Tesis de Licenciatura

Model Checking de código para propiedades basadas en eventos

Departamento de Computación

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Tesistas

Miguel Kiszkurno

Hugo Melendez

Roberto Somosa

Director

Victor Braberman

Noviembre, 2007

Resumen

En el presente trabajo se explora la problemática de las técnicas actuales de *Model Checking* y se extiende JPF (Java Path Finder, que es una reconocida herramienta en el área) a los fines de incorporar funcionalidades de verificación de escenarios de eventos, direccionamiento de las búsquedas de defectos y definición de propiedades TypeState (propiedades definidas sobre los objetos definidos dentro del modelo).

El análisis sobre la problemática de las técnicas de *model checking* presentado en este trabajo, tiene como principal foco la búsqueda de alternativas para acotar el universo de estados explorados por los model checkers definiendo patrones de eventos. Este análisis se realiza utilizando el modelo teórico desarrollado por los creadores de SPIN ([HOLZ03]), lo cual da una base concreta para el resto del trabajo.

Utilizando el framework desarrollado, se implementan varios ejemplos clásicos de la ingeniería de software, que permiten evaluar su utilidad y aplicabilidad.

1 Introducción 5
1.1 *Model checking* 5
1.2 Herramientas de *model checking* 6
1.3 Objetivo del presente Trabajo 7
1.4 El presente trabajo 8
2 Modelo de Verificación de Propiedades 10
2.1 Introducción 10
2.2 Breve descripción del modelo de verificación 10
2.3 Proceso de verificación 11
2.4 Discusión 14
3 Propiedades Basadas en Patrones de Eventos 15
3.1 Introducción 15
3.2 Eventos 15
3.3 Patrones de eventos 15
3.4 Escenarios de control 17
3.5 Representación de los patrones de eventos 18
3.6 Discusión 19
4 Java Path Finder 21
4.1 Introducción 21
4.2 Funcionamiento general 21
4.3 Arquitectura general 22
4.4 Search 23
4.5 Generación de estados 24
4.6 Definición de propiedades 26
4.7 Discusión 29
5 Verificación de patrones de eventos con JPF 31
5.1 Introducción 31
5.2 Eventos y patrones de eventos 31
5.3 Verificación de patrones de eventos 33
5.4 Escenarios de control 36
5.5 Typestate properties 38
5.6 Typestate properties sobre clases abstractas 42
5.7 Conclusiones 43
6 Verificación de patrones de eventos con SPIN 46
6.1 Introducción 46
6.2 Funcionamiento general de SPIN 46
6.3 Detección de eventos 50
6.4 Propiedades basadas en patrones de eventos 54
6.5 Conclusiones 60
7 Caso de Estudio 61

7.1 Introducción....61
7.2 Alcance del experimento....61
7.3 Desarrollo....61
7.4 Resultados....71
7.5 Conclusiones....75
8 Conclusiones generales y trabajo futuro....77
8.1 Introducción....77
8.2 Resultados obtenidos....77
8.3 Conclusiones....77
8.4 Trabajo futuro....79
9 Apéndices....82
9.1 Descripción técnica del framework desarrollado....82
9.2 Ejemplo completo de selección de líder....90
10 Bibliografía....92
11 Referencias....94
11.1 Capítulos....94
11.2 Figuras....94
11.3 Ejemplos....95

Capítulo 1

Introducción

1.1 *Model checking*

Las técnicas de verificación establecidas como estándares de la industria de la Ingeniería de Software se basan principalmente en análisis estáticos de los sistemas (revisiones, inspecciones, etc.) o en el empleo de técnicas de testing (Testing Funcional, Testing de performance, etc.) que no pueden ser empleadas para garantizar la ausencia de defectos.

Se ha comprobado empíricamente que tanto el testing (en cualquiera de sus variantes), como las revisiones e inspecciones de código son técnicas que, empleadas correctamente y con metodologías bien definidas, permiten encontrar y corregir a tiempo importantes porcentajes de defectos dentro de cualquier porción de software ([FAG76], [MCC04], [USD02], [MYE79]).

Otras técnicas, orientadas a demostrar la ausencia de errores, habitualmente son estudiadas en ámbitos académicos, ya que las demostraciones matemáticas y el empleo de complicadas fórmulas lógicas necesarias para llevarlas a cabo hacen muy dificultoso su empleo en grandes sistemas. Dichos mecanismos formales, pueden ser utilizados sobre las porciones más críticas del código.

El área de *model checking* permite un enfoque distinto en el análisis de software. Originalmente nació como una técnica de verificación algorítmica orientada a facilitar el hallazgo de defectos en el diseño se sistemas complejos. El proceso de verificación empleado consiste en determinar si un modelo satisface alguna propiedad descripta mediante algún lenguaje formal utilizando herramientas diseñadas específicamente para tal fin. Dichas herramientas permiten verificar el modelo y así determinar si satisface o no la propiedad. En este último caso, se genera un contraejemplo que permite rastrear la secuencia de pasos que llevó al sistema al estado de error.

Analizando el contraejemplo se puede conocer, de manera simple la porción, de código donde se produce el defecto, corregir el modelo y continuar las pruebas iterativamente. De esta manera, se encuentran los errores de manera sencilla y se cuenta con una demostración formal del correcto funcionamiento del sistema.

Habitualmente, las propiedades se escriben utilizando LTL ([GERTH95], [PNU77]), aserciones, predicados sobre los estados o escenarios basados en eventos (como por ejemplo la invocación a métodos de determinados objetos), mientras que el modelo se escribe utilizando algún lenguaje de programación.

Hoy en día se vislumbra un gran avance en todo lo que respecta a *model checking*, a partir del cual se han desarrollado una gran cantidad de herramientas de lo más variadas y sendas teorías que las respaldan. Dichas herramientas no sólo permiten el análisis de modelos como abstracciones de aplicaciones, sino que permiten ejecutar verificaciones sobre sistemas reales (desarrollados en lenguajes utilizados dentro del mercado tales como C++ o Java). De esta manera, es posible analizar el funcionamiento de sistemas críticos verificando el universo de estados del software probado, lo cual disminuye el costo de encontrar defectos y aumenta la efectividad, ya que se cuenta con herramientas que exploran exhaustivamente todas (o una gran parte de) las posibles ejecuciones del software evitando el trabajo manual.

1.2 Herramientas de *model checking*

La mayor fortaleza de los Model Checkers, es la posibilidad de analizar el espacio de estados de un determinado modelo en busca de violaciones a ciertos requerimientos descriptos en algún lenguaje formal. Un típico dominio de aplicación es el de los sistemas concurrentes, donde a menudo (durante la ejecución del software) se da la situación de tener que elegir una de varias posibles instrucciones en un instante.

Desde un punto de vista estrictamente teórico, es posible construir una herramienta que para cualquier sistema desarrollado en algún lenguaje de programación y sobre un universo finito, evalúe todo el espacio de estados, y en caso de encontrar algún defecto (ya sea excepciones no manejadas o el no cumplimiento de alguna propiedad) lo informe al usuario.

No obstante, desde un punto de vista más pragmático, a medida que los sistemas a verificar se hacen más complejos y más grandes, se evidencian ciertas restricciones que son difícilmente superables:

- La difícil escalabilidad: El espacio de estados crece en forma exponencial con respecto a la complejidad del sistema; principalmente si a la “complejidad” se la mide en términos de la cantidad de procesos que participan dentro del modelo.
- El uso de recursos: Todos los componentes (memoria, disco rígido, etc.) de las computadoras donde se ejecutan tienen capacidad finita. Esto, sumado a los problemas de escalabilidad, hace que se restrinjan sensiblemente los sistemas o modelos sobre los que se pueden realizar las verificaciones.
- La complejidad para definir propiedades de manera sencilla: el lenguaje LTL es una opción utilizada para especificar formalmente una amplia variedad de requerimientos de comportamiento. La desventaja que presenta es la complejidad de verificar la correcta escritura de las formulas. Aplicaciones como TimeLine Editor ([SHE01]), a menor escala que LTL, permiten expresar una gran porción de estos requerimientos de manera más simple ([HOLZ01]).

Los problemas antes mencionados son resueltos de diferentes maneras dependiendo de la herramienta. Sin embargo, el límite impuesto por la complejidad algorítmica es muy difícil de

superar; de esta manera, en muchos casos las herramientas no permiten analizar aplicaciones reales, sino versiones simplificadas o prototipos (modelos).

Por lo general, los model checkers implementan algunas optimizaciones sobre las búsquedas, permiten el uso de heurísticas para disminuir la cantidad de estados evaluados utilizan técnicas de Hashing para minimizar el espacio de almacenamiento utilizado al momento de la verificación y descartar caminos ya visitados durante la búsqueda (Partial Order Reduction).

Creemos que estas soluciones ayudan notoriamente en la búsqueda de defectos en el software; sin embargo tienen severas restricciones.

1.3 Objetivo del presente Trabajo

En los últimos años, los model checkers han evolucionado notablemente. Las causas de estos avances pueden encontrarse en el creciente interés de la comunidad científica en desarrollar aplicaciones de alta calidad. Actualmente es posible ejecutar verificaciones sobre código fuente real en lugar de modelos simplificados; por ejemplo, se han diseñado herramientas como JPF [VKBPL02] o Blast [BCHJM04] que permiten ejecutar verificaciones sobre aplicaciones en Java o C++ respectivamente.

En este contexto, es esperable que dichas herramientas continúen evolucionando y extendiéndose. Es por esto que nos planteamos explorar su problemática y algunas restricciones. Buscaremos soluciones para algunos problemas típicos, y analizaremos posibles mejoras. Nos concentraremos principalmente en dos herramientas que cuentan con casos de estudio:

- **SPIN**: es un model checker para sistemas concurrentes ([HOLZ03], [HOLZ97]). Su principal fortaleza radica en que permite modelar y verificar la interacción entre procesos concurrentes y provee un lenguaje de definición de modelos (Promela) muy simple y declarativo.
- **JPF**: es un model checker de código Java [VKBPL02]. Es ampliamente flexible en los mecanismos que provee tanto para definir propiedades como para determinar la forma en que se recorre el espacio de estados. Originalmente fue desarrollado por la NASA para ser utilizado en sus proyectos de desarrollo y ha tomado gran impulso su uso dentro de la comunidad.

Estos model checkers, al igual que la gran mayoría, fueron desarrollados para hacer verificaciones predicando sobre el estado de las aplicaciones y/o modelos. Sin embargo, nosotros creemos que en algunos casos es más simple y efectivo predicar sobre eventos y nos proponemos explorar la posibilidad de adaptar estas herramientas para soportar verificaciones de patrones de eventos.

Más concretamente, elegiremos un lenguaje abstracto y simple de descripción de patrones, pero con el suficiente poder de expresión para cubrir las necesidades del presente trabajo. Luego, para ambas herramientas por separado, evaluaremos las posibles

implementaciones de dichos patrones y los eventos que los componen. Y por último, intentaremos proveer la posibilidad de incluir distintos tipos de escenarios de control en las verificaciones.

En el caso de JPF esperamos definir un framework de desarrollo que cubra los objetivos anteriormente descritos, incluyendo también la posibilidad de definir propiedades typestate[1]. Este framework permitirá a los usuarios definir propiedades complejas con un metalenguaje simple y escenarios de control que restrinjan el espacio de estados sobre el cual se verifica la propiedad.

Creemos que es posible utilizar JPF no sólo para aplicación en proyectos reales sino también como herramienta de investigación. Es por esto que esperamos que el presente trabajo sea un aporte tanto a la industria como a futuros investigadores.

1.4 El presente trabajo

En el capítulo 2 presentamos brevemente un modelo de verificación simple y abarcativo, que luego será utilizado a lo largo de todo el trabajo. El análisis se realiza desde un punto de vista abstracto (sin contemplar las particularidades de cada herramienta), se describen los elementos que componen al modelo, y al proceso de verificación en sí. Partiendo de conceptos totalmente abstractos se logra un primer enfoque orientado a ilustrar la problemática típica de cualquier model checker. Luego, se presentan algunas discusiones de relevancia para el resto de la tesis.

En el capítulo 3 se complementan las definiciones presentadas en el capítulo 2. Comenzamos definiendo los conceptos de Evento y Patrón de Eventos, que son fundamentales para continuar con el desarrollo. Luego, se presentan los escenarios de control, que son mecanismos muy efectivos para acotar el universo de estados recorrido por el verificador. A continuación, se describe el lenguaje seleccionado para representar tanto a las propiedades como a los escenarios de control, y para finalizar, se realiza un análisis más detallado de algunos de los puntos más importantes.

En el capítulo 4 se presenta una descripción de JPF haciendo foco en los puntos más relevantes para nuestro trabajo. Comienza con una descripción general del funcionamiento de la herramienta y su arquitectura; Luego, se explica la forma en que se genera y recorre el espacio de estados; a continuación, se detallan los distintos mecanismos provistos para especificar propiedades; y para finalizar, se detalla el análisis sobre algunos aspectos de la herramienta, que tienen incidencia directa en la forma en que continuamos con el trabajo.

En el capítulo 5 se describen los objetivos planteados para el framework desarrollado, y la solución a la que hemos llegado en cada uno de los casos. Luego, se finaliza con un resumen de las conclusiones obtenidas.

[1] Las propiedades Typestate son utilizadas para verificar requerimientos sobre objetos definidos dentro del modelo. Serán explicadas en mayor detalle en la sección 5.5

En el capítulo 6 se presenta el trabajo realizado sobre el model checker SPIN. Primero se aborda la problemática de cómo representar un evento en una herramienta orientada completamente hacia la verificación de estados; luego se detallan los experimentos realizados, y para finalizar, se presentan las conclusiones derivadas de todo el trabajo realizado sobre esta herramienta.

El capítulo 8 describe las conclusiones generales de la tesis y las distintas alternativas de trabajo futuro descubiertas durante su elaboración

En el capítulo 9 se presentan detalles de diseño e implementación del Framework, las modificaciones concretas sobre el código de JPF y algunos ejemplos para la utilización y ejecución de verificaciones sobre la herramienta extendida.

Capítulo 2

Modelo de Verificación de Propiedades

2.1 Introducción

En la actualidad existe una amplia variedad de herramientas a las que podríamos denominar Model Checkers. No obstante, muchas de ellas difieren entre si, tanto en la forma en que resuelven la problemática, como también en los objetivos a los que apuntan.

Más allá de las diferencias, todas estas herramientas tienen un comportamiento similar con respecto a la forma en que se realiza la verificación. En este capítulo se presenta un modelo de verificación que es aplicable a prácticamente cualquier model checker, ya sea de estados o de eventos.

Presentaremos una descripción general de las herramientas de *model checking* y exploraremos su problemática. El análisis aquí presentado no está orientado a definir una jerarquía de model checkers ni a analizar la brecha entre los distintos tipos de herramientas.

2.2 Breve descripción del modelo de verificación

El modelo de verificación sobre el cual trabajaremos durante la presente tesis, se presenta de forma esquemática en la Figura 2.1.

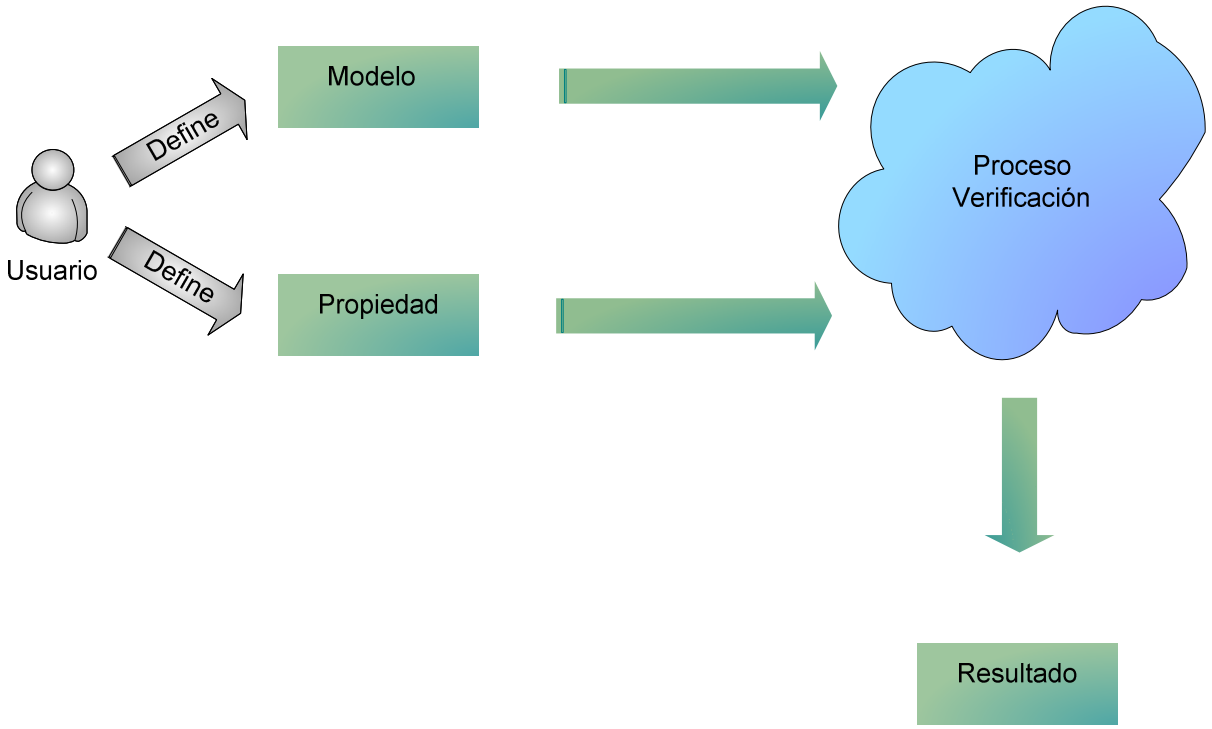

Figura 2.1: Modelo de Ejecución de una herramienta de *Model checking*

En este modelo pueden verse los siguientes elementos:

- **Modelo**: Es el programa o sistema a verificar. Puede tratarse de una aplicación real, o de una especificación (representación más acotada de la misma).
- **Propiedad**: Es el requerimiento a verificar sobre el modelo. El model checker debe proveer un lenguaje de definición de propiedades que permita expresarlas de una manera formal.
- **Proceso de Verificación**: Es el proceso por el cual la herramienta determina si el modelo cumple con la propiedad especificada.
- **Resultado**: Es la respuesta que el model checker informa al usuario luego de haber ejecutado el proceso de verificación. Los posibles resultados son: No se encontró una violación, Sí se encontró una violación (incluye una traza de cómo llegar a la misma).

El proceso de verificación no es estático. En general, es un proceso iterativo dentro del cual el usuario debe ir refinando la propiedad y el modelo a medida que encuentra defectos en los mismos. La forma en que se lleva a cabo el proceso de verificación completo puede describirse de la siguiente manera:

1. Definir el modelo a verificar: El modelo debe desarrollarse en un lenguaje que el model checker comprenda. Por ejemplo en el caso de SPIN, se utiliza Promela, mientras que en el caso de JPF se utiliza Java.
2. Definir las propiedades que el modelo debe cumplir: Al igual que para la definición del modelo, el model checker debe proveer un lenguaje de definición de propiedades no ambiguo. Nuevamente, SPIN acepta definición de propiedades definidas en LTL o Promela, mientras que JPF utiliza Java.
3. Ejecutar el proceso de verificación: En este paso, el model checker verifica la propiedad definida sobre el modelo, ejecutándolo y recorriendo el espacio de estados generado.
4. En caso de encontrarse algún error, corregir el modelo o la propiedad (dependiendo de cuál sea la causa del error) y volver al paso 3.

2.3 Proceso de verificación

En su forma más básica, un Model Checker es un sistema que recibe como input un modelo y una o varias propiedades que el mismo debe cumplir. Luego, permite ejecutarlo y al mismo tiempo verificar que la propiedad se cumpla.

En caso de que la propiedad sea violada, informa al usuario del error y (habitualmente) emite la secuencia de instrucciones ejecutadas que llevaron al sistema a dicho estado. Luego, el usuario tiene la posibilidad de corregirlos (modelo y/o propiedad) y volver a ejecutar la verificación.

Para que la verificación sea efectiva, el model checker debe ser capaz de analizar absolutamente todo el espacio de estados (entendemos por estado, al conjunto de variables

del programa y sus valores en un momento dado de la ejecución, además del call stack y el IP) del modelo en busca de violaciones a la propiedad. Hay distintas estrategias que permiten lograr esto: por ejemplo, SPIN hace una intersección entre los autómatas que representan al modelo y a la negación de la propiedad y verifica que dicho conjunto sea vacío; mientras que JPF permite definir al usuario el algoritmo de exploración del espacio de estados.

