienta por sobre las demás. En primer lugar, Java es un lenguaje conocido por los estudiantes, lo cual implica una menor curva de aprendizaje y una mayor facilidad para plasmar las implementaciones en código. En segundo lugar, a diferencia de Promela, es un lenguaje de uso común y no está orientado a modelos. Si se pretendiera utilizar las herramientas de forma directa, teniendo a los estudiantes programando directamente sobre las mismas, este cambio de paradigma hacia modelos implicaría una mayor carga de aprendizaje para el estudiante. Esto último también se cumple si se desea desarrollar una herramienta intermediaria que traduzca de otro lenguaje a JPF.

Por supuesto, también cabe destacar que utiliza model-checking, lo cual como se menciona anteriormente es una característica deseable para el desarrollo de este proyecto.

4.10. Comparativa

En los apartados anteriores ya se intenta resumir las fortalezas y debilidades de cada herramienta evaluada, en esta sección se busca profundizar en la comparativa. Se presenta un cuadro de comparación entre las herramientas detalladas anteriormente, con el objetivo de apoyar la decisión de qué herramienta o herramientas se adaptan mejor a la solución buscada.

La comparación se realiza en base a los puntos detallados con anterioridad sobre el foco del proyecto, en particular la detección de problemas como deadlocks, Posposición indefinida, data races o livelocks, la facilidad para el estudiante en la implementación de la solución y lectura e interpretación de la salida, tipo de testing exhaustivo y por último si la herramienta cuenta con algún agregado útil como representación gráfica o simulación interactiva.

No se incluyen en el cuadro comparativo las herramientas: OSHAJava, MutMut, Carisma y LCT, descartadas por no cumplir con características determinadas como obligatorias.

Características	**RacerD**	**TSA**	**ThreadSan**	**SPIN**	**JPF**
Detección de Deadlocks	NO	NO	NO	SI	SI
Detección de Data Races	SI	SI	SI	SI	SI
Detección de P. indef	NO	NO	NO	SI	SI
Detección de Livelocks	NO	NO	NO	SI	SI
Legibilidad de la Salida	SI	SI	SI	Verborrágico	Verborrágico
Facilidad Implem.	SI	SI	SI	NO	SI
Rep. Gráfica	NO	NO	NO	SI	SI
Simulación Interact.	NO	NO	NO	SI	SI
Testing exhaustivo	NO	NO	NO	SI	SI

En base a la anterior comparativa, se seleccionan SPIN y JPF en forma primaria para profundizar más en el estudio. En la siguiente sección, se presenta un análisis en mayor detalle de las diferencias entre dichas herramientas, realizando un estudio en términos de los parámetros vistos en esta sección. En este sentido, y como complemento, también se comparan las herramientas al momento de implementar las primitivas del curso como Semáforos, Monitores y Mailbox.

4.10.1. Comparación de primitivas

Semáforos

Promela, lenguaje de los programas que ejecuta SPIN, cuenta con su propia directiva para ejecutar instrucciones atómicas (atomic). Esta se puede utilizar para implementar las funciones P y V.

Por otro lado, JPF tiene como primitiva el tipo *Semaphore* con su correspondiente constructor (*init*), y sus métodos *acquire* (P) y *release* (V).

Monitores

Dado que SPIN no cuenta con una implementación nativa para monitores, se debe construir dicha primitiva de forma manual. Es decir, se debe implementar a base de funciones propias del lenguaje Promela.

Por el lado de JPF es más sencillo, ya que Java cuenta con librerías para la implementación de elementos de concurrencia [20], tales como *Locks* y *Conditions*, a partir de las cuales se puede implementar fácilmente una estructura de Monitores del estilo de los que se estudian en el curso. A su vez, Java cuenta con una palabra reservada llamada *synchronized*, la cual, al ser agregada al método de una clase hace que dicho método tenga el mismo comportamiento que una función de un monitor.

Mailboxes

SPIN implementa su propia cola de mensajes de comunicación indirecta, llamada *channels*. Es una estructura de cola de elementos que se utiliza para poder enviar mensajes de un proceso a otro. Son de recepción bloqueante y cuentan con una sentencia de envío de mensaje y otra para la recepción.
A modo de ejemplo se muestra una definición de un variable de tipo channel y la sintaxis de envío y recepción de mensaje:

- *chan buffer = [5] of { byte };* Se define un channel *buffer* de tamaño *5*, donde en cada lugar se almacena un byte.
- *buffer ! 2*; Esta sentencia es utilizada para envíar el valor *2* al *buffer*, que es insertado al final.
- *buffer ? msg*; Con esta sentencia se recibe el mensaje de *buffer* quitado del comienzo de la cola y se almacena en la variable *msg* para poder ser utilizada.

En cuanto a JPF, se puede utilizar la clase *LinkedBlockingQueue*, la cual implementa una cola de mensajes.

4.10.2. Legibilidad de la salida

Con respecto a la **Legibilidad de la salida**, en términos generales se puede decir que las salidas de ambas herramientas son bastante similares, pero que presentan algunas diferencias claves que son detalladas a continuación.

Con respecto a la indicación de **cuál fue el error**, JPF muestra cuál es la propiedad que es violada. En el caso de la presencia de deadlocks, muestra que la excepción que se dispara es *gov.nasa.jpf.vm.NotDeadlockedProperty*, o en el caso de data races muestra *gov.nasa.jpf.listener.PreciseRaceDetector*. En cambio, SPIN muestra solo el número de errores encontrados, en caso de ser mayor que 0 no se indica con exactitud cuál es el error encontrado, sino que se debe ejecutar y corroborar a través del stack trace generado. Como complemento a esto, es posible agregar funciones de *assert* en el código original donde, si alguna condición en particular no se cumple, el programa terminará y en la salida obtenida se marca qué condición del assert no se cumplió. Esto, por ejemplo, es útil a la hora de detectar la violación de cantidad límite de procesos adentro de una cierta zona critica.

Con respecto al **stack trace**, JPF presenta un mayor nivel de detalle, llegando a mostrar la sucesión de funciones de Java que fueron invocadas en el proceso. Esto puede resultar de utilidad si se implementan librerías que abstraigan partes del trabajo hecho por los estudiantes, por ejemplo, si se implementa alguna librería que brinde una API más similar a la que se usa para monitores en el curso que las opciones nativas de Java. También detalla qué línea fue la última ejecutada en cada paso del thread, incluyendo el último paso, lo cual permitiría al estudiante saber en qué línea quedó cada thread cuando se produce un deadlock/data race. A modo de aclaración, para el caso de SPIN se da que, para poder ver detalladamente el stack trace de la ejecución, se debe utilizar un segundo comando, además de la ejecución propia del modelo, *spin -t [NOMBRE_DE_ARCHIVO].pml*. Este segundo comando es el que muestra en mayor nivel de detalle el conjunto de pasos mediante el cual se llegó al error.

Por otro lado, SPIN presenta una utilidad interesante, que es la de mostrar los **valores** de cada una de las **variables** al finalizar la ejecución. También se encarga de mostrar en cada paso del stack trace, cuál es el nuevo valor de la o las variables afectadas, si es que las hay. Esta funcionalidad no está presente en JPF.

También cabe destacar que SPIN, al no tener las primitivas implementadas de forma nativa, termina **mostrando en su salida las líneas internas** de los métodos implementados a mano. Es decir, si se implementa por ejemplo la primitiva P de los semáforos, la salida va a mostrar las líneas correspondientes a su contenido ($S > 0$, $S = S - 1$).

En cuanto a las similitudes, ambas herramientas tienen una **sección de estadísticas** que muestra detalles tales como la cantidad de estados recorridos, la cantidad de estados por los cuales se pasó mediante el uso de backtracking, la profundidad máxima alcanzada en el espacio de estados, la cantidad de memoria máxima utilizada, etc. Dichas estadísticas, en principio, no resultan muy relevantes para esta comparación,

ya que no son de gran utilidad para el uso de los estudiantes, debido a que no proveen información útil para los propósitos del curso.

4.10.3. Facilidad de implementación

Una de las diferencias más importantes entre ambas herramientas es la **facilidad de implementación**, vista en términos de la dificultad que le puede representar a un estudiante escribir una solución a un problema. Java es un **lenguaje familiar**, que los estudiantes utilizan en varias unidades curriculares a lo largo de la carrera, es imperativo y orientado a objetos. Es decir, tiene las características de un lenguaje que naturalmente le resultaría familiar a un estudiante.