En cualquiera de los dos casos, la cantidad de recursos necesaria para recorrer el espacio de estados es muy alta. Esto obliga a realizar algunas optimizaciones que por lo general se enfocan en lograr disminuir el espacio en memoria utilizado para almacenar los estados (utilizando funciones de hashing) y minimizar la cantidad de caminos de ejecución recorridos.

Esto último se logra definiendo un árbol de ejecución con las siguientes características: su raíz es el estado inicial (estado en el que se encuentra el sistema antes de ejecutar instrucciones), sus nodos intermedios son aquellos estados a los que se llega ejecutando una instrucción desde su estado padre y sus hojas son los estados finales (estados sobre los cuales no se puede ejecutar ninguna instrucción, ya sea porque no hay mas instrucciones para ejecutar o porque se llego a la violación de la propiedad).

En la Figura 2.2 se muestran todas las posibles ejecuciones de dos procesos concurrentes con dos instrucciones cada uno. Sin mayor información de contexto, se puede notar que la cantidad de posibles ejecuciones del conjunto de procesos crece exponencialmente a medida que crece la cantidad de instrucciones.

En la Figura 2.3 se muestra el árbol resultante de ejecutar las instrucciones de la Figura 2.2. El estado final de cada ejecución puede encontrarse en las hojas de dicho árbol.

Tiempo	Proceso 1	Proceso 2		Proceso 1	Proceso 2		Proceso 1	Proceso 2
T1	Instrucción 1			Instrucción 1			Instrucción 1	
T2	Instrucción 2				Instrucción 1			Instrucción 1
T3		Instrucción 1		Instrucción 2				Instrucción 2
T4		Instrucción 2			Instrucción 2		Instrucción 2	

T1		Instrucción 1			Instrucción 1			Instrucción 1
T2	Instrucción 1			Instrucción 1				Instrucción 2
T3	Instrucción 2				Instrucción 2		Instrucción 1	
T4		Instrucción 2		Instrucción 2			Instrucción 2	

Figura 2.2: Posibles ejecuciones de dos procesos con dos instrucciones c/u

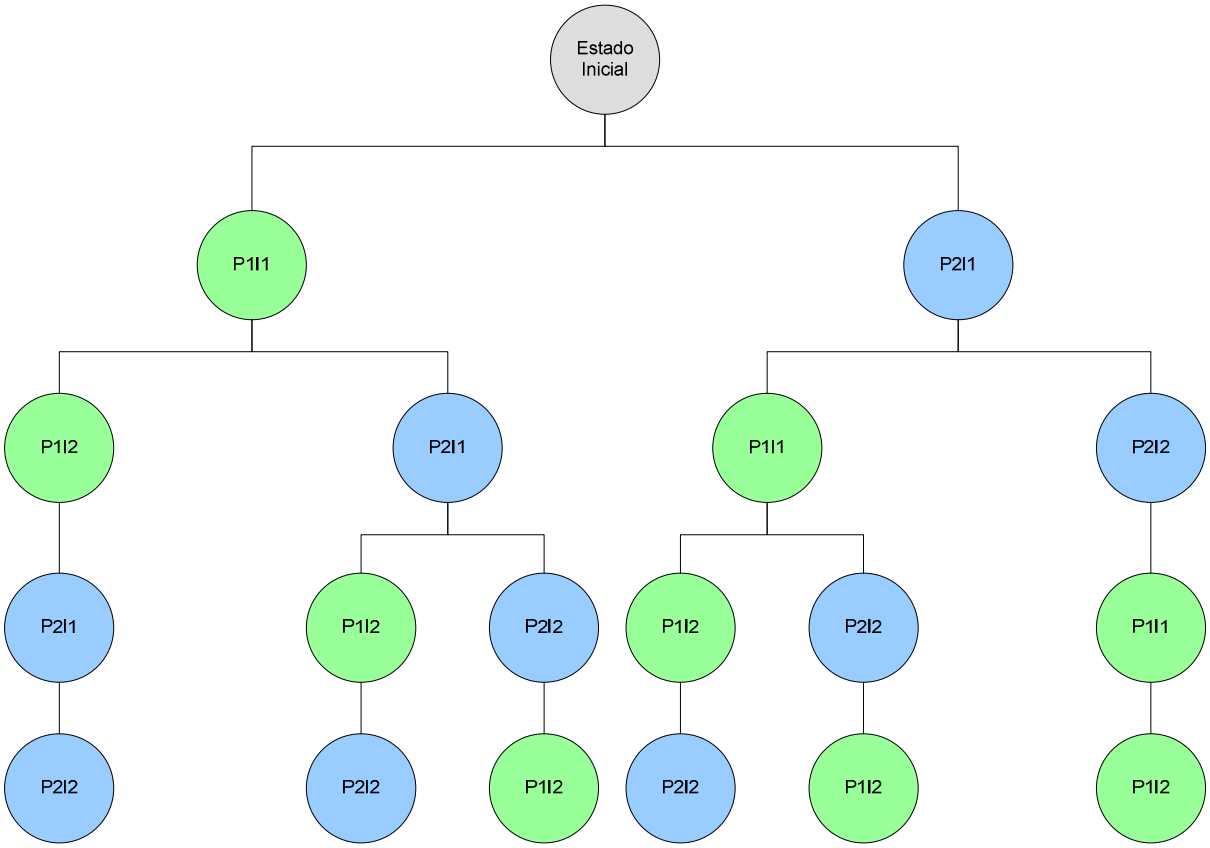

Figura 2.3: Posibles estados de la ejecución de dos procesos con dos instrucciones

Como se mencionó anteriormente, la búsqueda de violaciones podría hacerse recorriendo el árbol de ejecución con cualquier algoritmo conocido (por ejemplo DFS o BFS), pero el crecimiento de los nodos, es exponencial. Es por esto que los model checkers deben implementar optimizaciones que les permitan reducir este espacio.

Supongamos que la instrucción 2 del proceso 1 y la instrucción 1 del proceso 2 de la Figura 2.2 fueran iguales (suponiendo por ejemplo que incrementan en 1 el valor de alguna variable global). En este caso, habría determinados caminos de ejecución equivalentes.

Numerando las ejecuciones de arriba hacia abajo y de derecha a izquierda, la primera ejecución es equivalente a la segunda, mientras que la quinta es igual a la sexta. De esta manera, la cantidad de ejecuciones a explorar se disminuyó en 2.

En la mayoría de los casos, no es importante la instrucción en sí, sino el estado en que queda el modelo luego de ejecutarla. Esto es, si luego de ejecutar la instrucción el modelo queda en un estado ya visitado, entonces dicha rama se puede podar.

De esta manera, a medida que el model checker recorre el espacio de estados debe determinar en cada caso si dicho estado fue visitado, y en caso afirmativo, suspender la búsqueda por dicho camino (esto es, hacer backtracking) y volver al estado anterior dentro del árbol de ejecución.

2.4 Discusión

2.4.1 Clasificación de las Propiedades

El concepto de propiedad es usado a lo largo de todo el presente trabajo, y si bien no daremos una definición formal, es necesario describirlo de manera más detallada. En principio, distinguiremos entre dos tipos de propiedades: asercionales y orientadas a comportamiento.

Las primeras permiten predicar sobre el estado en el que se encuentra el modelo. Se definen en términos de predicados sobre el estado de la ejecución. Habitualmente toman la forma de: “la variable X no puede tener el valor Y” o “si la variable X tiene el valor Y, entonces la variable Z tiene que tener el valor H”.

Las últimas se utilizan para predicar sobre la traza de la ejecución del modelo o del árbol de ejecución. Se definen en términos de eventos que van ocurriendo durante la ejecución del sistema. De esta manera, es posible predicar sobre el camino de ejecución y tener en cuenta la traza de eventos que el modelo genera. Por ejemplo, se podría definir la siguiente propiedad: un proceso no puede recibir mensajes de un canal antes de que algún otro proceso lo llene.

En el presente trabajo, nos concentraremos principalmente en la utilización de propiedades orientadas a comportamiento, predicando sobre los eventos.

2.4.2 *Model checking* de código vs. *model checking* de modelos

El modelo de verificación aquí presentado enmarca distintos tipos de model checkers. Algunos, como es el caso de JPF, están orientados a verificar sistemas reales, mientras que otros, como SPIN son utilizados sobre modelos.

JPF ha sido diseñado para hacer verificación sobre programas escritos en Java. Esto le permite hacer presunciones sobre la forma en que se ejecutará el sistema, y en base a esto realizar algunas optimizaciones durante la exploración del espacio de estados. Por ejemplo: se producen cambios de estado cuando se accede a alguna variable compartida entre distintos threads. Esto elimina muchas opciones de schedulling que sin dicha presunción, deberían ser exploradas, mejorando el tiempo y disminuyendo el uso de recursos. Veremos más adelante, las implicancias que esta decisión tiene para el presente trabajo.

SPIN en cambio, provee un lenguaje propio para definición de modelos (y propiedades), es por esto que no puede hacer presunciones sobre las decisiones de schedulling y debe explorar absolutamente todo el espacio de estados. Esto implica que cada instrucción ejecutada genera un cambio de estado.

Capítulo 3

Propiedades Basadas en Patrones de Eventos

3.1 Introducción

El modelo de verificación planteado en el capítulo anterior, requiere que exista tanto un modelo como una propiedad a verificar sobre dicho modelo. En base a este input, el model checker debe recorrer el espacio de estados del modelo y verificar el cumplimiento de la propiedad.

Según los objetivos planteados para la presente tesis no existen restricciones para el modelo, pero la propiedad debe ser expresada en términos de patrones de eventos.

Los objetivos del presente capítulo son: definir, al menos en forma abstracta, los conceptos de evento y de patrón de eventos; definir un lenguaje simple, intuitivo y expresivo para las propiedades; y presentar brevemente los escenarios de control, que luego serán utilizados para acotar el universo de estados sobre el que se ejecuta la verificación.

3.2 Eventos

En BRAB05 se define a un evento como cualquier cambio observable durante la ejecución de un sistema que pueda ser relevante para la verificación de su correcto comportamiento.

Esta definición así planteada, es muy flexible, en el sentido que hay muy pocas restricciones sobre lo que puede ser considerado un evento; podría ser desde la ejecución de cualquier instrucción observable, hasta la ocurrencia de eventos externos al modelo a chequear (por ejemplo garbage collection, creación o eliminación de objetos en memoria, etc.).

Los eventos ocurren instantáneamente y llevan al sistema de un estado a otro; tienen que ver con “cambios observables” durante la ejecución del sistema. Esto implica que la herramienta de *model checking* debe proveer al usuario los mecanismos para capturar su ocurrencia y actuar en consecuencia. Este problema es complejo y distinto en cada herramienta.

3.3 Patrones de eventos

Intuitivamente, los patrones permiten definir cierto orden en el que determinados eventos deben (o no) ocurrir durante la verificación de un modelo. Esta noción de orden tiene que ver con el momento en que el evento ocurre, y es la que posibilita la descripción de las propiedades o requerimientos que el modelo debe cumplir.

En definitiva, un patrón de eventos no es otra cosa que una relación de precedencia temporal entre distintos eventos dentro de alguna posible ejecución del sistema.

Ejemplo 1: Un modelo abstracto

Se define un modelo en el que dos procesos se ejecutan concurrentemente, cada uno de dichos procesos genera distintos eventos:

- El proceso 1 (P1) genera los eventos A, B y C.
- El proceso 2 (P2) genera los eventos D, E y F.

Enunciada coloquialmente, la propiedad que se desea verificar es la siguiente: "No pueden ocurrir dos eventos A (no necesariamente consecutivos) sin que ocurra un evento E entre ellos".

Según lo definido hasta el momento, prácticamente cualquier instrucción de los procesos puede representar un evento dentro de la ejecución del sistema. Sin embargo, a los efectos prácticos de la propiedad que nos interesa verificar, las únicas instrucciones importantes son aquellas que provoquen la ocurrencia de alguno de los eventos antes mencionados.

Con los conceptos planteados no alcanza para definir la propiedad indicada en el ejemplo. No obstante, podríamos decir que el patrón (cualquiera que fuese) que represente a la propiedad, debería indicar de alguna manera que durante la ejecución del modelo, siempre que ocurran dos eventos del tipo A, debe haber en el medio una ocurrencia del evento E.

Nro Ejecución	Eventos						Resultado
1	A	C	A	E	C	F	No cumple
2	A	A	C	D	F	D	No cumple
3	D	E	A	A	E	A	No cumple
4	D	A	E	E	A	E	Cumple
5	D	F	F	C	A	B	Cumple
Instante	1	2	3	4	5	6	

Figura 3.1: Algunos posibles patrones de eventos del modelo del Ejemplo 1

En la Figura 3.1 pueden verse algunas posibles trazas de eventos del modelo definido en el Ejemplo 1. Es sencillo ver que las ejecuciones numeradas del 1 al 3 no cumplen la propiedad del ejemplo, ya que las ocurrencias de eventos A no se encuentran separadas por ocurrencias de E, mientras que en las ejecuciones 4 y 5, si la cumplen; en el primer caso porque las dos ocurrencias de A tienen en medio dos ocurrencias de E mientras que en el segundo hay una única ocurrencia de A.

Si vemos a una ejecución del modelo como una cadena de eventos, entonces podemos ver al conjunto de todas las trazas que el modelo genera como un lenguaje L_M (que no es necesariamente regular). Por su parte, la propiedad a verificar, también debe ser expresada en términos de eventos para contrastarla con el modelo.

El objetivo de la propiedad es permitir discriminar las trazas correctas (las que no violan la propiedad) de las incorrectas (las que sí la violan). Dicho en otras palabras, genera un lenguaje L_P, que se utiliza para saber si el modelo funciona correctamente.

En este punto, hay dos opciones, o bien definir la propiedad como un requerimiento que el modelo NO debe cumplir, o definirla como un requerimiento que el modelo DEBE cumplir. En el primer caso podemos hablar de antipropiedad, mientras que en el segundo hablamos de propiedad.

Para determinar si el modelo cumple la propiedad (o antipropiedad), es necesario analizar la intersección entre L_M y L_P; utilizando antipropiedades, deberá resultar vacía, mientras que utilizando propiedades no.

3.4 Escenarios de control

En muchos casos es interesante evaluar el comportamiento del sistema en determinados contextos de ejecución a los efectos de reducir la cantidad de estados sobre los cuales se realiza la verificación.

Hay diversos enfoques que nos permiten lograr el objetivo antes planteado. En el presente trabajo nos concentraremos en dos de ellos: Preámbulo y Contexto de ejecución. El primero es una condición que se impone sobre la ejecución del modelo para comenzar la verificación, mientras que el segundo es una condición que se le impone al modelo para determinar, en cada paso, si es correcto continuar la verificación de la rama actual de ejecución.

Al igual que las propiedades, los escenarios de control también generan lenguajes y nos permiten acotar la verificación a un subconjunto de L_M.

En el caso del preámbulo P, la restricción consiste en efectuar la verificación sobre un conjunto de trazas L_{PR} que se define como:

$L_{PR} \equiv \{T \mid P{+}{+}T \in L_M\}$ donde T es una traza, P es el preámbulo y "++" representa la concatenación.

En el caso del contexto C, permite efectuar la verificación sobre un conjunto de trazas L_{VC} que se define como:

$L_{VC} = \{T \mid T \in L_M \wedge T = Pr{+}{+}X \wedge Pr \in a\ L_C\}$ donde T es una traza y L_C es el conjunto de cadenas generado por C.

Continuando con el Ejemplo 1, supongamos que definimos como preámbulo a la cadena de eventos DFF; entonces, la verificación se llevaría a cabo únicamente sobre la ejecución 5, mientras que las ejecuciones 1 a 4 se podarían inmediatamente (de hecho, las primeras dos ejecuciones se podarían al ocurrir el primer evento, mientras que las 3 y 4 al ocurrir el segundo evento).

Supongamos que definimos el contexto sobre el que se hará la verificación como todas las cadenas que no tengan una D y luego una A (sin importar los eventos que ocurran en el medio). En este caso, las ejecuciones 3, 4 y 5 no se verificarán (el model checker decidirá podar esa rama del árbol de ejecución cuando detecte que el contexto ha sido violado), mientras que la 1 y la 2 sí lo harán.

En síntesis, el preámbulo acota el universo sobre el cual se realizará la verificación al conjunto de posibles ejecuciones cuya traza comienza con la cadena de eventos especificada. En cambio el contexto restringe la verificación siempre que la ejecución se mantenga dentro de sus límites.

3.5 Representación de los patrones de eventos

En el contexto del presente trabajo, utilizaremos como estructura de datos para representar a los patrones de eventos, autómatas finitos determinísticos (AFD), que cubren perfectamente todos los requerimientos necesarios para modelar las relaciones de precedencia necesarias entre los eventos.

Es importante destacar que las definiciones tomadas hasta el momento, nos permiten modelar como patrones de eventos a cualquier relación de precedencia temporal entre eventos dentro de la ejecución del sistema; esto es, puede utilizarse tanto para describir las propiedades a verificar sobre el modelo como los escenarios de control [GHNS93].

La Figura 3.2 corresponde a la representación gráfica de la propiedad definida en el Ejemplo 1. Como puede verse, cada vez que ocurre un evento A, el AFD pasa a un estado de espera de otro evento A, en caso de ocurrir, la propiedad fue violada, mientras que si ocurre una E, se "limpia" la primera ocurrencia de A.

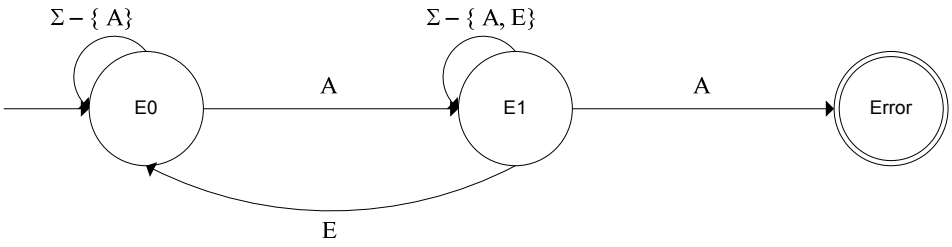

Figura 3.2: Representación gráfica de la Propiedad del Ejemplo 1

En la Figura 3.3 puede verse que el preámbulo llega a un estado de aceptación únicamente cuando consume la cadena de Eventos DFF, en cualquier otro caso, alcanza un estado de error, lo que provocará que se pode el subárbol de ejecución actual y se continúe con las pruebas por otra rama.

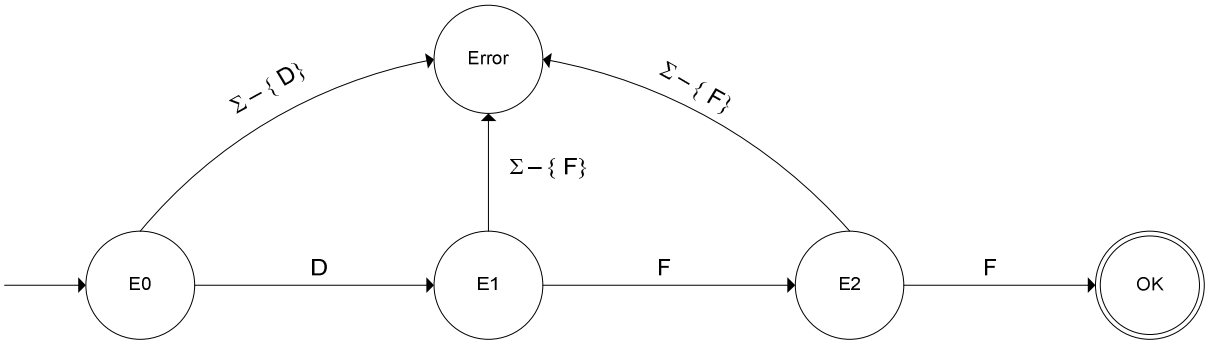

Figura 3.3: Representación gráfica del preámbulo del Ejemplo 1

En la Figura 3.4 puede verse el contexto definido para el Ejemplo 1. En este caso, si ocurriera un evento D y luego uno A (sin importar los eventos intermedios), se estaría violando el contexto, en cuyo caso, el model checker debe podar la rama actual y continuar por la siguiente.

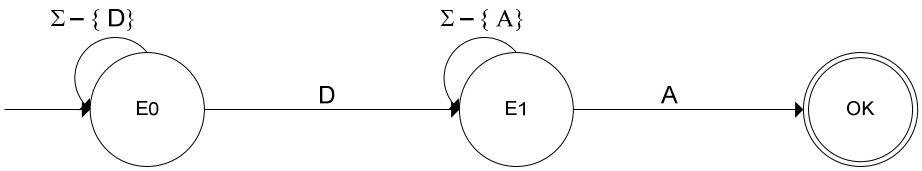

Figura 3.4: Representación gráfica del contexto del Ejemplo 1

3.6 Discusión

En esta sección se analizarán en más detalle algunos aspectos relacionados con las definiciones tomadas a lo largo del presente capítulo.

3.6.1 Eventos y patrones de eventos

Si bien el concepto de evento es fundamental para el presente trabajo, no hemos dado hasta el momento una definición clara. Se ha optado por una definición abstracta y abarcativa. Esta decisión tiene que ver con que el concepto de evento no debe ser, bajo ninguna circunstancia, dependiente de la herramienta con la que se decidiese trabajar.

Podríamos haber acotado arbitrariamente la definición, pero preferimos postergar la toma de esta decisión al momento en que se comience el trabajo con las herramientas. No obstante, es importante destacar que el tipo de eventos sobre los que se intentará trabajar son ejecuciones de determinadas instrucciones que impliquen algún cambio en el estado del sistema (asignación de variables, ejecución de funciones o métodos, etc.).

La definición de patrón de eventos, resulta de alguna manera un poco más intuitiva y más concisa, pero sólo después de haber definido el lenguaje a utilizar para representarlos.

3.6.2 Lenguaje de definición de propiedades

Hay ejemplos de lenguajes visuales, como los presentados en [SHE01] o [ABK04], más complejos y expresivos que el de los autómatas aquí presentados. No obstante, creemos que es un buen comienzo partir de un lenguaje simple, y con algunas restricciones en su poder expresivo, pero lo suficientemente sencillo, intuitivo y expresivo como para permitir especificar y verificar patrones de eventos de manera simple.

Además de la simplicidad, este lenguaje tiene otra ventaja fundamental que es la de establecer, de alguna manera, los parámetros mínimos que cualquier herramienta debe cumplir para ser utilizada para verificar patrones de eventos.

Definir este lenguaje no es el objetivo del presente trabajo sino una herramienta. En este caso, cuanto más complejo es el lenguaje de definición de propiedades menos probabilidades

de éxito tendremos a la hora de adaptar herramientas para su uso. De esta manera, la estrategia minimalista facilita el resto de los objetivos de la tesis, pero sin perder valor.

3.6.3 Escenarios de Control

A los efectos del presente trabajo, se presentaron dos mecanismos distintos para filtrar el espacio de estados recorrido. Existen otras variantes que podrían ser empleadas, pero al definir a las trazas del modelo en términos de lenguajes regulares, estas posibilidades surgieron naturalmente.

Esto se debe a que es relativamente sencillo expresarlas como resultado de operaciones entre lenguajes (no necesariamente regulares). Esto facilita también su representación abstracta, ya que se pueden utilizar las mismas estructuras (AFD) que definen las propiedades.

Capítulo 4

Java Path Finder

4.1 Introducción

JPF es una Java Virtual Machine que permite verificar programas desarrollados en Java. Maneja el concepto de búsqueda dentro del espacio de estados y provee diversos mecanismos para verificar el cumplimiento de propiedades.

Nació como un traductor de Java a Promela (con el objetivo de verificar los programas utilizando SPIN), pero hoy en día es un model checker altamente complejo y flexible.

En el presente capítulo se describe el funcionamiento de JPF. Se presentan las nociones básicas y una explicación a nivel general de los aspectos más importantes de la herramienta. Entre ellos, las clases y paquetes más importantes, la forma en que se genera (y en que se recorre) el universo de estados del modelo y se hace un repaso de las posibilidades que provee para la definición de propiedades.

El presente análisis no es exhaustivo, se enfoca principalmente en dar una idea general de la herramienta y cubrir los aspectos necesarios para entender las decisiones tomadas durante la construcción del framework desarrollado como parte de este trabajo.

4.2 Funcionamiento general

JPF es una Java Virtual Machine (JVM) que provee distintos mecanismos para observar el modelo durante su ejecución.

Habitualmente, la ejecución del modelo se lleva a cabo en el contexto de una verificación. Esto implica que JPF no recorre un único camino de ejecución, sino que los recorre todos. A medida que avanza en la ejecución, busca la existencia de violaciones a las propiedades definidas (como deadlocks o excepciones sin capturar); en caso de encontrarlas, lo reporta al usuario junto con la traza de instrucciones que llevaron al sistema al estado de error.

Mientras ejecuta el modelo, JPF detecta la ocurrencia de nuevos estados y los registra en una estructura de datos propia. En cada transición entre estados (esto es, cuando la ejecución finaliza el proceso del estado actual y pasa al siguiente) JFP verifica el cumplimiento de las propiedades especificadas.

La forma en que JPF recorre el espacio de estados es utilizando el algoritmo DFS. Al llegar a un estado final o uno ya visitado realiza un backtrack para evaluar otras posibles alternativas de ejecución. Existe un subconjunto de instrucciones o bytecodes capaces de generar nuevos estados y son, en principio, aquellas que pueden ocasionar problemas de

concurrencia en escenarios multithread (acceso a variables compartidas, métodos estáticos, etc.).

Las propiedades se definen utilizando alguno de los tres mecanismos que provee: **Aserciones**, **Listeners** y **Properties**.

Las **Aserciones** se utilizan instrumentando el código del modelo para establecer predicados que deben ser verdaderos en el momento de evaluarse.

Las **Properties** permiten predicar sobre el estado, tanto del modelo como de la JVM. La verificación de las distintas propiedades asociadas a un modelo se realiza cuando JPF determina que hay una transición (es decir un pasaje de un estado a otro).

Los **Listeners** son observadores de la ejecución del modelo que brindan los mecanismos necesarios para suscribirse a ciertos eventos relacionados a la búsqueda, como ser una transición de un estado a otro; o relacionados a la maquina virtual (VM), como son la ejecución de una instrucción o la creación de una variable. Cuando un evento ocurre, el Listener es informado para ejecutar las acciones que hayan sido definidas para la verificación.

En lo que queda del presente trabajo, el término **evento** se utilizará para referirse a la ejecución de cualquier bytecode o a cualquier evento externo al modelo (por ejemplo la ejecución del Garbage Collector, avanzar o retroceder en el espacio de estados, etc.); mientras que el término **estado** se utilizará para referirse al estado interno de la JVM (threads del modelo, valores de variables estáticas y variables dinámicas).

4.3 Arquitectura general

Como hemos mencionado JPF es, esencialmente, una Java Virtual Machine. No obstante, tiene algunas diferencias que le permiten ejecutar el modelo de forma de recorrer el espacio (finito) de estados completo.

La implementación fue hecha en Java y su arquitectura general consta de tres componentes principales: **JPF**, **VM** y **Search**.

- **JPF**: este componente tiene como principal responsabilidad configurar e instanciar los objetos clave de JPF, como ser la JVM y el Search.
- **VM**: es la implementación de la Java Virtual Machine. Es la responsable de ejecutar el modelo y administrar los estados (creación, almacenamiento y acceso).
- **Search**: es una interfaz de búsqueda para recorrer el espacio de estados. El principal objetivo de esta clase es recorrer el espacio de estados generado en cada paso de la verificación y determinar cuándo es necesario realizar backtracking.

La siguiente figura muestra de manera dinámica la interacción entre los componentes antes mencionados:

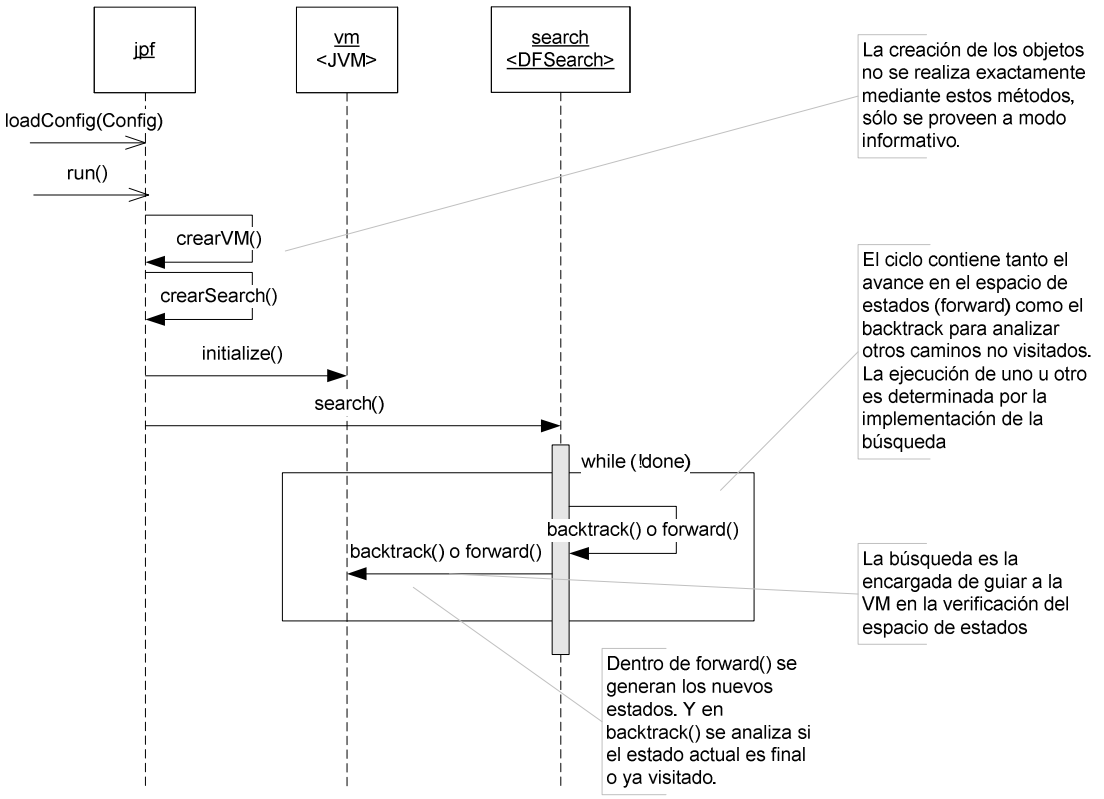

Figura 4.1: Diagrama de secuencias de la arquitectura de JPF

Como se observa, JPF carga la configuración de la verificación y crea los objetos que luego se utilizarán. A continuación, entre **Search** y **VM** exploran el espacio de estados y verifican las propiedades especificadas.

4.4 Search

Como mencionamos anteriormente, el componente **Search** es el encargado de guiar la verificación dentro del espacio de estados. El vínculo existente entre **Search** y **VM**, le permite al primero obtener información sobre el estado general de la ejecución, como por ejemplo, saber si el estado actual es final o conocer la naturaleza de las instrucciones ejecutadas hasta el momento, lo que determina si las mismas son propensas a generar nuevos estados de verificación.

Como parte de JPF se proveen, entre otras, las siguientes implementaciones de búsquedas:

- **DFS**: donde se ejecutan todas las instrucciones hasta llegar a un estado final, en cuyo caso se realiza un backtrack para analizar otro camino de ejecución.
- **HeuristicSearch's**: **JPF** cuenta con diversas clases que implementan exploraciones parciales del universo de estados. De esta manera, el espacio de estados explorado, se acota utilizando distintas heurísticas.
- **RandomSearch**: Este algoritmo ejecuta un único camino de ejecución de manera random. Esto es, no realiza backtrack; es una emulación de una JVM estándar.

La implementación de DFS se comporta según se describe en la siguiente figura.

```
Mientras queden ramas sin explorar
  Si no hay nuevos estados o el estado actual es final
    Si existe la posibilidad de backtrackear ->
      Hacer backtracking
    Sino ->
      Finalizar la búsqueda
  Sino
    Avanzar un estado
    Verificar las propiedades
Volver a intentar
```

Figura 4.2: Pseudo código del algoritmo DFS de búsqueda

Podemos observar el ciclo que recorre todos los estados existentes. Al llegar a un estado final o si se determina que no hay estados nuevos para visitar, se hace backtrack para analizar el resto de las posibilidades. Si se determina que el backtrack no es factible, la búsqueda finaliza y con ella la verificación.

Mientras queden estados por visitar, como vemos en la Figura 4.1, la búsqueda delega la responsabilidad de avanzar la ejecución a la **VM** usando el método **forward()**. Dentro del mismo es donde se decide la generación de los nuevos estados.

4.5 Generación de estados

Para entender bien el funcionamiento de JPF, es importante entender cómo se generan los nuevos estados a partir de las instrucciones. Ya se mencionó que la **VM** genera los estados, mientras que el **Search** determina la siguiente acción de la **VM** (crear un nuevo estado, volver a ejecutar uno, etc.).

Desde el punto de vista de la generación de estados, la verificación de un modelo puede resumirse en los siguientes pasos:

1. En la inicialización de **JPF**, se generan todos los objetos principales (**VM** y **Search** entre otros), se genera un estado inicial, y se da comienzo a la verificación (el **Search** realiza un **forward**).

2. Dentro de la **VM**, comienzan a ejecutarse las distintas instrucciones que conforman el modelo, actualizando el estado general del sistema a medida que se avanza. Todas las instrucciones ejecutadas hasta ese momento se agrupan en el estado inicial.

3. Si la instrucción actual, considerando el contexto donde se está ejecutando (esto incluye los threads actuales, instancias de objetos existentes hasta el momento, etc.), es candidata a crear una bifurcación, la **VM** cierra el estado actual, genera un nuevo estado por cada posible bifurcación y retorna el control a **Search**.

4. **Search** toma en cuenta la bifurcación existente para decidir cuál es el siguiente estado a analizar, según vimos en Figura 4.2. Si bien la bifurcación plantea un nuevo camino en la ejecución, éste puede derivar en un estado conocido y **Search**

es el encargado de realizar el backtrack necesario para analizar las otras posibilidades.

5. La ejecución de las instrucciones continúa hasta que las nuevas bifurcaciones creadas sólo derivan en estados ya visitados. En este momento es donde se realiza el backtrack para visitar los demás estados generados y cubrir la totalidad del espacio de estados.

Podemos observar en el paso 3, que la generación de nuevos estados está fuertemente ligada a la implementación que realiza **JPF** de las distintas instrucciones utilizadas en el modelo. La **VM** ejecuta las instrucciones que conforman el modelo hasta que alguna de las siguientes condiciones ocurre:

1. la próxima instrucción produce un valor no determinístico (random).
2. la próxima instrucción puede tener efectos colaterales que afecten otros threads, como por ejemplo, acceso a memoria compartida por intermedio de objetos en común.

La primera condición se produce al utilizar la clase verify descripta en [JPF01]. Con ayuda de esta clase, **JPF** permite instrumentar el código para producir valores no determinísticos acordes al contexto del modelo, como por ejemplo enteros, reales, boléanos, etc.

En cambio, la segunda condición está potencialmente presente en cualquier modelo multithread. Para verificar si la misma ocurre, **JPF** mantiene un contexto de la ejecución el cual es accedido por la instrucción a ejecutar a continuación. Si la instrucción determina que existe una posible bifurcación (utilizando esta información de contexto), se notifica a la **VM** que genera un nuevo estado.

En la Figura 4.3 (extraída de la documentación de JPF) se presentan las instrucciones de bajo nivel relacionadas con la generación de nuevos estados.

Las mismas están agrupadas por tipo de instrucción y según como puedan afectar al contexto de la ejecución. Los tipos son: *field insn*, para acceso a propiedades de un objeto; *sync insn*, acceso a elementos sincronizados; *invoke insn*, invocación de métodos; etc. Las formas en que cada una afecta al contexto son: *other runnable threads*, que afectan la ejecución de otros threads y *shared objects*, que acceden a memoria compartida.

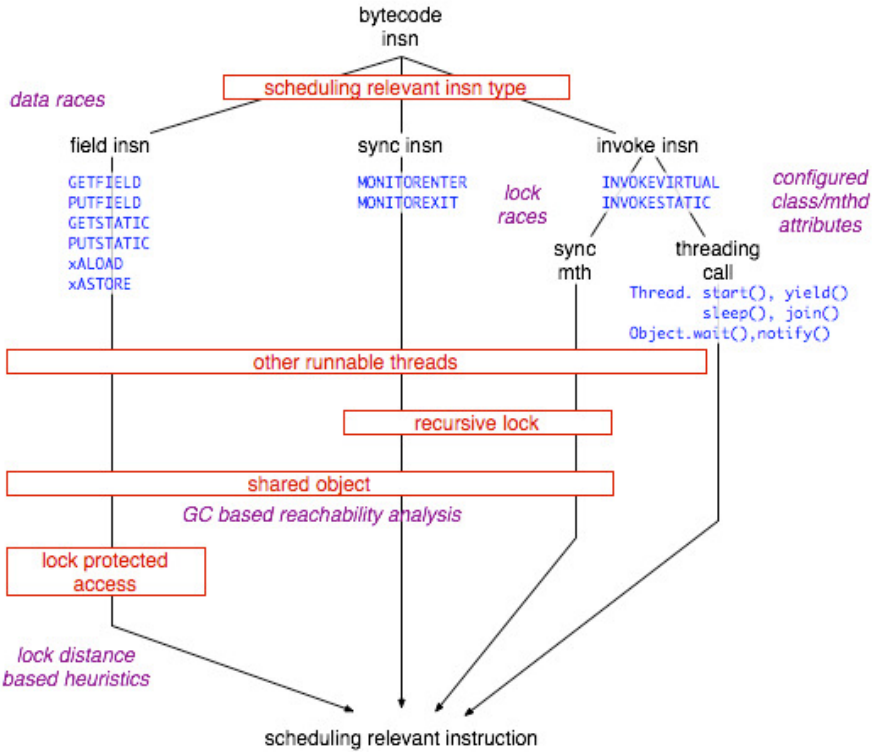

Figura 4.3: Esquema de instrucciones generadoras de estados en JPF

4.6 Definición de propiedades

JPF provee tres mecanismos para especificar las propiedades a verificar sobre el modelo: Aserciones, *Listeners* y *Properties*.

Las **aserciones** son instrucciones particulares provistas por la mayoría de los lenguajes. Se utilizan para instrumentar el código del modelo definiendo predicados que deben ser verdaderos en el momento de su evaluación. Por otro lado, las ***Properties*** y los ***Listeners*** son mecanismos más complejos, compuestos de una jerarquía de interfaces y clases que les permiten mantener toda la información necesaria para la verificación de la propiedad.

A pesar de ser mecanismos diferentes, los ***Properties*** y ***Listeners*** tienen una manera similar de establecer la comunicación entre ellos y el resto del entorno **JPF**, más precisamente con ***Search*** y ***VM***. Utilizan un concepto de listas encadenadas (*Multicasters*) donde los nodos son las propiedades definidas para la verificación y la comunicación (como el *check* de propiedades, notificación de los eventos ocurridos, etc.) entre **JPF** y las propiedades se propaga por toda la lista de un nodo al otro.

4.6.1 Properties

Las *Properties* provistas por **JPF** son en realidad implementaciones de la interfaz *Property*.

```
public interface Property extends Printable {
```

```
   boolean check (Search search, JVM vm);

   String getErrorMessage ();
}
```

Figura 4.4: Interfaz Property

Las propiedades definidas de esta manera se verifican en las transiciones entre estados mediante la ejecución del método *check()* (luego de completada la ejecución del método *forward()* dentro de **Search)**. Cuando este método retorna el valor falso, indica que la propiedad fue violada. En este caso, la verificación se da por finalizada y se muestra al usuario un log de todas las instrucciones ejecutadas que llevaron al error.

Por defecto, JPF brinda la posibilidad de verificar las siguientes: *NotDeadlockedProperty, que* verifica la ausencia de deadlocks; *NoUncaughtExceptionsProperty, que* verifica que todas las excepciones se capturan en tiempo de ejecución; e *IsEndStateProperty, que* verifica que se haya alcanzado el estado final.

Para definir una propiedad, el usuario necesita definir una clase que implemente la interfaz *Property* y el método *check*.

4.6.2 Listeners

Los *Listeners* son observadores de la ejecución del modelo e implementan el patrón de diseño Observer ([GHJV94]). Básicamente, permiten suscribirse a ciertos eventos del objeto **Search**, como ser una transición de un estado a otro; o del objeto **VM**, como son la ejecución de una instrucción, la ejecución del *Garbage Collector* o la creación de una variable.

La idea principal de los *Listeners* es proveer funcionalidad adicional en forma de extensiones realizadas al JPF, sin tener la necesidad de modificar las implementaciones existentes de **Search** o la **VM**.

Desde el punto de vista estrictamente técnico, **JPF** provee dos interfaces para implementar *listeners*: *VMListener* y *SearchListener*.