Promela, por otra parte, es un lenguaje 100 % orientado a modelos. Introduce un **nuevo paradigma**, cuyo proceso de adaptación puede significar una **curva de aprendizaje demasiado alta** para los estudiantes. Presenta diferencias en el uso de elementos básicos de programación, tales como las estructuras de control básicas (*if*, *while*), mostrando formas de uso "fuera de lo común"no solo en lo sintáctico, sino también en lo semántico. Por ejemplo, la sentencia *if* puede evaluar múltiples guardas de forma simultánea.

Algoritmo 4.1: If en Promela

```
1  if
2  :: (a > 3) -> ...
3  :: (a == 4) -> ...
4  fi
```

En el algoritmo 4.1, se presenta un ejemplo de utilización de *ifs* en Promela. Las sentencias entre paréntesis ubicadas luego de los dos doble-puntos se llaman *guardas*. El contenido de cada guarda (el código representado por los tres puntos consecutivos), es **potencialmente** ejecutado solo cuando la guarda resulta verdadera. Todas las guardas se evalúan de forma simultánea, y si hay más de una guarda verdadera, se escoge una de forma no determinista. Si ninguna guarda es verdadera, se bloquea el proceso hasta que alguna sí lo sea.

Habiendo dicho todo esto, también se podría interpretar este **nuevo paradigma como una ventaja**, ya que encasilla al programador (al estudiante en este caso) dentro de un cierto marco que orienta, en menor o mayor grado, a que el programa final sea pensado como modelo, es decir, como lo que finalmente va a ser evaluado. La implementación de soluciones en JPF puede terminar dando demasiada libertad al momento de utilizar elementos que no necesariamente son pensados para ser utilizados en el paradigma de modelos, que pueden traer como consecuencia desventajas, tales como la ineficiencia del código final (explosión del espacio de estados).

4.10.4. Representación gráfica

En lo que tiene que ver con la **representación gráfica**, es decir, la capacidad de las herramientas de generar gráficos que ayuden en la comprensión de los errores, tanto SPIN como JPF poseen herramientas gráficas externas.

Por el lado de SPIN existe jSPIN [5]. Si bien la característica más interesante es que tiene la capacidad de generar grafos que representan el espacio de estados del programa, tiene la desventaja de que para un programa medianamente extenso, estos grafos se vuelven muy complicados de interpretar y obtener un verdadero provecho. A modo de ejemplo, se puede observar la Figura 4.1, donde se representa el diagrama de estados para el problema de Productores-Consumidores implementado con semáforos.

En la sección 5.6 se provee un detalle más profundo de las posibilidades provistas por esta herramienta auxiliar.

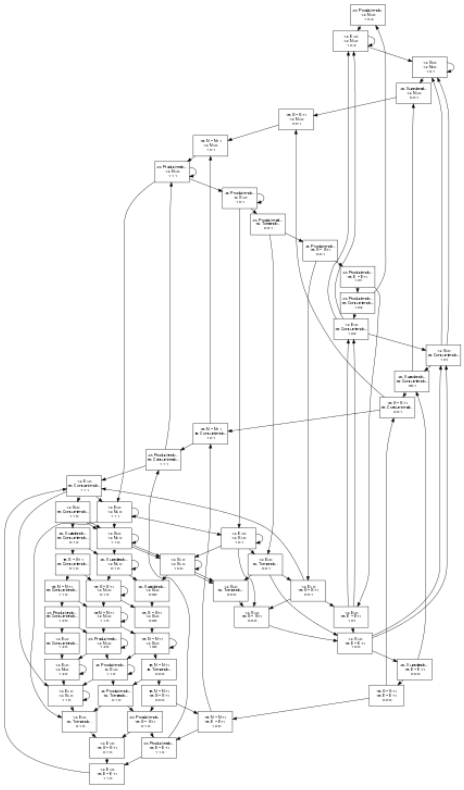

Figura 4.1: Grafo de estados para el problema de Productores-Consumidores con semáforos

Por el lado de JPF, existe una extensión llamada *jpf-visual* cuya interfaz gráfica se puede visualizar en la Figura 4.2.

La principal utilidad que se le podría dar a este tipo de representaciones gráficas, sería en programas pequeños, con el fin de brindarle al estudiante una **visión global** del proceso de ejecución de los distintos estados por los que puede pasar un programa. Con programas de mayor porte, el grafo puede tornarse inútil debido a la imposibilidad de comprenderlo dado su tamaño.

4.10.5. Eficiencia

En este punto se realiza una comparación de **eficiencia** utilizando los tiempos de ejecución de las herramientas. Los resultados se muestran en las siguientes gráficas, donde se aprecia una marcada tendencia a favor de SPIN. El crecimiento en términos de tiempos de ejecución con respecto a cantidad de threads para JPF tiene un comportamiento exponencial, incluso para problemas relativamente pequeños (Productores-Consumidores, Lectores-Escritores, etc).

A continuación, se presentan las gráficas comparativas para Lectores-Escritores, Productores-Consumidores y Filósofos comensales. Los resultados que se muestran de los valores de tiempos de ejecución son obtenidos realizando el promedio de 5 ejecuciones. Durante la experimentación con los distintos problemas se pudo ver que JPF tiene una marcada tendencia a terminar el programa por falta de memoria, incluso luego de aumentar la memoria máxima utilizable por Java por defecto, de 1GB a 6GB.

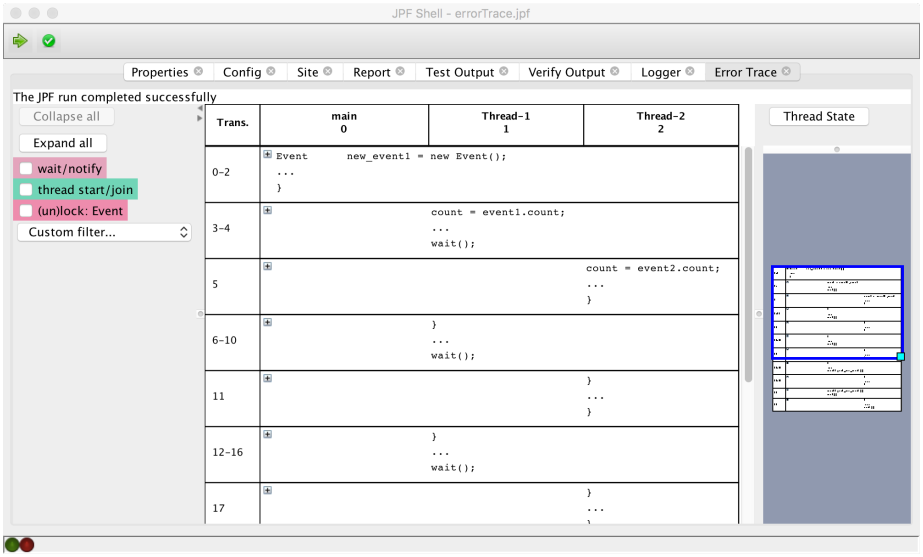

Figura 4.2: Interfaz gráfica de la extensión *jpf-visual*

Durante el proceso de experimentación, independientemente del problema, se pudo ver que ambas herramientas tienen una limitación a causa de la gran cantidad de memoria utilizada. Por un lado, JPF (que es la que sufre en mayor medida de este problema) requirió que se aumentara la memoria máxima utilizable por Java por defecto, de 1GB a 6GB utilizando la flag `-Xmx6144m` y, por parte de SPIN, es necesario un aumento significativo en la profundidad máxima que puede explorar en el autómata generado para en model checking, utilizando la flag `-m10000000`.

Es por esto último que los parámetros explorados son limitados a valores pequeños de cantidades de threads (por ejemplo, hasta 2 productores y 3 consumidores para el caso de semáforos).

Productores-Consumidores

Para este problema, se tienen tres parámetros:

- Cantidad de Productores.
- Cantidad de Consumidores.
- Tamaño del buffer.

Se decide fijar el parámetro del tamaño de buffer con un valor igual a 2 por dos motivos. Por un lado, utilizar tal tamaño resulta equivalente a usar uno de tamaño 4, a efectos de modelar el problema, dado que el tamaño del buffer no cambia la clase del problema. Por otra parte, durante la experimentación se pudo comprobar que la variación en dicho tamaño no afecta tanto a los tiempos de ejecución como lo hace la cantidad de Productores o Consumidores (es decir, la cantidad de threads).

Entonces, se tiene que los parámetros a variar son la cantidad de Productores y la cantidad de Consumidores. A efectos de poder mostrar las variaciones en 2 dimensiones, se decide utilizar varias gráficas por separado, fijando el parámetro $\#Productores$ en cada una de ellas.