Como puede verse en la Figura 4.5, la interfaz *VMListener* permite suscribirse a eventos relacionados con la ejecución de la **VM**. Por ejemplo, implementando el método *executeInstruction* se puede observar la ejecución de todos los bytecodes del modelo. Esto permite, por ejemplo, construir logs o trazas para seguir la ejecución.

```
public interface VMListener extends JPFListener {
   void executeInstruction (JVM vm);
   void instructionExecuted (JVM vm);
   void threadStarted (JVM vm);
   void threadBlocked (JVM vm);
   void threadWaiting (JVM vm);
   void threadNotified (JVM vm);
   void threadInterrupted (JVM vm);
   void threadTerminated (JVM vm);
   void threadScheduled (JVM vm);
   void classLoaded (JVM vm);
   void objectCreated (JVM vm);
   void objectReleased (JVM vm);
```

```
   void objectLocked (JVM vm);
   void objectUnlocked (JVM vm);
   void objectWait (JVM vm);
   void objectNotify (JVM vm);
   void objectNotifyAll (JVM vm);
   void gcBegin (JVM vm);
   void gcEnd (JVM vm);
   void exceptionThrown (JVM vm);
   void choiceGeneratorSet (JVM vm);
   void choiceGeneratorAdvanced (JVM vm);
   void choiceGeneratorProcessed (JVM vm);
}
```

Figura 4.5: Interfaz VMListener

Por otro lado, en la Figura 4.6, la interfaz *SearchListener* establece el protocolo para la suscripción a eventos relacionados con la búsqueda de estados. Por ejemplo, **JPF** provee una funcionalidad, que utilizando esta interfaz, construye y grafica el espacio de estados (*StateSpaceDot.java*).

```
public interface SearchListener extends JPFListener {
   void stateAdvanced (Search search);
   void stateProcessed (Search search);
   void stateBacktracked (Search search);
   void stateRestored (Search search);
   void propertyViolated (Search search);
   void searchStarted (Search search);
   void searchConstraintHit (Search search);
   void searchFinished (Search search);
}
```

Figura 4.6: Interfaz SearchListener

Utilizando los *Listeners*, el usuario tiene acceso a la información tanto del **Search** como de la **VM**, ya que estos objetos son pasados como parámetro en todos los métodos de la interfaz.

Además de las dos interfaces antes mencionadas, existe una clase, *PropertyListenerAdapter*, que implementa los métodos tanto de *Listener* como de *Property*.

El siguiente, es un ejemplo simple donde se muestra como construir una propiedad que verifica si la cantidad de invocaciones a métodos (a través de la ejecución de instrucciones *Invoke*) de una clase supera un número dado, y en caso afirmativo invalidar la verificación.

```
public class TestProperty extends PropertyListenerAdapter {
   private int count = 0;

   public boolean check (Search search, JVM vm) {
      return (count < 3);
   }

   public void executeInstruction (JVM vm) {
      Instruction li = vm.getLastInstruction();
      if (li instanceof InvokeInstruction) {
         count++;
      }
   }
}
```

Figura 4.7: Ejemplo de PropertyListenerAdapter

4.7 Discusión

Hemos presentado las características principales de **JPF**. No obstante, a continuación se detallará el análisis sobre algunos puntos de interés para el resto del trabajo.

- **JPF** es un emprendimiento Open Source, lo cual implica que se puede obtener, utilizar y modificar el código de la herramienta (siempre dentro de los límites del licenciamiento). Esto facilita y amplia las posibilidades de cualquier trabajo de investigación relacionado con el tema.
- Las posibilidades de extensión que presenta el framework son realmente muy variadas. En el contexto del presente trabajo, nos concentramos principalmente en buscar la forma de extender los *Listeners* para permitir la verificación de propiedades basadas en patrones de eventos y modificar la búsqueda para que esta tenga en cuenta en sus decisiones el estado del *Listener.* No obstante, se podría haber avanzado mucho más en este sentido.
- Hay una diferencia sustancial entre el modelo que el usuario define y el modelo que **JPF** verifica. El primero, es un conjunto de clases escritas en Java, mientras que el segundo, es un conjunto de bytecodes, producto de la precompilación del modelo. Una instrucción escrita en Java, puede ser representada por más de un bytecode. Esto implica que hay una diferencia de nivel entre el modelo y las propiedades, ya que cuando se define el primero, se hace en términos de objetos y estructuras de control especificadas en un lenguaje de alto nivel (Java), mientras que cuando se especifican las propiedades se lo hace en términos de bytecodes.
- Puntualmente, con respecto al algoritmo de búsqueda actual, puede ser modificado implementando una clase de búsqueda propia, lo cual a priori, no implica modificar las ya provistas por **JPF**.
- Actualmente **JPF** no permite realizar verificaciones sobre programas que incluyan código nativo del sistema operativo. Esto se debe a que dichas instrucciones no producen bytecodes y, por lo tanto, **JPF** no puede administrar el estado de estos fragmentos de código. Si bien esto representa una limitación para los modelos que se pueden verificar, es posible definir wrappers para simular a los componentes o clases que tengan código nativo.
- **JPF** es lo suficientemente flexible, como para permitir al *Listener* realizar acciones que influencien directamente la forma en que se realiza la búsqueda. Por ejemplo, se pueden agregar las clases apropiadas para la creación dinámica de nuevos estados.
- El hecho de que **JPF** haya sido diseñado para verificar únicamente código Java, permite hacer ciertas presunciones sobre el dominio del problema a modelar. En particular, la decisión de que no todas las instrucciones generen nuevos estados implica que habrá opciones de interleaving que no se estarán evaluando durante

la verificación. Lo cual, por un lado disminuye el espacio de estados a recorrer, pero por el otro quita flexibilidad para definir modelos mas abstractos.

- Relacionado al punto anterior, la verificación de las *Properties* no se realiza inmediatamente después de la ejecución de cada bytecode del modelo, sino que se realiza en cada transición entre estados. Esto implica que una vez que se produce una violación a la propiedad, el modelo puede continuar ejecutándose hasta la siguiente transición, provocando un retraso en la detección de la violación.

Capítulo 5

Verificación de patrones de eventos con JPF

5.1 Introducción

Una gran parte del trabajo desarrollado para la presente tesis ha tenido que ver con adaptar JPF para soportar la verificación de propiedades especificadas en términos de patrones de eventos.

Esta adaptación fue hecha modificando lo menos posible el núcleo de JPF e intentando ofrecer al usuario una forma relativamente sencilla de usarla. De esta manera, se desarrolló un framework que permite:

- Verificar propiedades especificadas en términos de patrones de eventos. Proveyendo una implementación de los autómatas y eventos que coincida con lo descrito en el capítulo 3.
- Limitar la exploración del espacio de estados de la verificación utilizando los escenarios de control también descritos en el capítulo 3.
- Definir propiedades Typestate (ver 5.5) sobre los objetos que participan del modelo.
- Definir propiedades Typestate (ver 5.6) sobre clases abstractas

En el presente capítulo se describirá en detalle cada uno de estos objetivos y se presentará la solución a cada una de las dificultades que fueron surgiendo a lo largo del desarrollo. Y para finalizar, se presentarán las conclusiones a las que llegamos a partir del trabajo con JPF.

5.2 Eventos y patrones de eventos

En el capítulo 3, se presentaron algunas definiciones tanto de Evento como de Patrón de Evento. Para poder llevar a cabo el presente trabajo, estas definiciones abstractas, deben tener su contraparte en JPF. Dicho de otra manera, es necesario definir tanto para los patrones como para los eventos qué son y cuál es la estructura de clases que los representa. Adicionalmente, es imprescindible tener alguna forma de reconocer la ocurrencia de los eventos durante la verificación del modelo.

Si se asocia un tipo de instrucción a un tipo de evento es factible identificar la ocurrencia de eventos detectando la ejecución de estas instrucciones (ver sección 4.6). El problema técnico es que antes de ejecutar la verificación, JPF produce los bytecodes que representan a las instrucciones del modelo, esto implica que la asociación antes mencionada debe ser hecha sobre los bytecodes.

Encontramos de particular importancia las invocaciones a métodos, ya que en un entorno de objetos, es razonable suponer que una parte importante de la lógica se implementa en el pasaje de mensajes (o ejecución de métodos) entre los objetos. De esta manera, la ocurrencia de un evento se ha implementado como la ejecución de un bytecode de tipo *INVOKE* (por ejemplo: *INVOKEVIRTUAL*, *INVOKESTATIC*).

Para simplificar la implementación dejamos afuera de esta versión la detección de ocurrencias de otro tipo de eventos distintos de INVOKE. No obstante, creemos que esto produce una pérdida de generalidad ya que el framework es fácilmente extensible.

A nivel de diseño, la representación de los eventos es simple. La información relevante para la verificación de patrones se compone del nombre del método, el nombre de la clase o instancia sobre la que se ejecuta (en caso de que sea relevante) y una etiqueta para poder referenciar su ocurrencia.

Por lo tanto, para cada evento se deberá definir, el nombre completo del método que interviene en la invocación y una etiqueta para identificarlo[2].

Es importante aclarar que los eventos se pueden mencionar desde dos puntos de vista: la definición del mismo (por ejemplo: evento de tipo *INVOKEVIRTUAL*, asociado al método *open*, de la clase *Canal*) y la ejecución de dicho bytecode en cierto instante de la verificación. Lógicamente, durante una verificación podría tener lugar la ocurrencia de dos eventos asociados a la misma definición (siguiendo el ejemplo, dos invocaciones al método *Canal.open*). En este caso, existirán tantas instancias de dicho evento, como ocurrencias se detecten durante el proceso de verificación de la propiedad. Esta distinción se ve reflejada en las dos clases que dan soporte a la generación de eventos:

- **EventBuilder**: Se construye al iniciar la verificación y es la encargada de interpretar y mantener la definición de los eventos ingresada como parámetro por el usuario.
- **Evento**: durante la verificación, se analizará cada bytecode ejecutado para determinar si corresponde a un evento dentro de la definición del **EventBuilder**. En caso positivo, se generará una instancia de la clase **Evento** con la información asociada al mismo.

Una vez definidos los eventos, los patrones de eventos son más simples. Definimos la clase *AutomataVerificacion*, que es una implementación sencilla de Autómata Finito Determinístico según lo descrito en el capítulo 3. Durante su ejecución, este autómata consume los distintos eventos definidos para el modelo y en base a estos, actualiza su estado interno.

Es importante destacar que esta implementación supone que los requerimientos (afds) se especifiquen como antipropiedades. Dicho en otras palabras, llegar a un estado final del autómata implica que la propiedad se ha violado.

[2] Más adelante, en el apéndice, se explicarán en mayor detalle los aspectos de implementación.

5.3 Verificación de patrones de eventos

Ya hemos definido a nivel general la forma en que se representan los Eventos y las propiedades. Ahora el problema reside en definir la forma en que estos se integrarán con el JPF.

El mecanismo que empleamos para detectar la ocurrencia de los eventos, es el *Listener*, que cumple justamente la función de observador de la ejecución del modelo. De esta manera, utilizando el método *ExecuteInstruction* (ver sección 4.6), se pueden capturar los eventos (ejecuciones de bytecodes de tipo INVOKE) e informarlos a nuestro autómata.

Este enfoque tiene un problema, que es que ni el *Listener* ni el autómata forman parte del modelo, y por ende, el algoritmo de búsqueda no tiene en cuenta sus estados al tomar la decisión de hacer backtrack o forward. Esto puede provocar que durante la verificación, se llegue a un estado del modelo ya visitado, pero con un estado del autómata distinto, y este camino se estaría podando cuando debería volver a recorrerse.

Hay tres alternativas para solucionar este problema:

- Modificar físicamente el árbol, agregando nodos (estados) manualmente. Para esto, se puede utilizar el Verify, o los *ChoiceGenerators*.
- Modificar la estructura interna de los estados, alternado el algoritmo de Hashing para que incluya en los cálculos al Listener y al autómata.
- Modificar el algoritmo de búsqueda para que al evaluar la acción a tomar (backtrack o forward) tenga en cuenta el estado del *Listener* y del autómata.

Las primeras alternativas son muy complejas y riesgosas, con lo cual fueron descartadas. De esta manera, se optó por la tercera, que es más simple, menos riesgosa y sólo requiere modificar la clase que implementa la búsqueda y exploración del espacio de estados. No se altera físicamente la estructura interna de los estados de la verificación ni se modifica el espacio; simplemente se interpreta de forma diferente

La forma en que se llevó a cabo la modificación, fue implementando una nueva clase que hereda de la existente en JPF (*DFSearch*), pero sobrescribiendo el método *search*.

Esta modificación no altera la cantidad física de estados del espacio, pero incrementa la cantidad de veces que un mismo estado debe ser procesado durante la verificación. Es decir, la cantidad total de nodos del árbol que se deben recorrer con el cambio del algoritmo es mayor o igual que con el algoritmo de búsqueda original (*DFSearch*), ya que un estado y el subárbol resultante puede volver a ejecutarse, dependiendo del prefijo de patrón de eventos que está en ejecución en la rama actual.

A continuación se presenta un ejemplo del funcionamiento del algoritmo de búsqueda modificado:

Ejemplo 2: Adaptación del algoritmo de Búsqueda de estados

En la Figura 5.1 se presenta un modelo escrito en JPF en el cual existen tres threads que realizan operaciones sobre un objeto de tipo Canal. Los eventos relevantes para la verificación son OPEN, WRITE y CLOSE.

```
package tesis.Ejemplo_EscControl;

public class Modelo {
   public static void main(String[] args) {
      Canal c = new Canal();

      Hilo o;
      Thread t;

      for (int i=0;i<3;i++) {
         o = new Hilo(c);
         t = new Thread(o);
         t.start();
      }
   }
}
```

```
class Canal {
   private int opens = 0;
   private int writes = 0;

   public void open() {
      opens++;
   }
   public void close() {
      opens--;
   }
   public void write() {
      writes++;
   }
}

class Hilo implements Runnable {
   private Canal c;

   Hilo (Canal canal) {
      c = canal;
   }

   public void run() {
      c.open();
      c.write();
      c.close();
   }
}
```

Figura 5.1: Modelo de Ejemplo de un Canal compartido por múltiples Threads

En la Figura 5.2 se presenta la propiedad que se desea verificar: “NO se puedan realizar tres eventos WRITE en toda la ejecución (sin importar los eventos intermedios) del programa”.

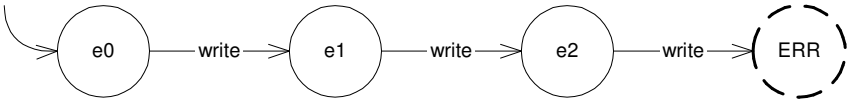

Figura 5.2: AFD para la verificación de 3 WRITEs

Una aclaración con respecto a los AFDs representados: no se incluyen explícitamente las transiciones reflexivas (por eventos tales como * - {write} en 1.2) debido a que la lógica predeterminada de estos AFDs, es la de mantener el mismo estado en el caso de que se detecte un evento para el cual no existe una transición definida.

En la Figura 5.3. se muestra el árbol de estados resultante de ejecutar la verificación del modelo con JPF y la búsqueda DFS original. Es importante notar, que este árbol es una versión simplificada, se muestran sólo los estados relevantes para poder interpretar la lógica de backtracking de JPF.

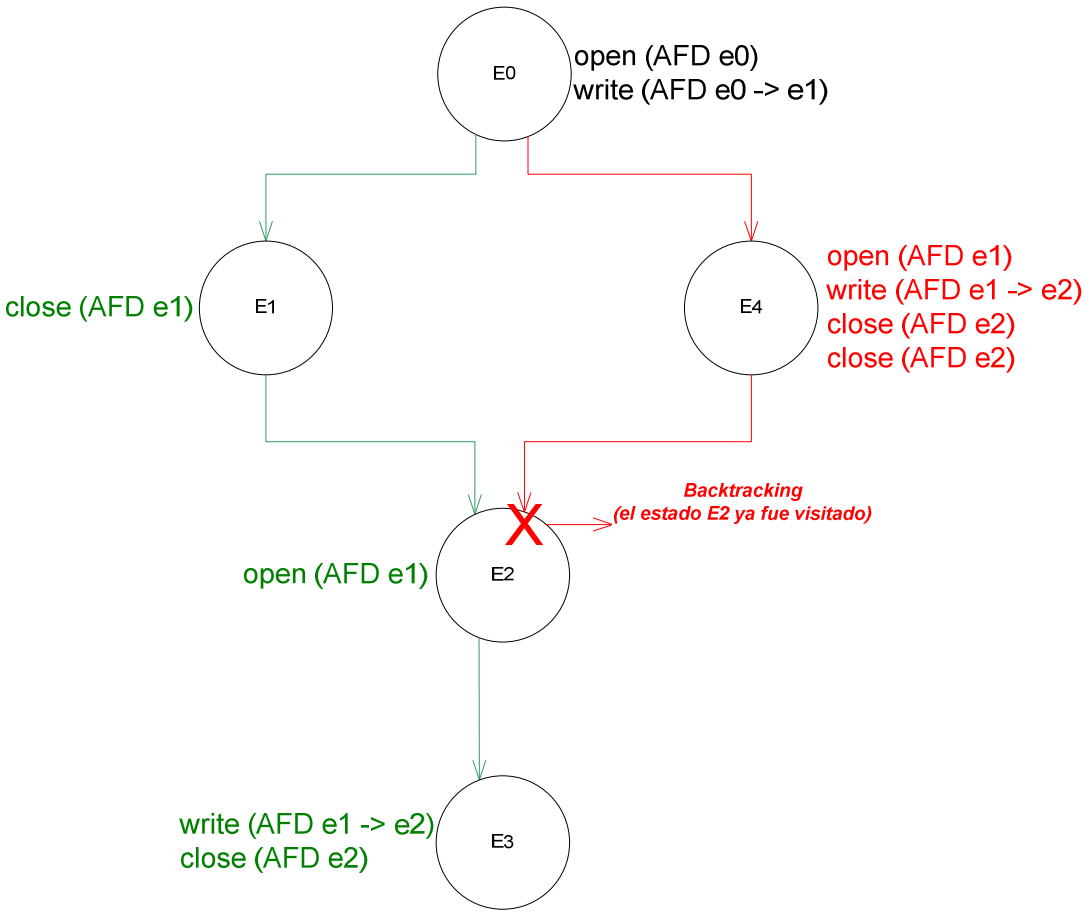

Figura 5.3: Árbol de estados, sin contemplar estado de la property (AFD)

Cuando la verificación llega al estado E2 por segunda vez, detecta que es un estado ya visitado y poda la búsqueda del subárbol resultante, sin importar el estado del AFD y que con la nueva exploración de la rama E2, E3 se podría detectar una violación a la propiedad definida (porque se produciría un 3er write).