Como se puede ver en las figuras 4.3, 4.4 y 4.5, y tal como se mencionó anteriormente, JPF tiene una degradación de performance mucho mayor que SPIN. Los valores

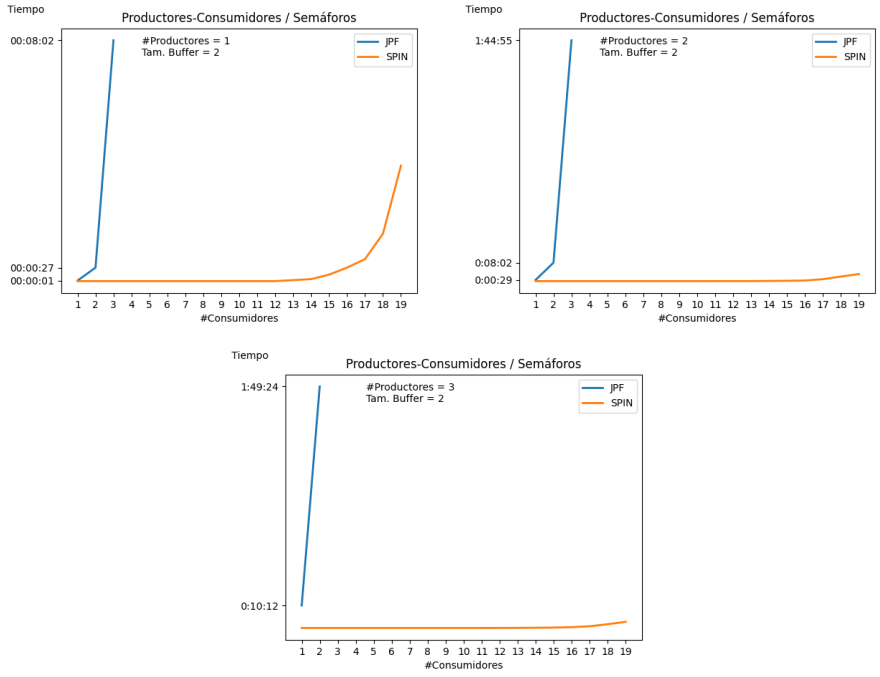

Figura 4.3: Semáforos - Tiempos de ejecución vs $\#Consumidores$

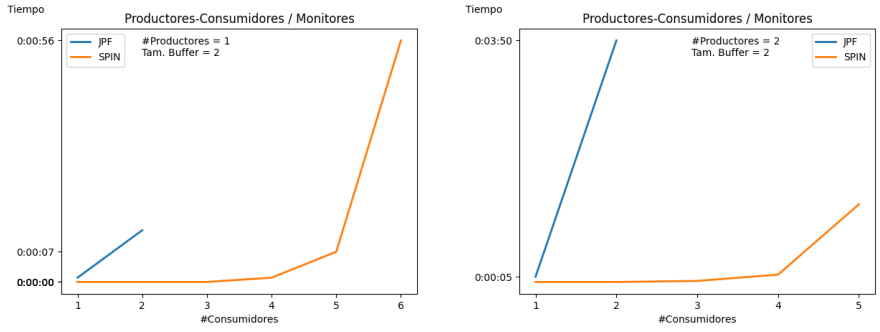

Figura 4.4: Monitores - Tiempos de ejecución vs $\#Consumidores$

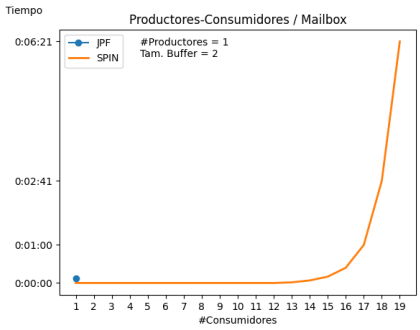

Figura 4.5: Mailbox - Tiempos de ejecución vs $\#Consumidores$

alcanzados por SPIN están muy cerca de los 0 segundos, mientras que JPF tiene un comportamiento exponencial en términos de tiempo de ejecución. También cabe destacar que no se lograron terminar las ejecuciones con JPF debido a falta de memoria para combinaciones de parámetros con valores mayores o iguales a los siguientes:

- Semáforos, $\#Productores = 3$ y $\#Consumidores = 3$
- Monitores, $\#Productores = 3$ y $\#Consumidores = 2$, $\#Productores = 2$ y $\#Consumidores = 3$
- Mailbox, $\#Productores = 2$ y $\#Consumidores = 1$, $\#Productores = 1$ y $\#Consumidores = 2$

Lectores-Escritores

Para este problema, se tienen dos parámetros:

- Cantidad de Lectores.
- Cantidad de Escritores.

Para este problema sucede lo mismo que en Productores-Consumidores. Al tener dos parámetros, se decide fijar $\#Escritores$ en cada gráfica, y variar $\#Lectores$.

Al igual que con Productores-Consumidores, se puede ver que SPIN tiene la tendencia de alcanzar valores cercanos a 0 para cantidades pequeñas de lectores y escritores, mientras que JPF alcanza valores cercanos a la hora para cantidades tan pequeñas como $\#Escritores = 2$ y $\#Lectores = 3$ en la implementación con semáforos.

Además, JPF no termina de ejecutar por falta de memoria, a partir de las combinaciones con valores mayores a:

- Semáforos, $\#Escritores = 3$ y $\#Lectores = 3$
- Monitores, $\#Escritores = 2$ y $\#Lectores = 2$
- Mailbox, $\#Escritores = 2$ y $\#Lectores = 1$, $\#Escritores = 1$ y $\#Lectores = 2$

Filósofos Comensales

Para este problema, se tiene un parámetro (*Cantidad de Filósofos*) por lo que no es necesario fijar ningún otro.

Al igual que con los otros problemas, SPIN tiende a 0 con valores pequeños de $\#Filósofos$, mientras que JPF sigue mostrando su comportamiento exponencial. Por otro lado, en este problema se puede ver que JPF tiene el problema de falta de memoria para valores relativamente pequeños de $\#Filósofos$ (4 para el caso de Monitores).