Una vez adaptado el algoritmo de búsqueda, y siguiendo con el ejemplo anterior, se puede observar esta situación, con la siguiente figura:

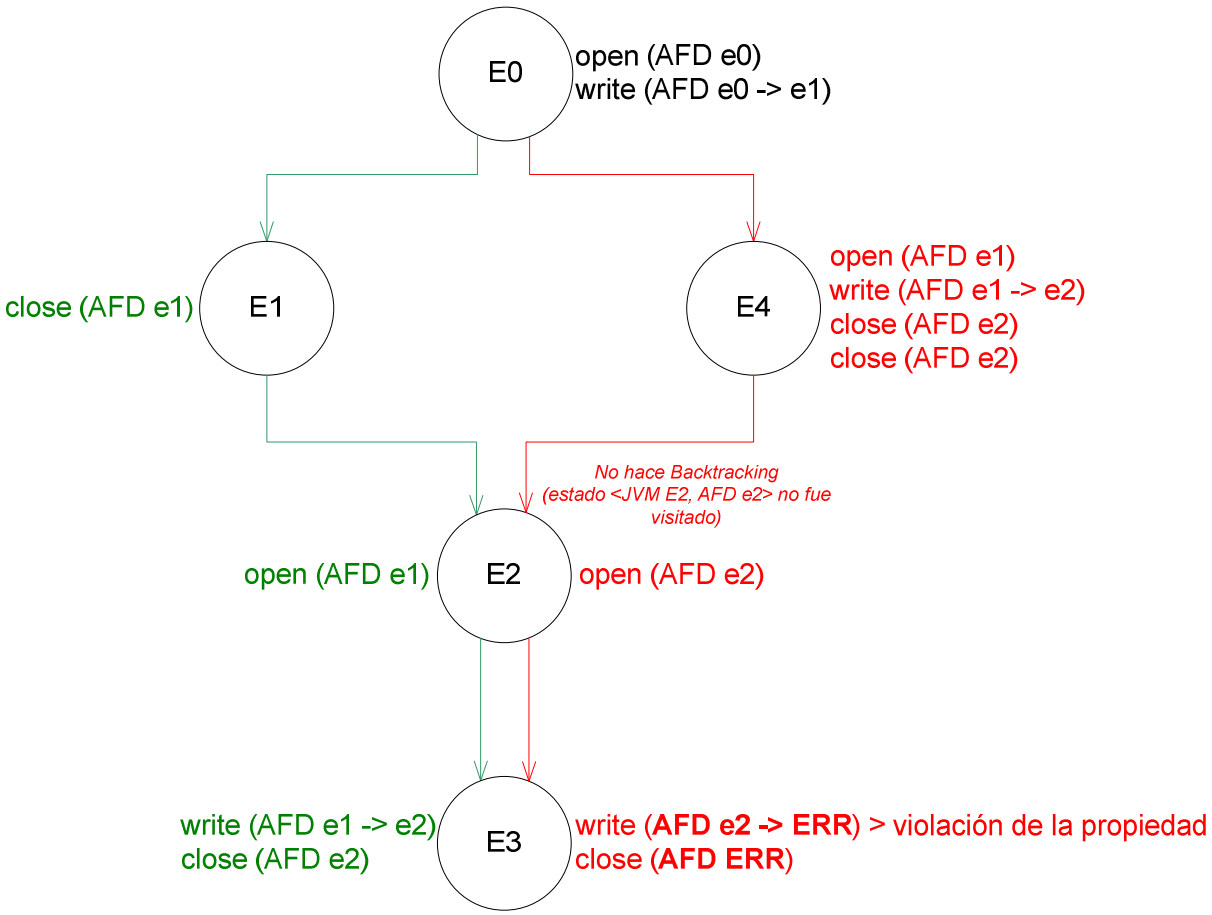

Figura 5.4: Árbol de estados, contemplando estado de la property (AFD)

En el momento de llegar nuevamente al estado E2 (desde el estado E4), el algoritmo de búsqueda detecta que, si bien es un estado ya visitado por sí sólo, contemplando la tupla <VM E2, AFD e2> vuelve a explorar el subárbol y deriva en la detección de la violación de la propiedad, ya que el patrón de eventos ocurridos (2 write's) en el camino actual es diferente al del primer camino de ejecución (1 write). Más adelante se discutirán aspectos de performance y el overhead que esta modificación genera en el algoritmo.

Es importante destacar, que se puede garantizar la finalización del ciclo de búsqueda, ya que el AFD de la propiedad tiene una cantidad finita de estados, y siempre que se llegue a un estado <VM EX, AFD eX>, se almacena en una colección que luego es recorrida para determinar el siguiente paso de la búsqueda (backtrack o forward).

5.4 Escenarios de control

Los escenarios de control brindan la posibilidad de limitar las trazas de los eventos generadas a partir de la verificación de un modelo. Aunque con enfoques distintos, tanto el preámbulo como el contexto de ejecución (mencionados en el capítulo 3) permiten optimizar la generación y exploración de los estados durante la verificación.

5.4.1 Preámbulo

El Preámbulo permite al usuario, especificar un determinado patrón de eventos que debe producirse desde el comienzo de la generación de eventos, para que se pueda dar inicio a la verificación de la propiedad en sí misma.

Una vez definido el preámbulo, la herramienta es capaz de realizar un matching de los eventos, que van ocurriendo al comenzar la ejecución del modelo, contra la definición del Preámbulo. Esta forma de realizar el matching, si bien es similar a la verificación de patrones de eventos de una propiedad, difiere en un sentido importante: al detectarse un evento que no se corresponde con el siguiente esperado en el Preámbulo, automáticamente esa rama de la ejecución es descartada (backtracking) para poder seleccionar otro camino y buscar la ocurrencia correcta de los eventos del Preámbulo.

Para poder identificar el momento en el cual hay que realizar backtracking, existe un estado especial denominado INVALIDO al cual avanza el Preámbulo una vez que se detecta un evento no esperado.

A continuación se describen los diferentes estados en los que se puede encontrar el preámbulo durante la verificación del Modelo:

1. **En ejecución**: indica que hasta el momento no se produjo ningún evento que invalide el preámbulo. Se continúa con la ejecución y la verificación del mismo.
2. **Satisfecho**: indica que se cumplieron todos los eventos en el orden establecido por el preámbulo, se puede comenzar con la verificación de la propiedad.
3. **Violado**: indica que se produjo un evento no esperado. Esto invalida la rama actual de ejecución. En este caso, será necesario realizar backtracking del estado actual de la JVM y continuar por otro camino.

5.4.2 Contexto

El Contexto tiene como base un patrón de eventos, el cual no debe invalidarse en ningún momento de la verificación para poder continuar con la ejecución del camino actual.

Se representa con una estructura muy similar a los Autómatas Finitos Determinísticos. La diferencia es que en el Contexto, el conjunto de estados finales (o de aceptación) siempre es vacío, ya que no tiene un estado de Aceptación por sí mismo. Al igual que el Preámbulo, existe un estado denominado INVÁLIDO que es alcanzado cuando se detecta una secuencia o patrón de eventos que lo invalida.

Comparando el Contexto con el Preámbulo, se pueden observar las siguientes diferencias:

- El Preámbulo se verifica sólo al comienzo de la ejecución y hasta que sea satisfecho (momento a partir del cual, comienza la verificación de la propiedad). El Contexto se verifica durante toda la ejecución en paralelo a la propiedad.

- En la modalidad del Preámbulo se busca que se alcance el estado final del mismo (satisfecho) para poder comenzar con la verificación. En el Contexto se busca que no se produzca ninguna secuencia de eventos que conduzca a un estado final que invalide el camino de la verificación actual.
- El Preámbulo simboliza, a nivel abstracto, una lista de eventos; en general, representa un prefijo admisible que deben cumplir todas las trazas de eventos a verificar. Análogamente, el Contexto es el patrón de eventos que compone el escenario de control que limita las trazas válidas durante la verificación.

Ejemplo 3: Restricción de las ramas, mediante el uso de Contexto de Verificación

Siguiendo con el Ejemplo 2, supongamos que queremos verificar una propiedad en la que no se realizan 3 WRITEs sin un CLOSE entre medio de ellos. En la siguiente figura se muestra el autómata que la representa:

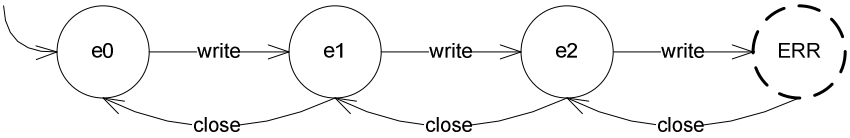

Figura 5.5: AFD para la verificación de 3 WRITEs sin CLOSEs intermedios

Ejecutando JPF para verificar esta propiedad, se observa que se recorren 1388 nodos del árbol de estados de la JVM hasta encontrar una violación a la propiedad. Ahora supongamos que se restringe la verificación con el siguiente contexto: no se admite ningún evento CLOSE. Este contexto se representa con el autómata de la siguiente figura:

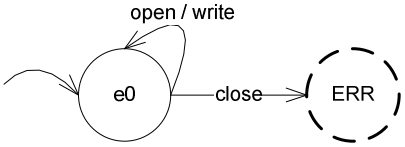

Figura 5.6: Contexto para restringir CLOSE

En este caso, al ejecutar la misma verificación, pero agregando el Contexto como Escenario de Control de la misma, se observa que se recorren sólo 23 estados contra los 1388 de la ejecución anterior.

5.5 Typestate properties

Las propiedades con las que trabajamos hasta el momento no permiten distinguir la ejecución de un método entre distintas instancias de una misma clase. Dicho en otras palabras, si dos objetos O_1 y O_2 de la misma clase C ejecutan el mismo método m, el framework supondría que son dos ocurrencias del mismo evento.

Las propiedades Typestate permiten verificar las propiedades sobre instancias de clases determinadas. Al igual que el caso de las propiedades estándar, su representación es un Autómata Finito Determinístico cuyos estados se corresponden con los eventos que va disparando el objeto referenciado por la propiedad. Alcanza su estado final (o de error) cuando se cumple el patrón.

Los eventos relevantes para una *Typestate Property*, dependerán del tipo de objeto para el cual fue definida. Por ejemplo, se cuenta con un modelo con 2 entidades relevantes para verificar: Canal y NodoRed. Canal cuenta con los siguientes métodos OPEN, WRITE y CLOSE, mientras que NodoRed con ON, OFF, ACTIVO.

Supongamos que para un objeto de tipo Canal, es posible que sean relevantes sólo los eventos (generados por los métodos) OPEN y CLOSE, mientras que para un objeto de tipo NodoRed interesen sólo ON y OFF.

Es preciso resaltar un punto importante que es el concepto de Tipo (type) en Java. Un Tipo en Java está representado de diversas formas (ej: Class, Interface). En el presente trabajo acotaremos la definición de TypeState Properties a las Clases, las cuales definen la base para la creación de objetos (instancias generadas en tiempo de ejecución), especificando el comportamiento y la esencia de cada uno.

Al asociar una propiedad Typestate a una clase, se verificará sobre todos los objetos de dicha clase. De esta forma, la misma propiedad deberá ser instanciada tantas veces como objetos de la clase se construyan durante la verificación del modelo.

Para ejemplificar el escenario: supongamos que existe un modelo en el cual se van creando varios objetos de tipo Canal. Estos objetos tienen la particularidad de, dado un instante en la ejecución, encontrarse en dos posibles estados: *opened* o *closed.* Los eventos posibles sobre un objeto de tipo Canal son: OPEN, WRITE, CLOSE.

Se quiere verificar que para cada instancia se cumpla cierto patrón de eventos. Definamos este patrón de eventos como: 1 OPEN, seguido de 1 ó más WRITE, seguido de 1 CLOSE. Esta propiedad, se puede representar con el siguiente autómata:

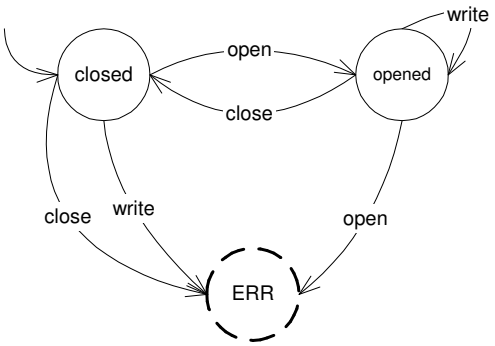

Figura 5.7: AFD para la verificación del correcto uso de un Canal

Si no se contara con la posibilidad de definir propiedades *Typestate*, no se podría verificar el patrón de eventos para cada Canal en particular; sólo se podría generar una propiedad de manera global. En este caso, sólo sería posible verificar el patrón de eventos para un modelo que genere un Canal a la vez, chequeando los eventos (OPEN, WRITE y CLOSE) globalmente.

Con la modalidad *Typestate*, se puede definir el autómata asociado para la clase Canal y, el framework se encargará de manejar un AFD para cada objeto de tipo Canal que se

genere. Para poder administrar los objetos que se van creando y sus instancias de AFD asociadas, se deberá monitorear la creación y destrucción de objetos para un determinado tipo (ver detalle de implementación más abajo).

A continuación veremos dos ejemplos de la propiedad de la figura anterior en los cuales se verificará bajo ambas modalidades el correcto uso de un canal: Global, *TypeState*. La propiedad se verifica sobre un modelo que genera varias instancias de tipo Canal. Para simplificar la notación llamaremos "AG" AFD Global y, "Ax" (siendo x el número de instancia de AFD) a la variante Typestate.

El modelo Java correspondiente es el siguiente:

```
package
   tesis.Ejemplo_TypestatevsGlobalProperty;

class Canal {
   private int opens = 0;
   private int writes = 0;

   public void open() {
      opens++;
   }

   public void close() {
      opens--;
   }

   public void write() {
      writes++;
   }
}
```

```
public class Modelo {
   public static void main(String[]
args) {
      Canal c1 = new Canal();
      Canal c2 = new Canal();

      c1.open();
      c1.write();
      c1.close();

      c1.open();
      c1.write();

      c2.open();
      c2.write();
      c2.close();

      c1.close();
   }
}
```

Figura 5.8: Modelo de Ejemplo para comparar GlobalProperty vs. TypestateProperty

En la siguiente figura podemos comprobar la verificación en dos formas diferentes: utilizando Global Property y luego utilizando el mismo AFD en modo Typestate:

Global Property		Typestate Property	
Instrucción Ejecutada	**Transición AFDs**	**Instrucción Ejecutada**	**Transición AFDs**
`Canal c1 = new Canal();`		`Canal c1 = new Canal();`	`new A1`
`Canal c2 = new Canal();`		`Canal c2 = new Canal();`	`new A2`
`c1.open();`	`AG closed -> opened`	`c1.open();`	`A1 closed -> opened`
`c1.write();`	`AG opened -> opened`	`c1.write();`	`A1 opened -> opened`
`c1.close();`	`AG opened -> closed`	`c1.close();`	`A1 opened -> closed`
`c1.open();`	`AG closed -> opened`	`c1.open();`	`A1 closed -> opened`
`c1.write();`	`AG opened -> opened`	`c1.write();`	`A1 opened -> opened`
`c2.open();`	**`AG opened -> ERROR`**	`c2.open();`	`A2 closed -> opened`
		`c2.write();`	`A2 opened -> opened`
		`c2.close();`	`A2 opened -> closed`
		`c1.close();`	`A1 opened -> closed`

Figura 5.9: Global Property vs. Typestate Property

En la figura anterior se puede observar que utilizando la propiedad en forma global se detecta, de manera incorrecta, una violación por uso indebido de un objeto de tipo Canal. Esto se debe a que el evento OPEN en el cual se detecta se produce sobre una instancia de Canal que sí se encontraba apta para ese evento. Utilizando el mismo AFD pero en modalidad *Typestate Property*, se verifica que el sistema se comporta según lo esperado.

Análogamente, se podría presentar otro modelo en el cual se observe que utilizando la propiedad definida Global no se detecte (incorrectamente) una violación, pero sí en la modalidad *Typestate*. El modelo es el mismo que el anterior, excepto que el método *main* es el siguiente:

```
public static void main(String[] args) {
   Canal c1 = new Canal();
   Canal c2 = new Canal();

   c1.open();
   c2.write();
   c2.close();
}
```

Figura 5.10: Modelo de Ejemplo para comparar GlobalProperty vs. TypeStateProperty

En la siguiente figura, podemos comprobar la verificación sobre el nuevo Modelo comparando nuevamente Global vs. *TypeState*:

Global Property		Type State Property	
Instrucción Ejecutada	**Transición AFDs**	**Instrucción Ejecutada**	**Transición AFDs**
`Canal c1 = new Canal();`		`Canal c1 = new Canal();`	`new A1`
`Canal c2 = new Canal();`		`Canal c2 = new Canal();`	`new A2`
`c1.open();`	`AG closed -> opened`	`c1.open();`	`A1 closed -> opened`
`c2.write();`	`AG opened -> opened`	`c2.write();`	**`A2 closed -> ERROR`**
`c2.close();`	`AG opened -> closed`	`c2.close();`	

Figura 5.11: Global Property vs. Typestate Property (2)

En la figura anterior se verifica que, utilizando la propiedad en modo Global no se detecta una violación por uso indebido de Canal. Esto se debe a que ocurrieron eventos WRITE y CLOSE en un canal sin que haya ocurrido un OPEN previamente sobre esa instancia (c2). Paralelamente, se puede comprobar que esta violación a la propiedad sí se detecta correctamente utilizando el modo *Typestate*.

Uno de los inconvenientes a afrontar dentro de JPF para implementar la funcionalidad de *Typestate Properties* está relacionado con la necesidad de poder identificar cada instancia del tipo en cuestión (Canal, en el ejemplo anterior). Esto se logró mediante la modificación de un componente dentro del kernel del JPF, que permita obtener información relacionada con la identidad del objeto sobre el cual ocurre un determinado evento (ver Apéndice para más detalles).

Por otra parte, fue necesario extender el algoritmo del verificador de patrones de eventos para que, cada vez que se crea un objeto de un Tipo para el cual hay definidas *Typestate Properties*, genere la instancia del AFD correspondiente.

Este último punto conlleva a la necesidad de modificar la lógica del framework, para poder administrar no sólo una propiedad, sino un conjunto de propiedades *Typestate*. Luego de esta modificación, el verificador de patrones está capacitado para administrar múltiples propiedades Typestate, así como también operar en paralelo con una propiedad Global.

5.6 Typestate properties sobre clases abstractas

En la sección anterior, se explicaron las características que tienen las *Typestate Property*. Si bien a nivel de implementación se logró soportar (dentro del framework extendido de JPF) la funcionalidad para verificar este tipo de propiedades, existen algunas consideraciones relacionadas con el tipo de modelos que se verifican con JPF. Los modelos son, específicamente, programas Java. Java se caracteriza como un lenguaje orientado objetos, es decir que provee, entre otras cosas, los conceptos de abstracción, polimorfismo, herencia, etc.

Sin entrar en detalles específicos sobre diseño y programación orientados a objetos, podemos mencionar que una buena práctica, en el momento de diseñar un modelo o aplicación, es pensar en términos de jerarquías de clases aislando comportamiento común a diferentes Tipos. Es por este motivo que surge la necesidad de proveer mecanismos para poder definir *Typestate Properties* sobre los diferentes miembros en una jerarquía de Clase, es decir que la verificación no esté limitada al uso de propiedades exclusivamente relacionadas con tipos finales (hojas de la jerarquía de clases) y/o concretos.

De esta forma, se podría definir una única propiedad *Typestate* que se aplicará, durante la verificación para diferentes tipos.

Ejemplo 4: Jerarquía de clases de Canal

Supongamos que se ha diseñado una jerarquía de clases como la siguiente:

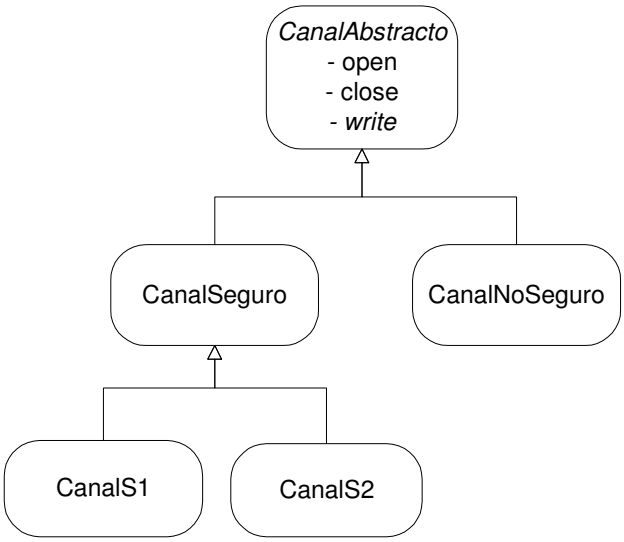

Figura 5.12: Jerarquía de clases de Canal

Aquí puede verse que existe una clase abstracta (*CanalAbstracto*) y el resto son simples especializaciones. Si se quisiera verificar una propiedad de patrón de eventos sobre las características de un canal, es decir, el comportamiento genérico de un canal y no sobre su especialización (por ejemplo la propiedad de la Figura 5.7), entonces deberíamos definir una única *Typestate Property* asociada a la clase *CanalAbstracto*. De esta forma, cada vez que en el modelo se construya un objeto de cualquiera de los subtipos (*CanalSeguro*, *CanalS1*, *CanalS2* y *CanalNoSeguro*) se generará una instancia del AFD correspondiente a la propiedad TypeState para verificar el patrón de eventos en dichos objetos.

El framework extendido sobre JPF, al estar implementado sobre plataforma Java, provee mecanismos de introspección de tipos, a través del uso de Reflection. Dichos mecanismos son la base para poder implementar la funcionalidad para definir propiedades sobre Clases Abstractas o padres, en la jerarquía de clases de un modelo.

Para explicar mejor el diseño y solución propuesta para este requerimiento, podemos definirlo en función de la interacción que realizan los distintos objetos durante la verificación. En la sección anterior, se ha explicado que, mientras se ejecuta una verificación que tiene asociadas una o más propiedades *Typestate*, existe un componente que monitorea la actividad de la ejecución del Modelo para poder determinar cuándo se crea o destruye un objeto de un determinado Tipo. La extensión que se implementó sobre este componente fue que, cuando se crea un nuevo objeto, no sólo busque las propiedades relacionadas con la clase concreta a la que pertenece dicho objeto sino que explorará ascendentemente su árbol de jerarquía para encontrar el conjunto de Clases de las cuales hereda y, de esta forma, poder asociar la propiedad correspondiente, en caso de que la hubiere.