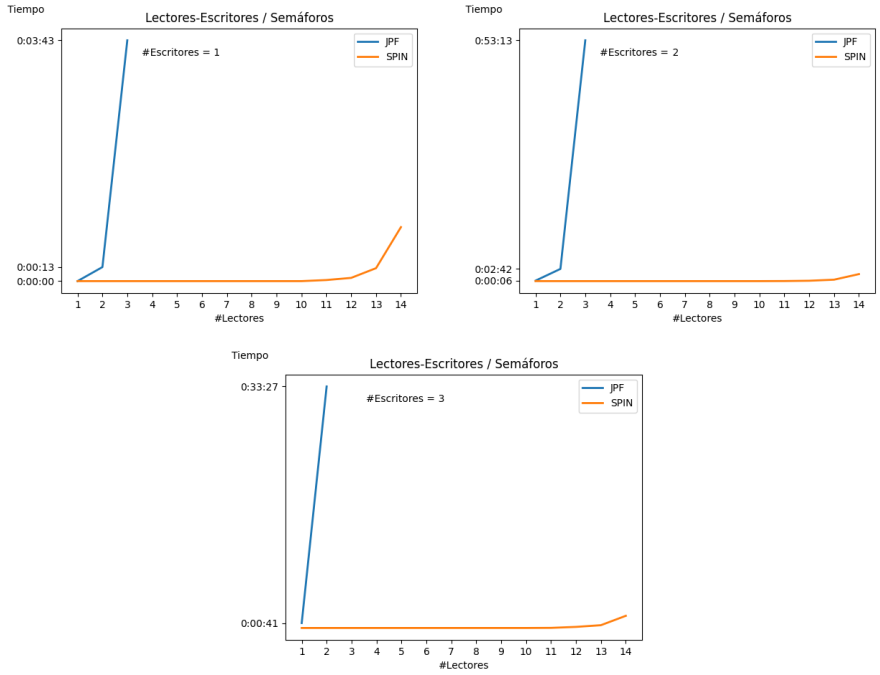

Figura 4.6: Semáforos - Tiempos de ejecución vs $\#Lectores$

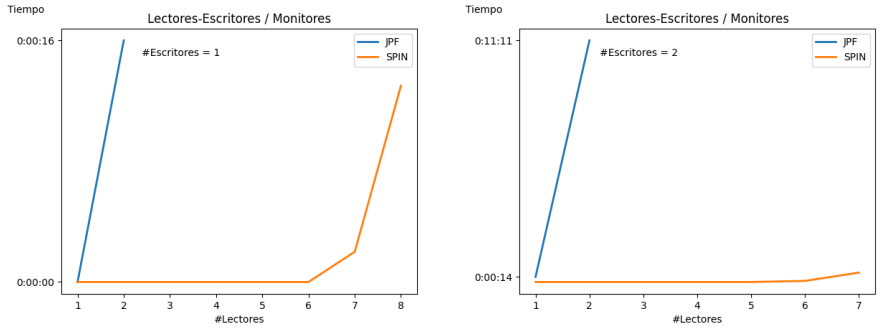

Figura 4.7: Monitores - Tiempos de ejecución vs $\#Lectores$

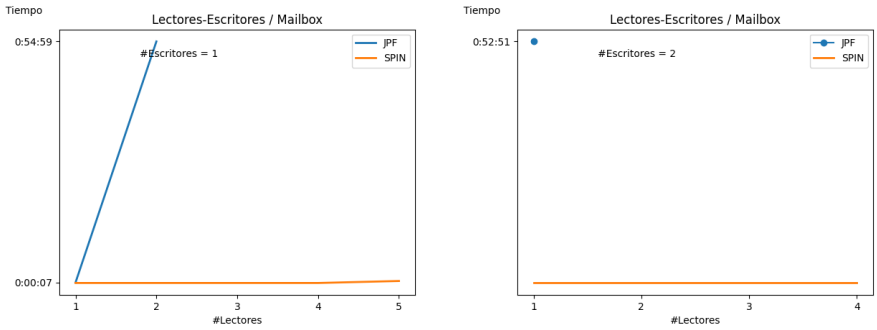

Figura 4.8: Mailbox - Tiempos de ejecución vs $\#Lectores$

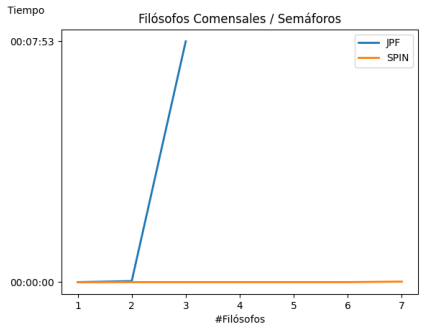

Figura 4.9: Semáforos - Tiempos de ejecución vs $\#Filósofos$

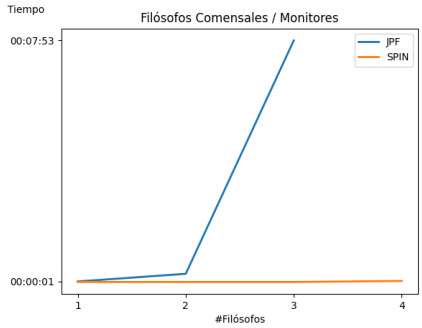

Figura 4.10: Monitores - Tiempos de ejecución vs $\#Filósofos$

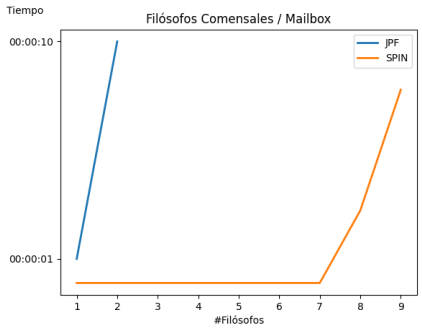

Figura 4.11: Mailbox - Tiempos de ejecución vs $\#Filósofos$

Como se menciona en el capítulo 3, model-checking se basa en la construcción de modelos para el análisis del código. Por lo que un lenguaje pensado explícitamente para esto será mucho más eficiente ante uno más genérico. Esto se puede ver en los tiempos de ejecución de SPIN (que utiliza Promela) sobre JPF que trabaja directamente sobre código escrito en Java [10].

La diferencia en tiempos de ejecución entre ambas herramientas resulta ser muy significativo y es una gran limitante al momento de decidir qué tipo, y sobre todo, qué tamaño de problemas se pueden abarcar con JPF.

Implementación de problema de examen

Como se puede apreciar hasta ahora, la comparativa apunta a que JPF no podría ser utilizado como herramienta de apoyo. Se decide abordar una última comparativa con el fin de medir las herramientas ante un problema de examen, que a priori son de mayor complejidad. En este caso un problema de concurrencia del examen **Julio 2019**.

La letra plantea la situación de una planta nuclear, donde trabajan 4 empleados, que tienen acceso a 4 paneles y cada empleado necesita en exclusividad 2 de esos paneles. Por otro lado existe un supervisor que también tiene acceso a esos paneles, donde revisa su funcionamiento de a 2 a la vez. La supervisión tiene prioridad sobre el uso de los paneles y además en caso de encontrar un problema, todos los empleados deben esperar a que se resuelva para volver a utilizar los paneles.

Este problema es elegido debido a que necesita como mínimo 5 procesos para cubrir todos los casos posibles y muestra de buena forma el alcance de las herramientas exploradas.

Otro de los problemas a resolver en este caso son las funciones auxiliares que aparecen en la letra y que tienen un impacto en el funcionamiento:

- *int que_verif()*: Ejecutado por el supervisor para obtener un panel para verificar. Debe cubrir las diferentes combinaciones de pares de paneles, lo que hace crecer bastante el espacio de soluciones a verificar por las herramientas.
- *bool verif_paneles(int p1, int p2)*: Devuelve **true** si no se encuentran anomalías en los paneles y **false** en caso contrario.

Para simular estas funcionalidades se deben realizar implementaciones adicionales en SPIN y JPF. En el siguiente capítulo se profundiza sobre las implementaciones de funciones auxiliares.

Como resultados, y como era de esperarse, ante el crecimiento del espacio de estado a explorar se aumenta el tiempo de ejecución. Por el lado de SPIN se llega a un tiempo

promedio en 5 ejecuciones de **170.6 segundos** (2:50 minutos). Por otro lado, JPF no llega a terminar la ejecución y luego de más de una hora de ejecución termina por falta de memoria.

4.10.6. Conclusión sobre la comparativa

Una vez finalizado el análisis se pueden encontrar en ambas herramientas diferencias claves al momento de cómo y cuándo utilizar cada una, con sus puntos a favor y en contra. A pesar de esto, se destaca las similitudes de SPIN y JPF, ya que ambas tienen la capacidad de detectar los principales errores cometidos en la programación concurrente (deadlocks, data races, etc) y, además, una salida muy similar. Es por esto que se considera que ambas pueden proveer de soporte a los profesores y estudiantes para distintos fines.

La gran desventaja de SPIN es la complejidad de su lenguaje, Promela. La dificultad de uso inherente de su paradigma orientado a modelos, y las consecuentes diferencias en la sintaxis y semántica en cosas tales como las estructuras de control con respecto a Java, hacen que la curva de aprendizaje para los estudiantes sea mayor al momento de intentar utilizarla. Sin embargo, existen también otras utilidades para la herramienta. Una de ellas podría ser la de pre-implementar algunos problemas (por ejemplo, Productores-Consumidores) y utilizar SPIN para analizar diferentes aspectos del problema y de la solución al mismo. En este punto, un elemento de SPIN que podría resultar de suma utilidad es la ejecución interactiva. También se podría trabajar realizando un análisis sobre los grafos de espacios de estados generados por SPIN para algunos problemas pequeños.

Sumado a esto, a diferencia de JPF, SPIN tampoco cuenta con una implementación nativa para semáforos y monitores. Es decir, ambas primitivas tienen que ser construidas e implementadas de forma manual. Si bien esto es posible, termina resultando en un problema al momento de leer la salida y detectar en qué línea se produjo cada error (deadlock, data race, etc), ya que SPIN muestra directamente la línea a más bajo nivel, lo cual podría ser demasiado confuso para detectar al origen del error.

A pesar de lo mencionado anteriormente, el uso de JPF es descartado ya que su limitación en términos de eficiencia al momento de resolver problemas con model-checking sobre Java son demasiado grandes en comparación con SPIN. Los tiempos de ejecución presentan un crecimiento exponencial y hasta se presentan casos donde la verificación se detiene por falta de recursos. Es decir, esto limitará la cantidad de procesos y resultados al momento de trabajar con JPF.

Este último punto es el que hace descartar la herramienta, ya que los resultados son muy evidentes. Al comienzo de las pruebas esta diferencia no había sido detectada, ya que en primeras etapas las pruebas se enfocaban en cómo detectaba (y si es que lo hacía) los problemas de la programación concurrente como deadlocks o data races y en este sentido JPF no mostraba las limitaciones de eficiencia que mostró al momento de la implementación de una solución de un problema concreto.

Por lo anteriormente mencionado, se concluye que SPIN es la herramienta que mejor se adapta a las necesidades del curso. En el siguiente capítulo se estudia la herramienta en mayor profundidad.

Capítulo 5

SPIN

En este capítulo se detallan algunas características y limitaciones de la herramienta, presentándolas desde un enfoque más técnico.

Se comienza destacando algunos tipos de datos nativos de Promela [16]:

- *bit* (1 bit).
- *bool* (1 bit).
- *byte* (8 bits).
- *int* (32 bits).
- *chan* Tipo channel.

Otras definiciones importantes son:

- *typedef* Definir un nuevo tipo de datos estructurado.
- *inline nombreFunc* define una función estática.
- *active proctype nombreFunc* define una función dinámica que es ejecutada al comienzo de la llamada al archivo de Promela.

Por último, la palabra reservada *atomic* se encarga de asegurar la atomicidad de todo lo que se ejecute dentro del cuerpo de la declaración.

Finalmente, como se menciona en la sección 4.10.2, si existe un error de ejecución de algún tipo (puede ser de concurrencia o no), se genera un archivo de traza (o *trail*) *.pml.trail* que contiene el camino recorrido por el algoritmo para llegar al error. Para visualizar los pasos se ejecuta *spin -t [NOMBRE DE ARCHIVO].pml* (notar que es sobre el archivo *.pml* original y no sobre el *trail* generado), además existen otras banderas opcionales que puede ayudar al seguimiento del error y agrega detalles a cada paso mostrado, como: -l (valor de las variables locales) o -g (valor de las variables globales).

5.1. Interpretación de la salida

Uno de los aspectos más importantes del uso de una herramienta de detección de errores en general (en especial si es de concurrencia) es el formato de la salida y la interpretación del mismo.

SPIN tiene varios tipos de ejecuciones, siendo las mas relevantes model-checking, ejecución aleatoria y ejecución interactiva. Una vez se tenga implementada la solución a un problema, los comandos para ejecutarlo de cualquiera de estas formas son los siguientes:

- Ejecución aleatoria: *spin [NOMBRE DE ARCHIVO].pml*.
- Ejecución interactiva: *spin -i [NOMBRE DE ARCHIVO].pml*. Una vez ejecutado el comando en consola, se muestra una lista con el identificador de cada proceso, el nombre y qué sentencia le corresponde ejecutar. Pudiendo el usuario seleccionar uno y así guiar la ejecución.
- Ejecución de validación con model-checking: *spin -run [NOMBRE DE ARCHIVO].pml*. Donde, en caso de ser necesario un aumento de la profundidad máxima que puede explorar el autómata generado, se puede especificar utilizando la bandera *-m*.

Un resultado de una ejecución exitosa con model-checking se puede ver en la imagen 5.1, donde en el recuadro **1** se muestran datos estadísticos y de configuración de SPIN al momento de la ejecución como lo son el uso de memoria y cantidad de estados visitados, siendo lo más destacado la cantidad de errores (cero en este caso).

Por otro lado, en el recuadro **2** se muestra el tiempo de ejecución en segundos y además un mensaje de *unreached* de los dos procesos involucrados (*productor* y *consumidor*). Este mensaje no es considerado como un inconveniente si solo muestra el estado final de los procesos (*-end-*), ya que estos están dentro de un *while(true)* por lo que nunca terminarán.

En la sección de Implementaciones 5.5 se mostrarán los casos donde SPIN detecta un error y cómo rastrearlo.

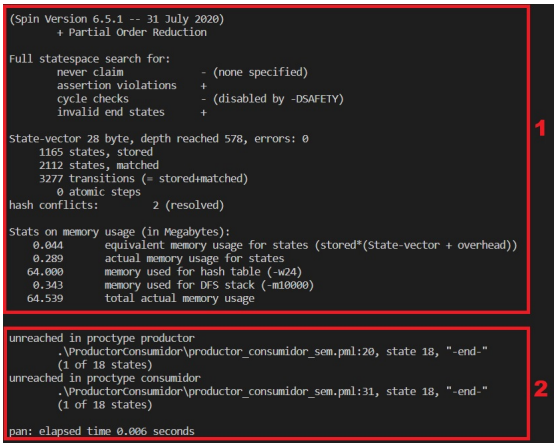

Figura 5.1: Ejecución exitosa utilizando Model Cheking en SPIN

5.2. Módulos

Al estar implementado sobre C, SPIN hace uso del preprocessor de dicho lenguaje para los archivos fuente *.pml*. Dicho preprocessor es invocado antes de que el compilador en sí sea ejecutado y se encarga (entre otras cosas) de incluir el código fuente [4]. Esta herramienta nos da la capacidad de evitar en gran medida la repetición de código.

Los módulos son definidos en archivos *.h* y pueden incluir definiciones de tipos y funciones de Promela, macros del estilo de *C* (definidos con la primitiva $\#define$), etc. Luego, para importar este archivo *.h* y hacer uso de las funciones/tipos provistos, basta con agregar una línea en el archivo original con la siguiente sintaxis: $\#include$ "*ejemplo.h*".

5.2.1. Ejemplos

for.h

Un primer caso de uso es el presentado en el libro *Pinciples of the Spin Model Checker* de Ben-Ari [4]. Dado que Promela no posee una implementación nativa de la estructura de control *for*, en el libro se muestra una haciendo uso de macros y de archivos *.h*:

```
#define for(I,low,high) \
    byte I; \
    I = low ; \
    do \
    :: ( I > high ) -> break \
    :: else ->

#define rof(I) \
    ; I++ \
    od
```

En este ejemplo se define el *for* como un par de funciones for y rof, las cuales abren y cierran respectivamente, el scope de dicho *for*. Luego, para hacer uso de la implementación, basta con incluirla e invocarla:

```
#include "for.h"

...

for (i, 1, CANTIDAD_VENDEDORES - 1)
    if
    :: (largo_min > colasObj.colas[i]) ->
        largo_min = colasObj.colas[i];
        mas_corta = i;
    :: else
    fi;
rof (i);

...
```

monitores.h

Los semáforos y monitores también se pueden implementar usando módulos, las cuales se presentaron en la sección 5.4.2.

De modo de evitar repetir las funciones que refieren a la implementación de monitores en sí (funciones $enterMon$ y $leaveMon$, definición del tipo $Condition$, etc) se puede implementar un único archivo del estilo *monitores.h* el cual contiene todas las definiciones e implementaciones de dichas funciones.

Luego, se agrega la línea *#include "monitores.h"* al principio de archivo que lo utiliza.

```
#include "monitores.h"

...

inline terminar_escribir() {
    enterMon(lect_escr_lock);
    escribiendo = false;
    if
    :: lectBloqueados > 0 ->
        signalC(okLeer, lect_escr_lock);
    :: else ->
        signalC(okEscribir, lect_escr_lock);
    fi;
    leaveMon(lect_escr_lock);
```

```
15  }
16
17  ...
```

Notar que se utilizan las funciones *enterMon*, *leaveMon* y *signalC*, definidas en *monitores.h*.

5.3. Escribiendo modelos eficientes

Dado que es una herramienta basada en model checking, la eficiencia resulta un aspecto crítico a tener en cuenta. La perfomance de SPIN tiene una gran dependencia sobre qué tan eficientemente puede actualizar y buscar una cierta estructura de datos que guarda los estados que ya fueron visitados. La estructura de datos que guardada los estados, llamada *vector de estados*, consiste de los Program Counters de cada proceso, y el valor que toman sus variables. Claramente, la verificación va a ser más eficiente si la cantidad de memoria requerida para guardar los estados es lo más pequeña posible. Algunas de las acciones que se pueden tomar son las siguientes [4]:

- **Usar la menor cantidad de procesos como sea posible**. Por ejemplo, si un proceso sirve solamente para generar datos que van a ser utilizados en un channel, se puede remover dicho proceso e implementar directamente la generación de los datos en el proceso que vaya a utilizarlos.
- **No declarar variables innecesarias**, y **declararlas con un tipo tan ajustado como sea posible**. De esta forma, *byte* es preferible antes que *int*, y *bit* y *bool* son preferibles antes que *byte*.
- **Evitar declarar capacidades de channels que excedan** lo que es necesario para verificar el modelo.
- **Usar *atomic* y *d_step* donde sea posible**, asegurándose de no estar enmascarando posibles estados de error al restringir de forma incorrecta al entrelazado.

5.4. Limitaciones

En el transcurso de este trabajo se mencionan las limitaciones o dificultades para usar esta herramienta. En esta sección se profundiza sobre cada caso.

5.4.1. Data Race

Una de las limitaciones de SPIN es la detección de data races, ya que si se quiere que la herramienta detecte un data race, se debe realizar ajustes en la implementación. Como se ve en el siguiente pseudocódigo, se agrega una nueva variable (*critical*) que es externa al problema, de forma tal que caundo un proceso va a ingresar a la zona crítica suma la variable. La palabra reservada **assert** evalua la condición cada vez, y si no se cumple lanza un error que detiene la ejecución.

```
1  active proctype P() {
2      do
3      ::  printf("..Otras tareas..");
4          critical++;
5          printf("Seccion critica");
6          assert (critical <= 1);
7          critical--;
8      od
9  }
10
11 active proctype Q() {
```

```
12      do
13      ::  printf("..Otras tareas..");
14          critical++;
15          printf("Seccion critica");
16          assert (critical <= 1);
17          critical--;
18      od
19  }
```

Esto, si bien tiene su limitación al tener que agregar código extra a la solución, el lado positivo es que ayuda al estudiante a identificar zonas críticas para poder utilizar bien el **assert**.