Una aspecto a tener en cuenta sobre esta extensión es que, dada una clase (concreta o abstracta), el framework es capaz de administrar a lo sumo una propiedad para cada objeto que se crea sobre la jerarquía de ese Tipo, por lo tanto, no se debería poder definir más de una propiedad para los padres abstractos de una clase. Esto no obedece a una decisión de diseño, sino a una limitación a los efectos de simplificar la extensión de JPF. Es perfectamente factible extender al administrador de AFDs para permitir verificar más de una *Typestate Property* para cada objeto que se desee.

5.7 Conclusiones

Elegimos Java Path Finder por sus características de diseño y codificación open source. Esto nos permitió realizar con éxito diversas modificaciones orientadas a verificar propiedades expresadas como patrones de eventos sobre aplicaciones Java.

En esta línea, se destaca la modificación principal en la búsqueda y ejecución del árbol de estados del verificador para poder contemplar, en forma online, la evolución de estados de los verificadores de propiedades representados, cada uno, por un AFD. Esta modificación se implementó con una nueva clase (*DFSearchTesis*) que extiende la provista por JPF (*DFSearch)*. De esta forma, no se altera el código de JPF ni la estructura interna de los estados

del árbol de verificación, sino el algoritmo que lo recorre. Más precisamente, se modificó la lógica en la que se evalúan las condiciones para realizar backtracking sobre un estado ya explorado.

Una vez implementada la modificación principal, se despeja el camino para desarrollar nuevos features como son el contexto y preámbulo de verificación, dentro del marco de Escenarios de Control, y las propiedades *Typestate*. Sobre éstas últimas, podemos destacar algunas características como ser la posibilidad de definir propiedades, ya no a nivel global sino, sobre el comportamiento específico de diferentes tipos dentro de la jerarquía de clases del modelo. Este factor amplía el universo de propiedades que se pueden verificar y, por lo tanto, el conjunto de problemáticas que se pueden atacar con la verificación de patrones de eventos sobre JPF.

En JPF los *Listeners* se ejecutan en otro nivel de abstracción (por fuera del JPF y sobre la JVM que está corriendo la verificación). Un tema importante para analizar es el de caracterizar los tipos de propiedades que se pueden describir en JPF, específicamente, con los agregados del framework para la verificación de patrones de eventos.

En JPF, existe un conjunto reducido de instrucciones (bytecodes de Java, no instrucciones del modelo) que ocasionan un cambio de estado del árbol de verificación y, por lo tanto existen combinaciones en la ejecución concurrente de procesos que no se estarán explorando.

Es por este último motivo que se definieron los tipos de eventos que se pueden utilizar para describir propiedades de patrones de eventos en JPF, que son los asociados a instrucciones de invocación de métodos. Si bien, no todas las invocaciones generan cambios de estado en el árbol del verificador, es razonable suponer que una gran cantidad de eventos se defina sobre invocaciones (de métodos) dinámicas, debido a que las propiedades de patrones sobre JPF permiten explotar la verificación del comportamiento de instancias dentro del modelo.

Otro de los aspectos analizados es el tema del concepto con el cual se describe una propiedad. Una de las características para definir una propiedad basada en patrones de eventos es la modalidad mediante la cual se la representa. Como se puede ver en el capítulo 3, en el presente trabajo se decidió optar por utilizar la estructura de AFD para representar propiedades de patrones de eventos. Sin embargo, existen lenguajes que permiten predicar sobre las propiedades en forma no determinística, como por ejemplo Visual Timed Scenarios ([BRAB05]). Si bien puede ser complejo el proceso de determinizar un autómata o propiedad no determinística, existen herramientas que resuelven la problemática.

Con respecto a la detección de la violación de una propiedad, existe un factor a considerar que es el paralelismo en el que se ejecutan, por un lado el motor de verificación de JPF y por otro el verificador de propiedades basadas en patrones de eventos. Esto significa que la evaluación del estado actual de una propiedad se realiza únicamente cuando se completa la ejecución de un estado del verificador, es decir durante una transición de estados.

Por lo tanto, existirá forzosamente un retraso desde el instante en que un evento ocasiona una violación a la propiedad (o se invalida el contexto/preámbulo) hasta que se ejecuta la última instrucción del estado actual de la verificación.

El framework desarrollado sobre JPF está orientado principalmente a la verificación de patrones de eventos sobre modelos multithreading, debido a que es muy probable que la verificación sobre un solo thread no genere nuevos estados de la JVM (se omiten los casos en que el modelo haya sido instrumentado, por ejemplo, con el uso de la funcionalidad Verify[3] [JPF01]).

Por último, y como se describe en la sección 9.1, todas las modificaciones que se realizaron sobre el código y los componentes de Java Path Finder, siempre estuvieron orientadas a no alterar el diseño original y a modificar la menor cantidad de elementos. En este sentido, se destacan las modificaciones realizadas en el algoritmo de búsqueda.

Para poder realizar la obtención dinámica de información sobre los objetos que se van creando durante la verificación, fue necesario realizar modificaciones puntuales en declaraciones de métodos y visibilidad de una clase (ver 9.1.3).

[3] Verify es una clase incluida dentro del paquete de distribución de JPF, que provee, entre otras cosas, mecanismos para generar datos de input en la verificación del modelo (por ejemplo, todos los enteros dentro de un rango, todos los booleanos, etc.) de forma tal de generar múltiples ramas de ejecución.

Capítulo 6

Verificación de patrones de eventos con SPIN

6.1 Introducción

SPIN es un model checker de estados. Esto implica que para realizar la verificación, construye un grafo de estados, que permite determinar la forma en que estos se conectan. Esta noción de estados se extiende también a la forma en que se formalizan las propiedades.

El presente capítulo explora las posibilidades que SPIN provee para describir y verificar propiedades basadas en patrones de eventos. Para esto, se hace una breve descripción de la herramienta, su lenguaje de descripción de modelos y las principales estructuras de control de flujo.

Luego, se describen las distintas alternativas evaluadas para cumplir el objetivo antes mencionado. En definitiva, es necesario ajustar los conceptos de evento y patrón de eventos para adaptarlos a lo que SPIN puede proveer. En otras palabras, es necesario encontrar una manera de expresar la ocurrencia de eventos como propiedades de estados.

En particular, se presentan dos posibles implementaciones de patrones de eventos, una utilizando procesos monitores y otra utilizando *never claims*. En ambos casos, nos encontramos con limitaciones que impidieron continuar por dicho camino.

6.2 Funcionamiento general de SPIN

SPIN se basa en el hecho de que se pueden usar autómatas de estados finitos para modelar el comportamiento de procesos asincrónicos dentro de sistemas distribuidos. El lenguaje de especificación de base es Promela (acrónimo de Process Meta-Languaje); el mismo no es un lenguaje de implementación, sino que puede ser utilizado como lenguaje de descripción o especificación de sistemas (con énfasis en el modelado de la sincronización y coordinación de procesos).

Modelando al sistema como un único autómata (resultante de la sincronización de todos los procesos que lo componen), y utilizando el concepto de aceptación de Büchi, SPIN puede reconocer ejecuciones infinitas (esto es, saber si una ejecución visitaría infinitas veces un estado determinado). Para esto, se utilizan dos propiedades de los autómatas de Büchi:

- es decidible verificar si un lenguaje es vacío (aplicado a este problema, esto sería similar a decidir si el conjunto de ejecuciones del autómata de Büchi es vacío).
- es decidible la intersección de lenguajes (esto es, la intersección entre la composición de los autómatas del modelo y el never claim).

Tiene un motor semántico que ejecuta al modelo paso a paso. Esto es, en cada paso, selecciona una sentencia ejecutable (si hay más de una, elige cualquiera de ellas), la ejecuta,

actualiza el estado del proceso al que dicha instrucción pertenece y continua con la siguiente hasta que no quedan más instrucciones ejecutables.

Recorre el espacio de estados utilizando distintos algoritmos como DFS, Depth Limited Search (DFS pero con un límite de profundidad) o BFS.

6.2.1 Promela

Es el lenguaje utilizado por SPIN para la definición de modelos. Soporta estructuras de control no determinísticas, e incluye primitivas para la creación de procesos y la comunicación entre ellos. Utiliza tres tipos de objetos: procesos, que definen el comportamiento del sistema; objetos de datos, que pueden ser tipos básicos como bit, bool, byte, etc. o tipos definidos por el usuario; y canales, que se usan para modelar el intercambio de datos entre los procesos.

Cualquier instrucción, dependiendo del estado del sistema puede ser ejecutable o estar bloqueada. Cuando un proceso alcanza un punto en el que no tiene ninguna sentencia ejecutable, se bloquea (hasta que alguna instrucción pase a ser ejecutable). Las asignaciones y las sentencias print son siempre ejecutables; las expresiones son ejecutables si y solo si evalúan a verdadero (o son iguales a 0); las sentencias send() son ejecutables cuando el canal no está lleno; las sentencias receive() son ejecutables cuando el canal no es vacío.

Promela provee cinco sentencias de control de flujos: **atomic**, **if**, **do**, **d_step**, y **unless**. Nos enfocaremos en las primeras cuatro, que son las más importantes a los efectos del presente trabajo.

atomic se utiliza principalmente para ejecutar un conjunto de instrucciones de manera indivisible. En el caso de la Figura 6.1, a y b intercambian sus valores sin que se ejecute otra instrucción de otro proceso en el medio.

En el caso en que durante la ejecución de una secuencia atómica se encuentre alguna instrucción no ejecutable, se bloquea el proceso y otro toma el control. Luego, cuando la ejecución vuelve al proceso bloqueado, se retoma como si nunca se hubiera perdido el control.

```
atomic {
       tmp = b;
       b = a;
       a = tmp }
```

Figura 6.1: Ejemplo de uso de atomic

if (Selection) permite alternar el flujo de control de manera no determinística dentro de la ejecución del modelo. De todas las guardas ejecutables, SPIN las selecciona de a una de manera no determinística.

En el ejemplo de la Figura 6.2, puede verse una estructura de **if** con cuatro guardas. Asumiendo que las últimas 3 son ejecutables, SPIN las tomará como tres posibles caminos distintos de ejecución, e intentará recorrerlos a todos.

```
if
:: (a == c) -> Acción 1
:: (a == b) -> Acción 2
:: (b == c) -> Acción 2
:: (a == a) -> Acción 2
Fi
```

Figura 6.2: Ejemplo de uso de if

do (Repetitions) es de alguna manera similar al **if**, pero de forma cíclica. En la Figura 6.3 podemos ver un proceso formado principalmente por una estructura **do**, dentro de la cual se suma y resta (no deterministicamente) el valor 1 a la variable count.

Es importante notar la similitud del **do** con el **if**, en el sentido de que en ambos casos hay guardas que se ejecutan no deterministicamente. La principal diferencia entre ambas estructuras, es que el **do** permite ciclar indefinidamente hasta que la ejecución llega a un break.

```
active proctype counter(){
        do
        :: (count != 0) ->
                if
                :: count++
                :: count--
                fi
        :: (count == 0) -> break
       od}
```

Figura 6.3: Ejemplo de uso de do

6.2.2 Descripción de propiedades

SPIN permite ejecutar verificaciones tanto de propiedades Safety (propiedades cuyo contra ejemplo es finito), como liveness (propiedades cuyo contra ejemplo podría no tener longitud acotada). Dichas propiedades se formalizan a través ciertas construcciones provistas por Promela, estas son: Aserciones, etiquetas end-state, etiquetas progress-state, etiquetas accept-state, never claims y trace assertions.

Las aserciones son predicados que cuando evalúan a falso, le indican al model checker que se ha producido un error.

Las etiquetas **end-state** son etiquetas que permiten indicar al model checker que un determinado estado final es válido. Por defecto, los únicos estados finales válidos son aquellos en los que los procesos del sistema alcanzan el final de su código. Pero existen situaciones en las que es necesario que un proceso quede aguardando en un estado determinado esperando que se cumpla alguna condición. Las etiquetas end-state permiten al usuario indicarle al verificador que dichos estados son efectivamente válidos.

```
mtype { p, v };
chan sema = [0] of { mtype };

active proctype Dijkstra() {
        byte count = 1;
end:    do
        :: (count == 1) -> sema!p; count = 0
        :: (count == 0) -> sema?v; count = 1
        od
}
active [3] proctype user(){
        do
        :: sema?p;        /* enter */
critical: skip;           /* leave */
          sema!v;
        od
}
```

Figura 6.4: Ejemplo de uso de end-state

Las etiquetas **progress-state** se utilizan para indicar al model checker que la ejecución de determinadas sentencias efectivamente implica un progreso en la ejecución del modelo (esto es, no es un ciclo infinito) [HOLZ03]. Si durante la ejecución de la verificación se encuentran ciclos no marcados como "de progreso", el model checker puede inferir la existencia de un ciclo infinito y posiblemente algún problema de inanición.

```
active proctype Dijkstra() {
        byte count = 1;
end:    do
        :: (count == 1) ->
progress:       sema!p; count = 0
        :: (count == 0) -> sema?v; count = 1
        od
}
```

Figura 6.5: Ejemplo de uso de progress-state

Las etiquetas **accept-state** se utilizan para indicar al model checker que deben encontrarse todos los ciclos que pasan por al menos una de dichas etiquetas [HOLZ03].

Los **never claims** son un mecanismo más complejo y poderoso que los mencionados hasta el momento. Habitualmente se utilizan para especificar comportamiento (finito o infinito) que no debería ocurrir (antipropiedades). Permiten chequear la propiedad justo antes y justo después de cada ejecución de cada sentencia del modelo; éstas son las que es necesario verificar en cada paso de la ejecución.

En el ejemplo de la Figura 6.6, se chequeará la condición p en cada paso de la ejecución del modelo hasta que evalúe a falso. En dicho caso, el never claim saldrá del ciclo y terminará su ejecución, y SPIN reportará un error.

```
never {
        do
        :: !p -> break
        :: else
        Od
}
```

Figura 6.6: Ejemplo de uso de never claim

Los trace assertions permiten describir secuencias de operaciones que los procesos pueden hacer sobre los canales de mensajes. Se aplican únicamente a operaciones send() y receive(). No se ejecutan sincrónicamente como el never claim. Se ejecutan únicamente cuando ocurre la operación monitoreada.

6.3 Detección de eventos

La complejidad de definir eventos en SPIN, radica en que es un model checker basado en estados. Como tal, los mecanismos y herramientas provistos para definir predicados, se orientan a este tipo de construcciones. No provee un mecanismo de manejo de eventos. No es un concepto para el cual la herramienta haya sido concebida y por lo tanto no se encuentra realmente preparada para soportarlo.

Para detectar la ocurrencia de eventos durante la ejecución del modelo, se evaluaron distintas alternativas:

- Envío / recepción de mensajes por canales: La idea general de este enfoque, es suponer que el evento ocurre en el momento en que se envía o recibe un mensaje por un canal.
- Remote referencing: La idea general de este enfoque, es suponer que el evento ocurre en el momento en que la ejecución del modelo entra o sale de una determinada línea de código identificada mediante una etiqueta.
- Cambios de valores en las variables del modelo: La idea general de este enfoque, es que el evento ocurre en el momento en que alguna variable cambia de valor.

La primera alternativa fue descartada debido a que no hay instrucciones que permitan observar el estado del canal. El único mecanismo para actuar ante cambios en los canales, es el de las Trace Assertions, que tienen importantes restricciones (por ejemplo, no se pueden declarar ni referenciar objetos de datos).

La segunda alternativa fue descartada debido a que, utilizando este mecanismo, en cada paso de la ejecución, SPIN permite determinar cuál es el conjunto de instrucciones ejecutables (no bloqueadas), pero no permite saber exactamente cuál de estas se va a ejecutar en el próximo paso (ni tampoco cual se ejecutó). Lo cual hace que no sea de utilidad para detectar eventos. Otra restricción es que implica el uso de instrumentación de código agregando un label por instrucción.

La tercera alternativa es la que se analizó en mayor detalle. En este caso, es inevitable la instrumentación de código, pero es la que a priori parece con más posibilidades de éxito. La

idea general, es incluir variables (flags) en el código del modelo, y setear sus valores para indicar la ocurrencia de los eventos.

Los eventos son instantáneos, es por esto, que los “flags” utilizados para representarlos deben ser limpiados luego de su uso. Ya que de otra manera, esto puede derivar en inconsistencias dentro del modelo.

Sin embargo, esta alternativa presentó dificultades y también fue descartada. A continuación, se muestra la forma en que es necesario instrumentar el código para detectar la ocurrencia de eventos y mostraremos las dificultades encontradas.

Ejemplo 5: Elección de líder en un anillo unidireccional

El ejemplo muestra un algoritmo distribuido en una red de topología de anillo unidireccional donde los distintos nodos se envían mensajes entre sí para determinar el próximo líder. Se desea verificar que “el algoritmo selecciona un único líder dentro de la red”.

En la Figura 6.7 puede observarse código Promela del modelo sin instrumentación de código. Cuenta con un proceso node, que representa a los nodos de la red (una instancia del proceso por nodo de la red); y un proceso init, utilizado para ejecutar el modelo. El objetivo de la sentencia atomic es evitar que comiencen a ejecutarse los nodos antes de que se encuentren todos inicializados.

```
1  #define N 5
2  #define I 3
3  #define L 10
4  mtype = { one, two, winner };
5  chan q[N] = [L] of { mtype, byte};
6
7  proctype node (chan in, out; byte mynumber){
8    bit Active = 1, know_winner = 0;
9    byte nr, maximum = mynumber, neighbourR;
10   xr in;
11   xs out;
12   printf("MSC: %d\n", mynumber);
13   out!one(mynumber);
14   end: do
15        :: in?winner, nr ->
16           out!winner, nr;
17           break;
18        :: in?one(nr) ->
19           if
20           :: Active ->
21              if
22              :: nr > mynumber ->
23                 printf("MSC: LOST\n");
24                 Active = 0;
25                 out!one, nr;
26              :: nr == mynumber ->
27                 assert(nr == N);
28                 out!winner, mynumber;
29                 printf("MSC: LEADER\n");
30                 break;
31              :: nr < mynumber ->
32                 out!one, mynumber;
33              fi
34           :: else ->
35              out!one, nr;
36           fi
37     od
38   }
39 init {
40 byte proc;
41 atomic {
42    proc = 1;
43    do
44    :: proc <= N ->
45       run node (q[proc-1], q[proc%N], (N+I-proc)%N+1);
46       proc++
47    :: proc > N ->
48       break
49    od
50 }
51}
```

Figura 6.7: Modelo de selección de líder

Para representar los eventos definidos es necesario realizar varios ajustes al código. El primer paso es declarar las variables que representarán los distintos eventos que intentaremos reconocer: InicioEjecución, MensajeNW y MensajeW (Figura 6.8).

Para “agregar” las “ocurrencias” de los eventos al código del modelo se incluyen las modificaciones a los flags en los puntos del código donde dichos eventos se producen. En la Figura 6.9, se muestra el código del modelo luego de esta instrumentación. Como puede verse, a continuación de las líneas 24, 27 y 31, se han agregado distintos llamados a la macro *states*.

La macro *states* se utiliza para simplificar el código del modelo. Ante la ocurrencia de cada evento es imprescindible actualizar TODOS los flags al mismo tiempo (esta actualización debe realizarse atómicamente para garantizar que la operación no sea interrumpida por ningún otro proceso). Supongamos que el sistema ha detectado la ocurrencia del evento Inicio Ejecución (la variable InicioEjecucion ha tomado el valor 1) y, algunas instrucciones más adelante, se detecta el evento Mensaje No Winner (la variable MensajeNW ha tomado el valor 1); si el primer flag no se limpiara, entonces, se corre el riesgo de detectar cualquiera de dichos eventos (ya que ambos estarían activados al mismo tiempo).

Dependiendo de los parámetros con que se invoca a la macro *states*, indica la ocurrencia de los distintos eventos definidos. Con los parámetros (1,0,0) indica la ocurrencia del evento Inicio Ejecución; con los parámetros (0,1,0) indica la ocurrencia del evento Mensaje No Winner; y con los parámetros (0,0,1) indica la ocurrencia de un evento Mensaje Winner.