5.4.2. Implementación de Semáforos y Monitores

Se realizan las siguientes implementaciones con el fin de encapsularlas en módulos y tener implementaciones más limpias de las soluciones de los problemas.

Semáforos

Un semáforo S, se inicializa como una variable de tipo *byte*, donde sus funciones **P** y **V** se definen como:

```
1  inline P(S) {
2      atomic {
3          S > 0;
4          S--;
5      }
6  }
7  inline V(S) {
8      S++;
9  }
```

Una desventaja de esta definición es que agrega ruido a la salida generada, ya que muestra las líneas internas al llamar a P(S) o V(S): *(S > 0, S = S 1)*

Monitores

Dado que SPIN no contiene una implementación nativa de monitores, se deben mapear utilizando funciones propias de Promela.

Para lograr esto, las *conditions* de monitores se implementan desde cero creando un nuevo tipo estructurado con *typedef*, que cuenta con un booleano para bloquear el proceso y un entero que muestra cuantos procesos se encuentran bloqueados en esa instancia de *condition*. Además, cada función de un monitor está mutuo-excluida por una variable global. En el siguiente algoritmo se muestra la implementación de monitores en Promela:

```
1  typedef Condition {
2      bool gate = false;
3      byte waiting = 0;
4  }
5
6  inline Enter_Mon(boolean lock) {
7      //Primero se bloquea hasta que liberen el monitor
8      //mutuexcluido por lock. Luego toma el monitor.
9      atomic {
10         !lock;
11         lock = true;
12     }
13 }
14
15 inline Leave_Mon(boolean lock){
```

```
16      lock = false;
17  }
18
19  inline wait(condition C, boolean monitorLock){
20      atomic {
21          C.waiting ++;
22          Leave_Mon(monitorLock);
23          C.gate; //Se bloquea hasta que le hagan el signal
24          C.gate = false;
25          C.waiting --;
26      }
27  }
28
29  inline signal(condition C, boolean monitorLock){
30      atomic {
31          if (C.waiting > 0){
32              C.gate = true;
33              !monitorLock; //Espera a que liberen el monitor.
34              monitorLock = true; //Toma el monitor
35          }
36      }
37  }
38
39  inline Pro_Mon_i() {
40   Enter_Mon(lock_i)
41      ...
42      codigo Pro_Mon_i;
43      ...
44   Leave_Mon(lock_i)
45  }
```

Si bien los módulos en general son útiles, ya que permiten importar solo las funcionalidades necesarias, para el caso de Monitores se debe recalcar que al no ser Promela un lenguaje orientado a objetos, no se puede implementar una *clase* para cada monitor, lo que lleva a que las variables y funciones que deberían ser locales al monitor se vuelvan globales a todo el ejercicio. Esto no solo agrega complejidad a la hora de lectura del código, sino que también requiere un mayor cuidado a la hora de implementar más de un monitor, ya que cualquier función puede tener acceso a cualquier variable que se encuentre fuera de ella.

5.5. Implementaciones y ejemplos

En esta sección se presentan implementaciones de distintos ejercicios de parciales o exámenes con el objetivo de mostrar en practica la herramienta. El código completo de cada implementación, como los de los problemas clásicos pueden ser encontrados en: `https://gitlab.fing.edu.uy/leandro.paz/so-herramientas-concurrencia`

5.5.1. Examen Julio de 2014

La letra del ejercicio de concurrencia de este examen plantea una fábrica donde trabajan 8 personas. En la misma hay una maquinaria especial que requiere de por lo menos 3 personas para su encendido, por lo que si la maquinaria estaba apagada, los trabajadores deberán esperar a que haya 3 personas para comenzar a trabajar. Una vez que la maquinaria está en funcionamiento, pueden sumarse o restarse personas (si en algún momento no hay personas, ésta se apaga). Se debe controlar el acceso a esta maquinara y encenderla/apagarla según corresponda.

Por otro lado, la fábrica cuenta con un único baño con dos gabinetes, donde no puede haber dos personas de distinto sexo al mismo tiempo. Lo que genera un control de ingreso y salida del baño de forma mutuexcluida según sexo y cantidad de personas dentro del baño.

Como suele suceder en estos ejercicios, existen funciones auxiliares que ayudan a modelar la realidad planteada y deben ser implementadas de alguna forma. Estos son:

- queSoy():hombre, mujer que llamado por una persona indica su sexo.
- queHacer():trabajar, descansar, baño que llamado por una persona indica que es lo próximo que va a hacer.
- trabajar() que es llamado por una persona para trabajar
- descansar() que es llamado por una persona para descansar
- usoGabinete() que es llamado por una persona para utilizar un gabinete.

En el siguiente fragmento de código se muestra un ejemplo de cómo se pueden implementar este tipo de funciones auxiliares en Promela:

```
1  int ACCION_TRABAJAR = 0;
2  int ACCION_IR_SANITARIO = 1;
3  int ACCION_DESCANSAR = 2;
4
5  proctype trabajador() { //Procedimiento trabajador
6      int queHacer;
7      int queSexoSoy;
8      select(queSexoSoy: 0..1); //Implementacion de queSoy();
9      do
10     ::  select(queHacer: 0..2); //Funcion queHacer();
11         if
12         :: (queHacer == ACCION_TRABAJAR) ->
13             maquinaria_trabajar();
14             trabajar();
15             maquinaria_salir();
16         :: (queHacer == ACCION_DESCANSAR) ->
17             descansar();
18         :: (queHacer == ACCION_IR_SANITARIO) ->
19             sanitario_entrar(queSexoSoy);
20             usarGabinete();
21             sanitario_salir();
22         :: else
23         fi;
24     od
25 }
```

Las funciones auxiliares *trabajar*, *descansar* y *usarGabinete* se implementan como una impresión en consola, ya que no afecta el funcionamiento del problema.
Por otro lado, las implementaciones de *queSoy* y *queHacer* se utiliza la función *select(var, ini, fin)*, que asigna a la variable *var*, un valor de tipo entero que se encuentre entre *ini* y *fin*.

5.5.2. Examen Febrero de 2017

En este examen se plantea un cruce de una carretera entre autos y camiones de carga. Los camiones deben atravesar la carretera para realizar sus tareas de carga y regresar, mientras los autos van sobre la carretera. Si en algún momento hay 5 o más camiones esperando para atravesar la carretera, estos pasan a tener prioridad sobre los autos.

La solución se implementa utilizando semáforos, mutuo excluyendo la carretera según el vehículo.

En la figura 5.2, se muestra el resultado de la ejecución de la solución implementada. Nuevamente aparecen los estados de *unreached* que marcan solo el final de los procesos concurrentes (en este caso *proc_auto* y *proc_camion*) y no se considera un inconveniente y, además, al comienzo se muestra la cantidad de **errores** encontrados, en este caso **0**.

De acuerdo al escenario planteado, pueden darse diferentes errores que SPIN muestra en consola de diferente manera y apoyándose en el archivo de *trail* generado.

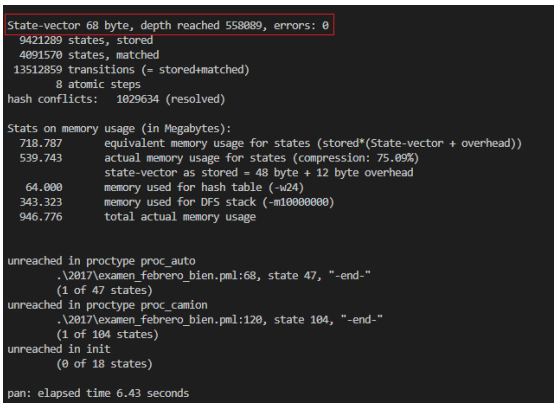

Figura 5.2: Ejecución de la solución utilizando Model Checking en SPIN

Deadlock

Para este problema particular pueden haber varios casos donde se produzca un deadlock. Por ejemplo, si un proceso auto quiere pasar por la carretera (mutuo excluida por un semáforo *mutex*), primero debe obtener el paso, realizando un *P(mutex)*. Si al obtenerlo y cuando finalice de cruzar no libera este *mutex*, *V(mutex)*, cualquier proceso que quiera obtenerlo quedará bloqueado en ese semáforo, quedando el sistema en estado de deadlock.