Las expresiones ie, nw y w (Figura 6.9, parte superior) son expresiones booleanas que luego serán utilizadas para la construcción del autómata.

```
bit InicioEjecucion;
bit MensajeNW;
bit MensajeW;
```

Figura 6.8: Variables definidas para la observación de eventos

```
#define ie InicioEjecucion==1
#define nw MensajeNW==1
#define w MensajeW==1

#define states(newIE,newNW,newW) \
  atomic{InicioEjecucion=newIE;
         MensajeNW=newNW;
         MensajeW=newW;}
#define clearstates() states(0,0,0)
```

Figura 6.9: Instrucciones definidas para la instrumentación de código

```
21              if
22              :: nr > mynumber ->
23                 printf("MSC: LOST\n");
24                 Active = 0;
                   states(0,1,0);
25                 out!one, nr;
26              :: nr == mynumber ->
27                 assert(nr == N);
                   states(0,0,1);
28                 out!winner, mynumber;
29                 printf("MSC: LEADER\n");
30                 break;
31              :: nr < mynumber ->
                   states(0,1,0);
32                 out!one, mynumber;
33              fi
```

Figura 6.10: Instrumentación realizada usando instrucciones auxiliares

La problemática existente con respecto a la detección de eventos vía flags es que no es posible realizar un test&set de la variable. Esto significa que luego de detectar la ocurrencia de un estado (al setear los flags) no es posible garantizar que, desde el mismo proceso, se limpien dichas variables. Esta dificultad nos plantea la necesidad de encontrar un mecanismo que: 1. permita setear una variable para indicar que un evento ocurrió; 2. permita observar la ocurrencia del mismo desde un observador (ie, un monitor); y 3. permita limpiar el valor del flag para futuras referencias al mismo evento. Y todos estos pasos deben hacerse de manera atómica.

6.4 Propiedades basadas en patrones de eventos

Si bien no es sencillo el mecanismo de instrumentación de código definido para reconocer la ocurrencia de eventos; en principio parece ser viable. El problema ahora, es encontrar una manera de implementar las propiedades basadas en patrones de eventos que usen esta instrumentación de código.

SPIN provee mecanismos para definir estructuras de control similares a los autómatas que representan las propiedades, pero aún así es dificultosa la tarea de encontrar un mapeo entre ambos

Un primer enfoque fue utilizar procesos monitores. La posibilidad de utilizar etiquetas e instrucciones goto (entre etiquetas) permitió modelar los patrones de eventos de forma simple y declarativa. No obstante, no fue suficiente para cumplir con el objetivo planteado. La forma en que SPIN analiza el espacio de estados hace que siempre se encuentre un camino de ejecución erróneo, que es inherente a la forma en que se construye esta estructura. El detalle de este problema se explica en el experimento 1 donde se implementa la verificación de patrones mediante procesos monitores.

La estructura provista por SPIN para definir y verificar propiedades es el Never Claim. Utilizando esta estructura podemos emplear los mismos conceptos que para el monitor y definir escenarios utilizando labels e instrucciones goto.

Este nuevo enfoque permitió superar el primer problema de los monitores. No obstante, nos vimos forzados a incluir instrucciones de actualización de variables dentro del código del never claim. Esto hace que SPIN, muestre warnings de compilación, y en definitiva, imposibilite la verificación de propiedades.

6.4.1 Experimento 1: Patrones de eventos implementados sobre monitores

La idea principal, es describir al autómata dentro de un proctype utilizado para monitorear la verificación del modelo. De esta manera, para ejecutar la verificación, SPIN debe hacer la sincronización del modelo y el monitor.

Como resultado de esta sincronización, SPIN recorrerá todos los posibles caminos de ejecución del modelo (sincronizado con el monitor) y, en caso de haber alguno en que el autómata llegue a un estado final inválido, lo reportará.

La construcción del monitor en sí, es sencilla. Es simplemente un proceso Promela estándar que tiene el mismo tratamiento y manejo de schedulling que los demás procesos del modelo.

La construcción del autómata se realiza utilizando etiquetas para representar a los estados e instrucciones goto para representar las transiciones. Es imprescindible para el correcto funcionamiento del autómata, incluir un estado de error y las transiciones a dicho

estado en los casos en que ocurra un evento que viole la propiedad. Esto último se logra incluyendo un estado final con ciclo infinito.

A priori, hay casos en que este enfoque funciona, pero en [HOLZ03] se recomienda usarlo sólo para casos simples. Puntualmente para la forma en que se decidió construir el autómata, siempre hay una ejecución, que incluye un ciclo infinito. Motivo por el cual, la verificación falla.

Para presentar una explicación más detallada de la problemática, continuamos con la explicación del ejemplo Ejemplo 5.

Ejemplo 6: Elección de líder en un anillo unidireccional (continuación)

En la Figura 6.11 se muestra la representación gráfica de la propiedad a verificar. Los eventos que incluye son los siguientes: Inicio de Ejecución (ocurre cuando comienza la ejecución del algoritmo), Mensaje NO Winner (ocurre cada vez que se envía un mensaje distinto de Winner) y Mensaje Winner (ocurre cada vez que se envía un mensaje Winner).

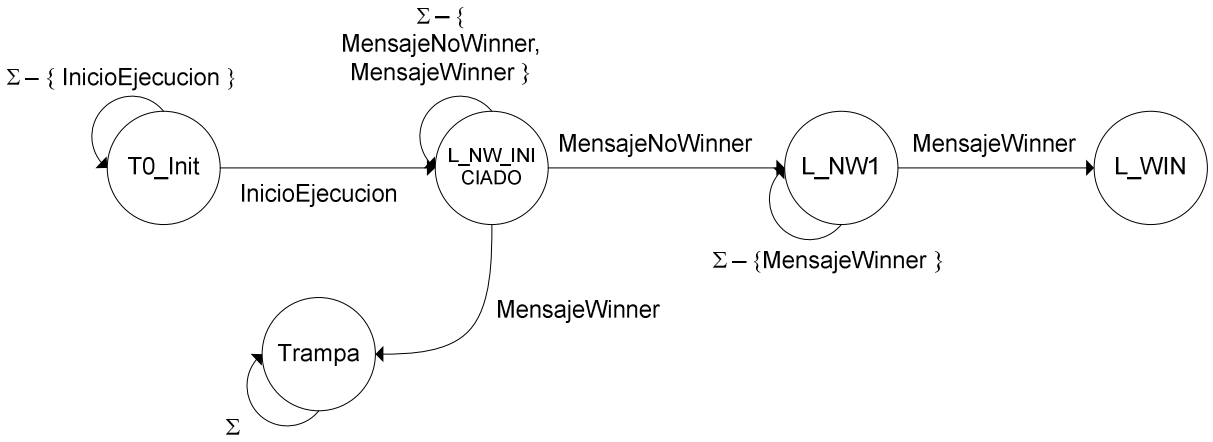

Figura 6.11: Diagrama de la propiedad del Ejemplo 6

La forma en que se define el autómata que representa a la propiedad se muestra en la siguiente figura. Es un proceso típico de SPIN, que contiene una etiqueta por cada estado del autómata (T0_init, L_NW_INICIADO, L_NW1, Trampa y L_WIN). Trampa es el estado que representa el estado inválido en la ejecución y L_WIN el estado donde convergen todos los escenarios de eventos que cumplen la propiedad.

Al estado Trampa se llega por una serie de eventos inválidos o no deseados. La estructura de Trampa se define como un ciclo infinito para lograr que SPIN lo invalide como estado final logrando que la ejecución total sea inválida.

```
proctype Monitor (){
T0_init:
   if
   :: (ie) -> goto L_NW_INICIADO
   :: goto T0_init
   fi;
L_NW_INICIADO:
   if
   :: (w) -> goto Trampa
   :: (nw) -> goto L_NW1
   :: goto L_NW_INICIADO
   fi;
L_NW1:
   if
   :: (w) -> goto L_WIN
   :: goto L_NW1
   fi;
Trampa:
   do
   :: skip
   od;
L_WIN:
   skip;
}
```

Figura 6.12: Proceso monitor que representa la propiedad

Cuando se ejecuta la verificación del modelo instrumentado según lo descrito durante esta subsección, SPIN hace la intersección de todos los procesos participantes (5 instancias distintas del proceso node y una instancia del proceso monitor) y busca todas las posibles ejecuciones.

En particular, hay una ejecución en la que el primer proceso que comienza a ejecutarse es el monitor. En este caso, el monitor continúa ejecutándose infinitamente en el primer ciclo (estado T0_Init) y esto hace que la verificación falle. A priori esto podría corregirse incluiyendo flags de inicio de cada proceso y, una vez seteados todos los flags, dando comienzo a la ejecución del monitor. No obstante, esto retrasaría el problema, ya que igualmente SPIN encontraría una ejecución en la que existe un ciclo infinito dentro del monitor.

6.4.2 Experimento 2: Patrones de eventos implementados sobre never claims

En cierto sentido, los never claims (o NC para abreviar) son similares a los monitores, pero presentan una ventaja fundamental; son la única construcción que provee SPIN que permite hacer validaciones al modelo antes y después de ejecutar cada instrucción.

Con el uso de NCs se puede superar la dificultad presentada por los monitores, ya que SPIN ejecuta una instrucción del NC por cada instrucción del modelo. Es importante notar que en el caso de los monitores, estos eran sincronizados por SPIN como parte del modelo, y ésa era la causa por la cual no se pudieron utilizar para la definición de propiedades, mientras que a los NCs no se los considera parte del modelo.

El problema con esta alternativa, es que en el NC es necesario ejecutar sentencias de actualización de los flags. Si bien el compilador de SPIN no prohíbe la modificación de variables en dentro de los NCs, muestra un warning. Por otro lado, en la documentación de la herramienta se recomienda no hacerlo.

El resultado final, fue que las verificaciones realizadas de esta manera no funcionaron correctamente. Para describir más detalladamente el enfoque seguido y los problemas hallados, continuamos con el ejemplo de selección de líder.

Ejemplo 7: Elección de líder en un anillo unidireccional (usando never claims)

El cuerpo del never claim se construye de la misma manera que el del monitor, con la excepción de que los estados Trampa y Final no son necesarios. En la Figura 6.13, se muestra el autómata que representa a la propiedad. Es importante destacar que es distinto al anterior, porque es necesario cambiar la propiedad para reflejar lo que el modelo NO debe hacer en lugar de lo que debe hacer.

El Never Claim escrito en código Promela puede verse en la Figura 6.14. Tiene dos estados representados por las etiquetas E0 y OK. Al ejecutarse dos ocurrencias del mensaje Winner, el Never Claim llega al final (label Err) y la verificación falla (la propiedad se viola).

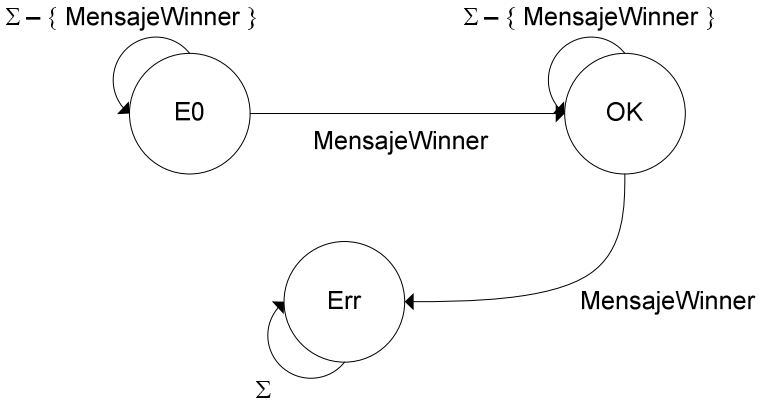

Figura 6.13: Diagrama de la propiedad "no se eligen dos winners"

```
#define w   MensajeW==1

never {
   E0:
      do
      :: w -> break;
      :: !w
      od;
   OK:
      do
      :: w -> break;
      :: !w
      od;
   Err:
      skip;
}
```

Figura 6.14: Never Claim que representa la propiedad "No se eligen 2 Winners"

Esta implementación de los escenarios presenta algunos problemas que finalmente no pudieron resolverse de manera satisfactoria. Es por esto que tuvimos que descartar la opción de seguir trabajando con SPIN.

Como podemos ver en el ejemplo anterior, en ciertos modelos es necesario observar la ocurrencia de un mismo evento más de una vez. Es por esto que, una vez observado el evento, es necesario limpiar la variable (es decir, regresarla a su valor inicial).

En el Never Claim implementado de la Figura 6.14, se busca que el evento *w* ocurra dos veces. Si no se reseteara la variable MensajeW luego de observada la primera ocurrencia, en la siguiente observación se volvería a detectar (erróneamente) la ocurrencia del evento. Es por esto, que la variable MensajeW debe ser limpiada luego de ser observada. Para esto, hay dos opciones: o bien en el modelo, luego del seteo de la variable; o en el Never Claim, luego de la observación del evento (test & set).

En la Figura 6.15, puede verse la porción de código resultante de agregarle la limpieza de los flags al código de la Figura 6.10. Aquí puede verse claramente que esta solución no es viable, ya que luego de ejecutar la sentencia states (x,y,x) del modelo, SPIN ejecutará la sentencia correspondiente del never claim y volverá la ejecución al modelo, pero sin garantías de que se ejecute la sentencia clearstates(); de hecho, no hay forma de conocer a priori, que proceso del modelo se ejecutará a continuación.

Código del modelo sin la Limpieza de flags	Código del modelo luego de agregar la limpieza de los flags

if
:: nr > mynumber -> printf("MSC: LOST\n"); Active = 0; **states(0,1,0);**
 	
  out!one, nr;
:: nr == mynumber -> assert(nr == N); **states(0,0,1);**
 	
  out!winner, mynumber; printf("MSC: LEADER\n"); break;
:: nr < mynumber -> **states(0,1,0);**
 	
  out!one, mynumber;
Fi
	if
	:: nr > mynumber -> printf("MSC: LOST\n"); Active = 0; **states(0,1,0);** **clearstates();** out!one, nr;
	:: nr == mynumber -> assert(nr == N); **states(0,0,1);** **clearstates();** out!winner, mynumber; printf("MSC: LEADER\n"); break;
	:: nr < mynumber -> **states(0,1,0);** **clearstates();** out!one, mynumber;
	Fi

Figura 6.15: Código de la Figura 6.10 agregando la limpieza de los flags

En la Figura 6.16, puede observarse la porción de código resultante al agregarle la limpieza de los flags, dentro del código del never claim (Figura 6.14). En este caso, determinar la validez de esta opción es más complejo. Luego de ejecutar la instrucción states(0,0,1), SPIN pasa a ejecutar el never claim. Esto es, se evalúa la condición (si se cumple w, lo cual es lo mismo que determinar si MensajeW es igual a 1) y luego se limpia el flag.

El problema con este enfoque es que SPIN sólo ejecuta una instrucción del never claim y para que funcione correctamente es necesario observar el evento, limpiar la variable y ejecutar lo correspondiente a la ocurrencia del evento, todo de forma atómica.

Código del never claim sin la Limpieza de flags	Código del never claim luego de agregar la limpieza de los flags

`#define w MensajeW==1`
 	
`never {`
`  E0:`
`  do`
`  :: w -> break;`
 	
`  :: !w`
`  od;`
`  OK:`
`  do`
`  :: w -> break;`
 	
`  :: !w`
`  od;`
`  Err:`
`  skip;`
`}`
	`#define w MensajeW==1`
	
	`never {`
	`  E0:`
	`  do`
	`  :: w ->`
	`  **clearstates();**`
	`  break;`
	`  :: !w`
	`  od;`
	`  OK:`
	`  do`
	`  :: w ->`
	`  **clearstates();**`
	`  break;`
	`  :: !w`
	`  od;`
	`  Err:`
	`  skip;`
	`}`

Figura 6.16: Código de la Figura 6.14 agregando la limpieza de los flags

Para resolver este último problema, se puede utilizar instrucción *atomic*, de esta forma, podemos implementar una instrumentación con características similares al TestAndSet(X) que permita verificar el valor de la variable X y luego setearlo, o resetearlo, para poder observar la ocurrencia del evento más de una vez.

```
if
  :: atomic {
     (expresion) ->
       cambio de estado;
       GOTO sgte_estado
  }
fi;
```

Figura 6.17: Test And Set de una expresión

Si bien, a priori, esta implementación ad-hoc parece permitir salvar la dificultad antes mencionada, sigue siendo problemática, ya que durante la compilación SPIN genera un warning y durante la ejecución de la verificación, se llega a la máxima cantidad de estados a verificar sin encontrar errores (donde debería encontrarlos).

Analizando otras alternativas, se podría haber planteado la posibilidad de interceptar eventos a nivel de canales (channels) de Promela. Es decir, definir propiedades en las cuales los eventos estén relacionados con la lectura y escritura de un determinado canal entre dos o

más procesos. Esto podría llevarse a cabo agregando en el modelo un proceso que sea el intermediario de la escritura y lectura del canal por parte de los otros procesos. Entendemos que esta instrumentación del código del modelo es bastante compleja y se está modificando gran parte de la lógica corriendo el riesgo de introducir más errores de los que se encuentran.

6.5 Conclusiones

SPIN es un model checker de estados. No provee mecanismos nativos para la captura y tratamiento de eventos, es por esto que para expresar propiedades basadas en patrones de eventos utilizando esta herramienta, se necesitan dos cosas: poder expresar la ocurrencia de eventos como propiedades de estados y poder expresar las propiedades como patrones de eventos.

En el primer caso, la mejor alternativa, es utilizar variables como flags para detectar el momento en que se produce el evento. Esta solución requiere mucha instrumentación de código, ya que no sólo es necesario definir las variables, sino que también setearlas en los momentos dados dentro del código del modelo.

En el segundo caso, la mejor alternativa es la utilización de never claims, ya que permiten utilizar etiquetas e instrucciones goto para modelar los patrones de eventos de forma simple y declarativa.

Esta solución no funciona, ya que es condición obligatoria la utilización de instrucciones que modifiquen los flags dentro del never claim.

Capítulo 7

Caso de Estudio

7.1 Introducción

El objetivo del caso de estudio es evaluar el framework desarrollado sobre un modelo de complejidad superior a los ejemplos presentados hasta el momento. Los requerimientos definidos para la elección del modelo fueron los siguientes:

- Debía ser un modelo de complejidad media o alta.
- Debía ser un modelo que emplee las funcionalidades de multithreading de Java.

El primer requerimiento permite especificar mayor variedad de propiedades y enriquece los experimentos. El segundo garantiza que durante la verificación JPF genere un espacio de estados de un tamaño interesante.

A partir de estos requisitos, se eligió un sistema de controlador de ascensores como caso de estudio. Este sistema se encarga de controlar y administrar los pedidos que recibe el conjunto de ascensores asociado a él.

Para poder verificar el modelo definido hubo que diseñar *stubs* que representen tanto a los ascensores como a las personas, y operen en threads de ejecución distintos. La idea principal es que el modelo funcione de manera similar a la realidad.

Las pruebas se realizaron sobre distintos escenarios, cada uno de ellos con una configuración distinta de ascensores y personas.

7.2 Alcance del experimento

Si bien el modelo permite definir muchas instancias de cada una de las entidades participantes (ascensores y personas), para el presente experimento se limitaron los escenarios a los que representan la interacción entre a lo sumo 3 personas y 2 ascensores.

Por otro lado, la cantidad de propiedades y contextos utilizados se restringió a cinco. Esto no garantiza que el modelo no contenga errores, pero el objetivo principal no era garantizar la ausencia de defectos sino determinar la utilidad del framework desarrollado.

7.3 Desarrollo

7.3.1 Descripción del modelo

El modelo utilizado para este experimento es un controlador de un conjunto de ascensores. Cuenta con clases para representar los componentes más importantes: *ControladorAscensor*, *Ascensor* y *Persona*.

Los dos últimos son simplemente *stubs* utilizados para simular la comunicación con el Controlador, y como puede verse en el código entregado con este trabajo, su lógica es realmente simple.

Es importante destacar que el Controlador de por sí solo no es auto contenido para realizar la verificación, en el sentido en que, de no agregar los simuladores de ascensores y personas, no podríamos verificar ninguna propiedad (a diferencia del ejemplo de selección de líder).

En la siguiente figura, se presenta el protocolo de comunicación del controlador. Cada tipo de mensaje que este acepta (o envía) se representa con un método Java de la clase que lo implementa.

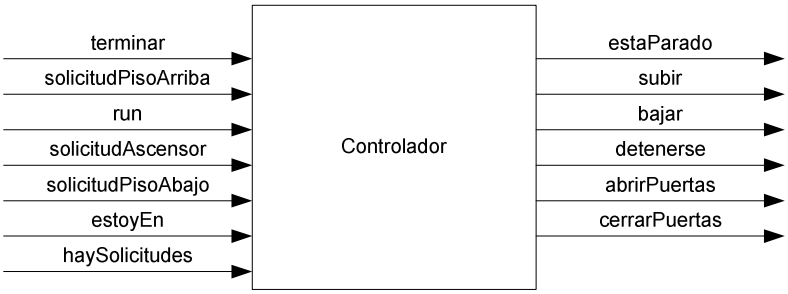

Figura 7.1: Protocolo de comunicación del Controlador

En el modelo, se utiliza un thread por cada objeto de la clase *Ascensor*, *Persona* y *ControladorAscensor*. Esto es para simular la independencia entre los distintos participantes de la verificación.