Una vez ejecutado SPIN, el mensaje de salida es muy similar al de la figura 5.2, con la diferencia que muestra que hubo un error y además en la consola avisa que creó el archivo de trail: *pan: wrote .[nombre_archivo].pml.trail*. Al ver el contenido se muestra paso a paso qué proceso iba ejecutando y cómo las variables del problema iban cambiando, dependiendo del camino tomado y la cantidad de procesos que se ejecuten concurrentemente es qué tantas líneas tendrá el archivo.

Como se ve en la figura 5.3, la consola muestra los valores finales de las variables del problema una vez que el programa terminó su ejecución. Solo uno de los procesos (el de inicialización *init*) termina en un estado válido mientras que los otros nueve (en este caso tres procesos *auto* y seis *camiones*) quedan en un estado incompleto. Además, se muestra cómo la variable *camionesEsperando* finalizó con el valor seis, siendo un estado no correcto dado el problema planteado.

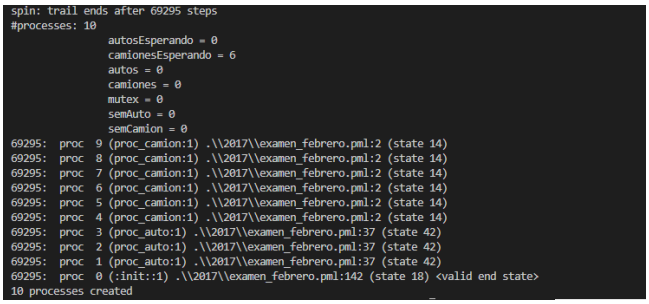

Figura 5.3: Líneas finales del archivo *.pml.trace* generado en caso de error en la verificación.

Data Race

Como se comentó en 5.4, la detección de data races puede llegar a ser más complicadas ya que depende de agregar código extra a la implementación. En este caso se utiliza un contador de procesos en la sección crítica (en este caso, la carretera dentro del semáforo *mutex*), donde cada vez que un proceso entra a la sección crítica, éste suma uno a ese contador y si en algún momento hay dos procesos (vehículos) cruzando la carretera, la aserción será violada y SPIN detendrá su ejecución mostrando un error.

Esto puede darse si, por ejemplo, el proceso camión no pide el semáforo *mutex* para obtener la carretera y cruzar. Si se ejecuta el verificador al final el log de SPIN es casi igual que para el caso de un **deadlock**, donde se imprime la existencia de un error y en qué archivo escribió el *.pml.trail*. La mayor diferencia es que también aclara que se violó una aserción: *pan:1: assertion violated (critical <= 1) (at depth 26)* que muestra que más de un proceso estuvo en la sección critica.

5.5.3. Examen Febrero de 2020

En este examen el ejercicio de programación concurrente se debe resolver utilizando **mailbox**. El ejercicio plantea simular una muestra de un parcial, donde existe un salón con una cantidad máxima de alumnos que pueden ingresar a la vez, que son atendidos por orden de llegada. Por otro lado, por cada ejercicio que el alumno quiera revisar existe cierta cantidad de profesores asignados a ese ejercicio.

En el siguiente fragmento de código se muestra una parte de la solución del problema. Se muestran los diferentes mailbox que se utilizan y cómo estos se implementan y usan en SPIN. Se recuerda que los **channels** en Promela son de receive bloqueante y send no bloqueante.

```
1  #include for.h
2  #define CAPACIDAD_SALON 10
3  #define CANTIDAD_EJERCICIOS 3
4
5  typedef array { int mutex[CAPACIDAD_SALON] };
6
7  chan salir = [1] of { byte };
8  chan salon = [CAPACIDAD_SALON] of { byte };
9  chan numero[CAPACIDAD_SALON] = [1] of { byte };
10 chan mbx_mutex = [1] of { array };
11
12 proctype Estudiante(int numExamen) {
13     byte posicion;
14     array lista;
15
16     salon ? posicion ->
17         for (i, 0, CANTIDAD_EJERCICIOS - 1)
18             * . . . *
19
20             mbx_mutex ? lista ->
21                 lista.mutex[id] = i;
22
23             * . . . *
24         rof (i);
25     salir ! posicion;
26 }
```

Las variables definidas al comienzo se interpretan de la siguiente manera:

- En la línea 6, un mailbox de capacidad 1, donde solo se permite enviar variables de tipo byte.
- En la línea 7, un mailbox de capacidad CAPACIDAD_SALON, donde se aceptan variables de tipo byte.

- En la línea 8, un arreglo de tamaño CAPACIDAD_SALON donde cada lugar es un mailbox de capacidad 1 de tipo byte.
- En la línea 9 se muestra la implementación de un mailbox de capacidad 1, donde su contenido es una variable *array*. Esta es la forma de implementar este tipo de datos en Promela, ya que no existe un tipo array de datos nativo, se debe definir globalmente (línea 4).

Dentro del procedimiento *Estudiante* se ve la utilización del mailbox *salon*, donde en la **línea 16** se representa la espera de su lugar en el salón, permaneciendo bloqueado hasta tener uno. Por otro lado en la *línea 25* se muestra un ejemplo de un *send* al monitor *salir*, representando su salida del salón y devolviendo su lugar. Sumado a esto, se muestra un ejemplo de la utilización de un mailbox de un tipo no nativo como es **mbx_mutex**, que es del tipo definido en la *línea 5*, *array*. Para acceder al campo *mutex*, una vez obtenido el mailbox, se accede de la siguiente forma: *mbx_mutex.mutex*. Ahí se manipula el arreglo.

Finalmente, algo que vale la pena destacar es la utilización del **for** y su importación al comienzo del archivo. La sentencia *for* en Promela no existe de forma nativa, por lo que se realiza una implementación externa en un modulo **.h** de forma de utilizarla en cualquier implementación.

5.6. jSPIN

Se finaliza este capítulo hablando de una extensión de SPIN, jSPIN [4], que es un entorno de desarrollo integrado para SPIN, con una interfaz gráfica de usuario. En la Figura 5.4 se ve un ejemplo de la ejecución utilizando model-checking del problema de productores consumidores, donde jSPIN provee tres secciones: a la izquierda muestra el código del programa en Promela, a la derecha muestra la salida generada por SPIN (sentencias ejecutadas, valores de varibles, resultado de la ejecución, etc) y debajo muestran mensajes de ambos SPIN y jSPIN.

La mayoría de los argumentos utilizados por SPIN, son provistos automáticamente por jSPIN, de manera tal que el usuario simplemente tiene que presionar el botón correspondiente para ejecutar a SPIN en alguno de sus modos. Se pueden agregar y quitar argumentos en el menú *Options*.

jSPIN es implementado utilizando la librería Swing de Java usada para construir interfaces gráficas de usuario. SPIN, el compilador de C, y los verificadores son ejecutados por subprocesos generados a partir de forks de manera tal que se puedan ejecutar los comandos necesarios con los argumentos adecuados. La salida de texto de los subprocesos es devuelto a jSPIN para ser filtrada y mostrada.

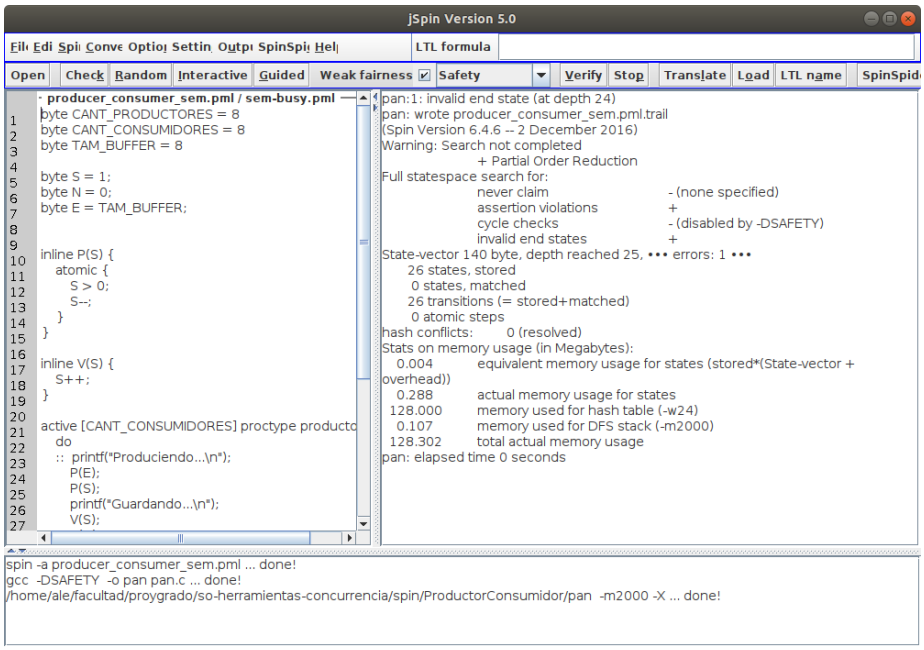

Figura 5.4: Interfaz gráfica de la *jSpin*

Esta página ha sido intencionalmente dejada en blanco.

Capítulo 6

Conclusiones y trabajo futuro

Tanto en el ámbito académico como en el comercial existe una colección sumamente variada de herramientas que asisten en la detección y corrección de errores de concurrencia. Cada una de ellas presenta un enfoque diferente para abordar dichas tareas, desde hacer muestreos obtenidos a partir de múltiples ejecuciones de un mismo programa, hasta utilizar conceptos como el de model-checking.

Por otro lado, la utilización de herramientas que ayuden en el proceso de aprendizaje de conceptos relacionados con la Programación Concurrente en el contexto de la unidad curricular de Sistemas Operativos podría brindar una gran ayuda tanto para alumnos (desde la práctica y para tener una forma fácil y rápida de obtener una devolución de la implementación) como para profesores (desde la transmisión de conceptos de forma práctica).

En este proyecto se analizan múltiples herramientas, siguiendo con especial foco en las herramientas orientadas a model-checking debido a su característica de detección exhaustiva. Las dos principales herramientas analizadas son SPIN y JPF, de las cuales la segunda fue descartada debido a su baja performance al momento de analizar programas incluso de porte relativamente pequeño.

De los objetivos planteados al comienzo del trabajo, con SPIN se logra obtener una herramienta que permite la ejecución y verificación de programas concurrentes, detectando deadlocks y data races. Aunque no resulte visualmente práctico para programas con más de algunas pocas líneas, SPIN también provee la opción de generar grafos de estados que representan el entrelazado de los procesos. También posee la posibilidad de realizar una ejecución interactiva, que si bien no permite volver hacia atrás en la ejecución de los procesos, sí permite la ejecución paso a paso de las instrucciones de código de cada proceso. Se implementa también un conjunto de problemas (tanto en SPIN como en JPF) entre los cuales se incluyen los problemas clásicos de concurrencia vistos en la unidad curricular de Sistemas Operativos, y algunos exámenes de la misma presentados en años anteriores.

Si bien se logra encontrar una herramienta que abarca un buen número de los objetivos del proyecto, cabe destacar que presenta algunos problemas tales como la complejidad en la sintaxis de Promela, la no existencia de primitivas de concurrencia nativas, así como también la mala performance en programas de mediano porte o superior.

Es por esto último que podría resultar interesante realizar una exploración en mayor profundidad sobre otras herramientas que tengan un enfoque distinto al de model-checking (análisis estático, análisis dinámico, etc), y potencialmente combinarlas. Siguiendo esta línea se podría realizar una investigación exhaustiva de como funciona

JPF internamente, con el objetivo de bajar los tiempos y requerimientos de memoria absurdamente grandes.

Otro de los aspectos explorados en el transcurso de este proyecto, pero que finalmente se decide no abarcar, es el de implementar un software especialmente dedicado para el curso. Una de las opciones es la de diseñar alguna especie de transpilador que sea capaz de transformar la sintaxis de un lenguaje similar al utilizado en el curso, a código escrito en Promela, de manera tal que ese código pueda ser analizado con model-checking.

Por otro lado, y siguiendo la línea de lo que aporta SPIN en sí, existe aún terreno para seguir explorando la herramienta en mayor profundidad, ya sea probando distintas configuraciones, o implementando una mayor cantidad y variedad de problemas.

Otra línea que es necesario abordar como trabajo futuro es evaluar los desarrollos realizados en el proyecto como parte de la metodología de la unidad curricular de Sistemas Operativos. En otras palabras, llevar a la práctica la utilización de las herramientas del proyecto en el contexto del dictado del curso. A partir de eso, se puede empezar a obtener un feedback más directo, viendo cómo resulta la interacción entre los estudiantes y las herramientas.

Referencias

[1] Sistemas Operativos 2022. Diapositivas mailboxes. `https://eva.fing.edu.uy/pluginfile.php/330947/mod_resource/content/0/clase13.pdf`. Último acceso: 03/09/2022.

[2] Sistemas Operativos 2022. Diapositivas sección crítica. `https://eva.fing.edu.uy/pluginfile.php/330941/mod_resource/content/0/clase8.pdf`. Último acceso: 28/08/2022.

[3] Mordechai Ben-Ari. *Principles of concurrent and distributed programming*. Pearson Education, 2006.

[4] Mordechai Ben-Ari. *Principles of the Spin model checker*. Springer Science & Business Media, 2008.

[5] Mordechai Ben-Ari. jspin–java gui for spin: User's guide, 2010.

[6] Sam Blackshear, Nikos Gorogiannis, Peter W O'Hearn, and Ilya Sergey. Racerd: compositional static race detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–28, 2018.

[7] Wray Buntine, Bernd Fischer, Klaus Havelund, Michael Lowry, Tom Pressburger, Steve Roach, Peter Robinson, and Jeffrey VanBaalen. Transformation systems at nasa ames. In *Software Transfer Systems 1999*, 1999.

[8] Andrew Bwogi and Tuncay Dagdelen. Configuring java pathfinder for concurrent java programs, 2017. `https://www.diva-portal.org/smash/get/diva2:1105877/FULLTEXT01.pdf`. Último acceso: 28/11/2022.

[9] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 1066–1071. IEEE, 2011.

[10] Louise A Dennis, Michael Fisher, and Matt Webster. Using agent jpf to build models for other model checkers. In *International Workshop on Computational Logic in Multi-Agent Systems*, pages 273–289, 2013.

[11] Milos Gligoric, Vilas Jagannath, and Darko Marinov. Mutmut: Efficient exploration for mutation testing of multithreaded code. In *2010 third international conference on software testing, verification and validation*, pages 55–64. IEEE, 2010.

[12] Milos Gligoric, Lingming Zhang, Cristiano Pereira, and Gilles Pokam. Selective mutation testing for concurrent code. In *Proceedings of the 2013 international symposium on software testing and analysis*, pages 224–234, 2013.

[13] Klaus Havelund, Mike Lowry, and John Penix. Formal analysis of a space-craft controller using SPIN. *IEEE Transactions on Software Engineering*, 27(8):749–765, 2001.

[14] Gerard J Holzmann. Software model checking with spin. *Advances in Computers*, 65:77–108, 2005.

[15] Gerard J. Holzmann. *Explicit-State Model Checking*, pages 153–171. Springer International Publishing, 2018.

[16] Bell Labs in the Unix group of the Computing Sciences Research Center. Spin documentation, 2016.

[17] Kari Kähkönen, Olli Saarikivi, and Keijo Heljanko. LCT: A parallel distributed testing tool for multithreaded java programs. *Electronic Notes in Theoretical Computer Science*, 296:253–259, 2013.

[18] Alexander C Klaiber and Henry M Levy. A comparison of message passing and shared memory architectures for data parallel programs. *ACM SIGARCH Computer Architecture News*, 22(2):94–105, 1994.

[19] Silvana M Melo, Simone RS Souza, Rodolfo A Silva, and Paulo SL Souza. Concurrent software testing in practice: A catalog of tools. In *Proceedings of the 6th International Workshop on Automating Test Case Design, Selection and Evaluation*, pages 31–40, 2015.

[20] Oracle. Package java.util.concurrent. `https://docs.oracle.com/javase/8/docs/api/index.html?java/util/concurrent/package-summary.html`. Último acceso: 10/09/2022.

[21] Peter O'Hearn Sam Blackshear. Open-sourcing racerd: Fast static race detection at scale, 2017. `https://engineering.fb.com/2017/10/19/android/open-sourcing-racerd-fast-static-race-detection-at-scale/`. Último acceso: 28/11/2022.

[22] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*, pages 62–71, 2009.

[23] The Clang Team. Clang 15.0.0 - git documentation.

[24] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated software engineering*, 10(2):203–232, 2003.

[25] Yin Wang, Terence Kelly, Manjunath Kudlur, Stéphane Lafortune, and Scott A Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *OSDI*, volume 8, pages 281–294, 2008.

[26] Benjamin P Wood, Adrian Sampson, Luis Ceze, and Dan Grossman. Composable specifications for structured shared-memory communication. *ACM Sigplan Notices*, 45(10):140–159, 2010.

[27] Ke Zhai, Boni Xu, WK Chan, and TH Tse. CARISMA: A context-sensitive approach to race-condition sample-instance selection for multithreaded applications. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 221–231, 2012.

Índice de tablas

Esta página ha sido intencionalmente dejada en blanco.

Índice de figuras