En la Figura 7.2 se presenta un diagrama de secuencias del modelo ejecutando utilizando un Ascensor y una Persona.

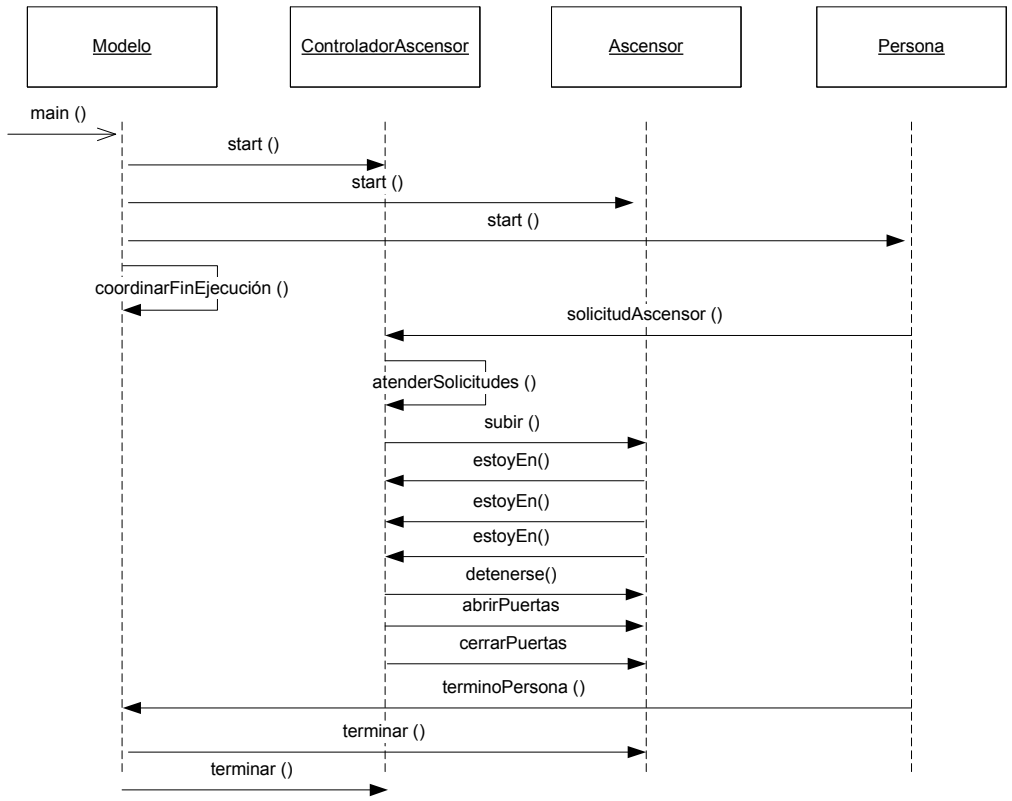

Figura 7.2: Diagrama de secuencias del controlador

Como se observa en la Figura 7.2, existe un objeto Modelo que es el encargado de crear todo el ambiente de la simulación. El mismo dispara la creación de los elementos y la inicialización de los mismos. La Persona, luego de realizar una solicitud, se comunica con el Modelo para indicarle que finalizó con sus solicitudes. El Modelo utiliza este mensaje para cerrar las operaciones de los demás objetos (ascensores y controlador).

7.3.2 Propiedades

Sobre el modelo presentado en la sección anterior se definieron cinco propiedades. Las verificaciones de dichas propiedades se ejecutaron utilizando diversos escenarios de control para disminuir el tamaño del universo de estados explorado.

7.3.2.1 Propiedad 1: "El ascensor no puede recibir orden de subir ni bajar si la puerta no esta cerrada"

Esta propiedad está orientada a verificar que el ascensor no pueda moverse teniendo las puertas abiertas. En la siguiente figura se muestra la representación gráfica de la antipropiedad correspondiente para verificar la violación de este comportamiento:

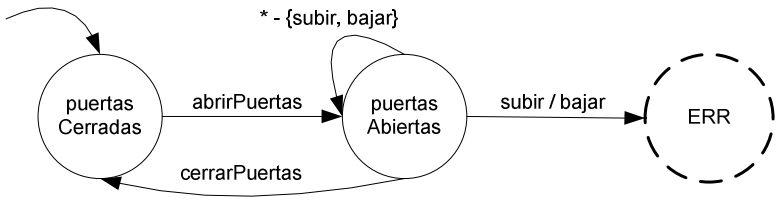

Figura 7.3: Representación gráfica de la primera propiedad del Controlador

Los eventos utilizados en este caso fueron

- runPersona
- runAscensor
- runControlador
- abrirPuertas
- cerrarPuertas
- subir
- bajar
- terminoPersona
- detenerse

En este caso, utilizamos un contexto que fuerza a JPF a podar las ejecuciones que cumplen con las siguientes condiciones:

- Los threads de Ascensor, Controlador y Persona comienzan a ejecutarse en un orden distinto al aquí mencionado.
- El thread Persona llega a su fin (se ejecuta el método *terminoPersona*).

Es importante destacar que no consideramos oportuno continuar con la verificación luego de que el thread que representa a la Persona termina su ejecución, ya que se producirán varias ramas de ejecución en las cuales: la persona realiza los requests y, a continuación, informa al thread main el fin de su ejecución (terminoPersona) que desencadenará la terminación de todos los threads del modelo (Ascensores y ControladorAscensor) sin que hayan podido ejecutarse siquiera y satisfacer los pedidos. Estas particularidades se deben a una simplificación del modelo para coordinar la destrucción de los threads.

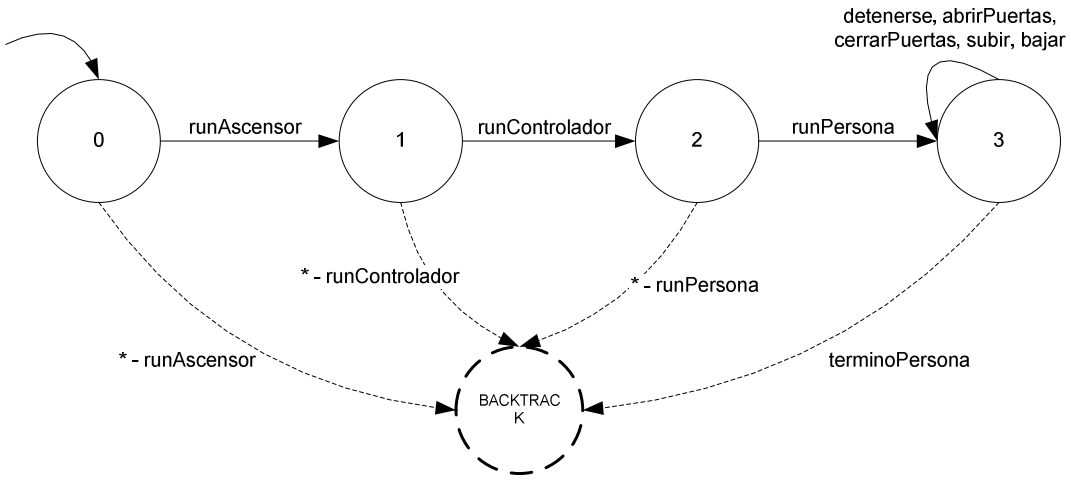

Figura 7.4: Representación gráfica del escenario de control utilizado para la primera propiedad del Controlador

7.3.2.2 Propiedad 2: "El ascensor no debe abrir las puertas si está en movimiento".

En la siguiente figura se muestra la representación gráfica de la antipropiedad:

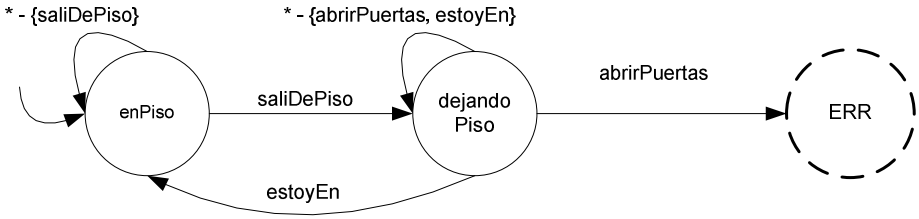

Figura 7.5: Representación gráfica de la segunda propiedad del Controlador

Los eventos definidos para la verificación de la propiedad y el escenario de control fueron:

- runPersona
- runAscensor
- runControlador
- abrirPuertas
- estoyEn
- saliDePiso
- terminoPersona

En la siguiente figura puede verse el escenario de control utilizado para la verificación de la propiedad. Es muy similar al utilizado en la propiedad 1. La única diferencia se da en los eventos que representan transiciones del estado 3 a sí mismo. Como puede verse en ambas figuras, los eventos que participan en el escenario de control de la propiedad 1 y no en el de la propiedad 2 (*cerrarPuertas, subir, bajar y detenerse*) no se están analizando en esta última.

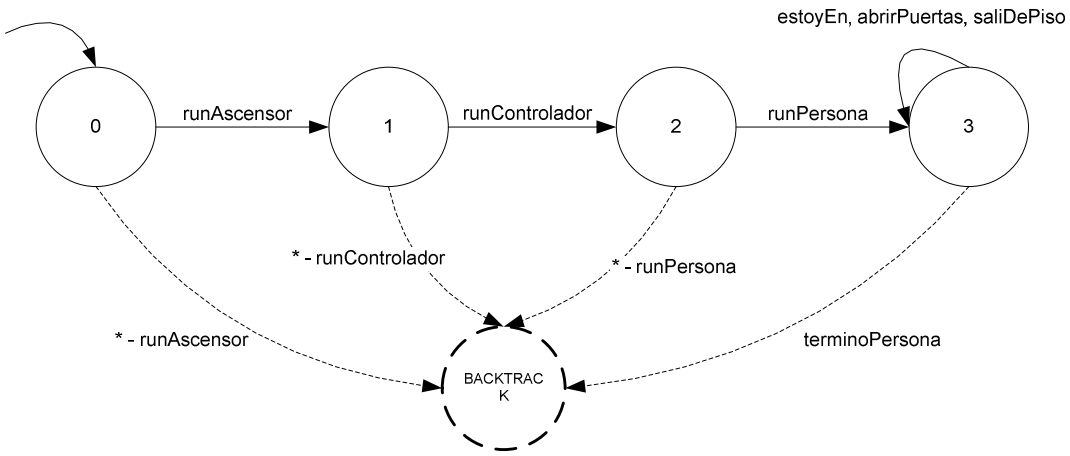

Figura 7.6: Representación gráfica del escenario de control utilizado para la segunda propiedad del Controlador

7.3.2.3 Propiedad 3: "Siempre que el ascensor llega a algún piso, tiene que abrir la puerta y luego cerrarla"

Esta propiedad podría expresarse de la siguiente forma: "Siempre que hay un evento detener, tiene que haber a continuación un evento abrir puertas y luego uno cerrar puertas", intentando simular la situación donde un ascensor llega a un piso para atender una solicitud, las personas bajan o suben del mismo y luego cierra las puertas para continuar su tarea.

En la siguiente figura se muestra la representación gráfica de la antipropiedad correspondiente:

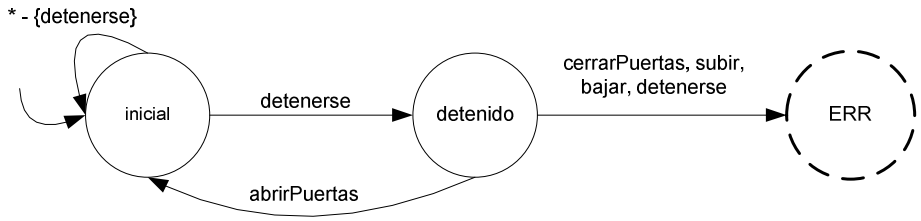

Figura 7.7: Representación gráfica de la tercera propiedad del Controlador

Los eventos y el escenario de control utilizados son los mismos que para la propiedad 1, y dado que en ninguno de los casos se encuntraron violaciones, el espacio de estados generado es el mismo para ambas propiedades (con y sin escenario de control, ver 7.4).

7.3.2.4 Propiedad 4: "Si se ha recibido una solicitud para el piso 1, el ascensor debe atenderla cuando pasa por ese piso (independientemente en que se generen en el mismo instante en que el ascensor arriba al piso)".

Para verificar esta propiedad, hubo que instrumentar en cierta forma el modelo debido a que nuestro framework no soporta la detección de ocurrencia de eventos contemplando los valores de los parámetros de los métodos que lo representan.

De esta manera, se agregaron al modelo los métodos *solicitudAscensorPiso1* y *estoyEnPiso1* (para poder identificar sólo las solicitudes y arribos a determinado piso).

Por lo tanto, esta propiedad podría expresarse de la siguiente forma: “si luego de un evento *solicitudAscensorPiso1* llega un evento *estoyEnPiso1*, entonces el siguiente evento debe ser *atenderSolicitudPiso*”.

En la siguiente figura se muestra la representación gráfica de la antipropiedad:

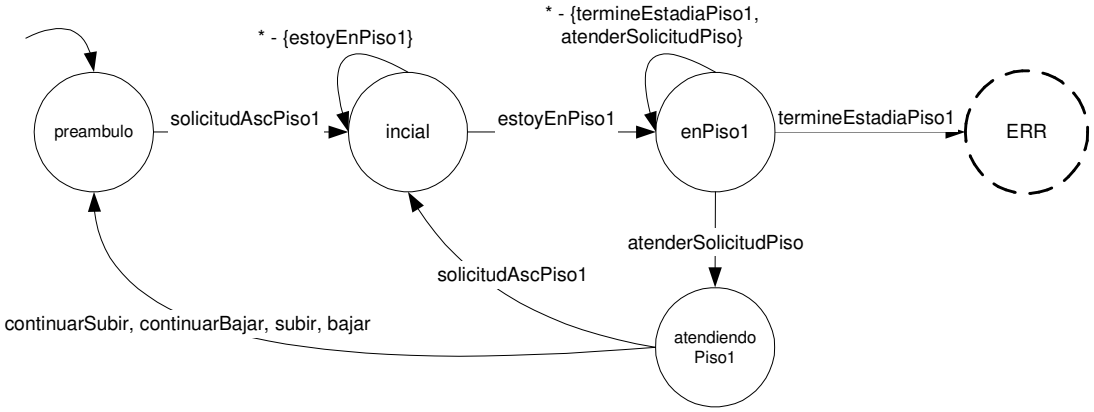

Figura 7.8: Representación gráfica de la cuarta propiedad del Controlador

El código del método solicitudAscensor instrumentado para poder representar la propiedad, se muestra a continuación.

```
public void solicitudAscensor(Ascensor a, int pisoDestino) {
   switch (pisoDestino) {
      case 1: solicitudAscensorPiso1(); break;
   }
   super.solicitudAscensor(a, pisoDestino);
}
```

Como puede verse, para el caso donde *pisoDestino* es 1, se invoca al método *solicitudAscensorPiso1* (que no tiene código) para representar la ocurrencia del evento *solicitudAscensorPiso1*.

Al ejecutar la verificación de esta propiedad, el framework reporta un error, como era esperado. Esto se debe a que, la ocurrencia del evento *solicitudAscensorPiso1* se detecta antes de que se actualice el estado del sistema para indicar que el ascensor tiene una solicitud para ir al piso 1 (invocación del método super.solicitudAscensor).

Para solucionar este defecto, es necesario hacer la siguiente modificación al código.

```
public void solicitudAscensor(Ascensor a, int pisoDestino) {
   super.solicitudAscensor(a, pisoDestino);
   switch (pisoDestino) {
      case 1: solicitudAscensorPiso1(); break;
   }
}
```

Para la verificación de esta propiedad se utilizaron dos escenarios de control distintos (Figura 7.9 y Figura 7.10).

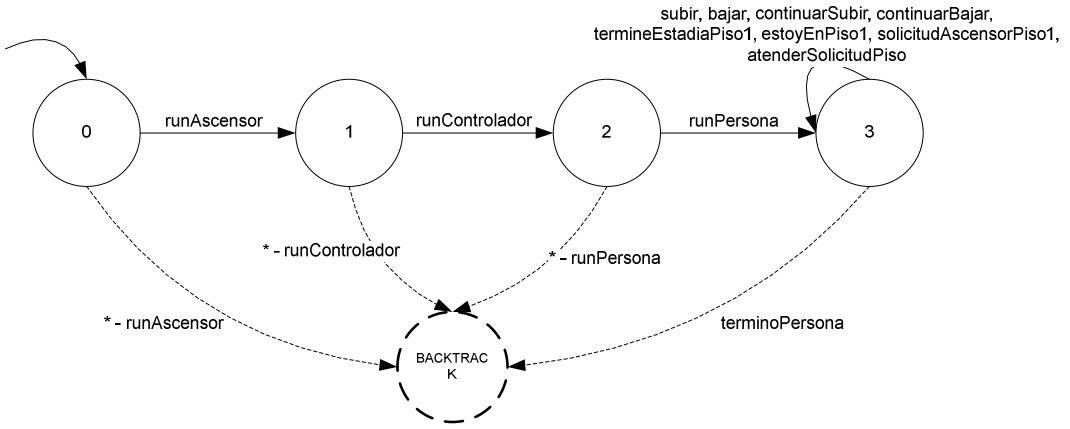

Figura 7.9: Representación gráfica del escenario de control utilizado para la cuarta propiedad del Controlador

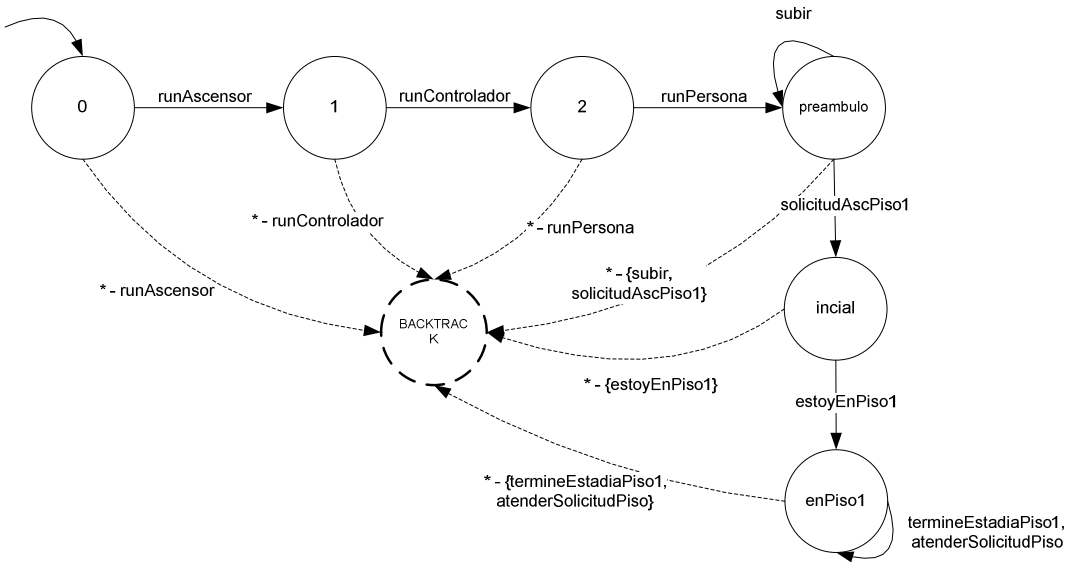

Figura 7.10: Representación gráfica del escenario de control mejorado utilizado para la cuarta propiedad del Controlador

El segundo escenario de control se generó a modo de prueba para mostrar que, si se restringe la búsqueda a los estados que generan exactamente los eventos en el orden que se esperan, se explora una única rama, recorriéndose una única vez los estados necesarios. Esto, obviamente se puede realizar al conocer de antemano (al haber verificado la propiedad con el primer escenario de control) que la propiedad fallará y que es factible recorrer esa única rama (la que se indica en el path to error generado por JPF en la primera verificación).

Utilizando el primer escenario de control, que tiene la misma estructura y restricciones que el escenario de control de las primeras tres propiedades, el tamaño del espacio de estados explorados se reduce aproximadamente a la mitad (de 14.654.429 a 6.947.795) para un modelo con 2 personas y 1 ascensor.

Mientras que con el segundo escenario (que explora directamente la rama que conduce a la violación de la propiedad) la cantidad de estados se reduce a 55 (valor incomparable con el total de estados).

7.3.2.5 Propiedad 5: "Si el ascensor está subiendo para atender una solicitud en el piso 4, no debe cambiar de dirección antes de atender dicha solicitud".

La propiedad se basa en la lógica de prioridades que debe cumplir el ascensor de no modificar el sentido (hacia arriba o hacia abajo) durante un viaje, si es que quedan solicitudes sin atender en la dirección en la que se está moviendo.

En forma similar a lo que se hizo en la propiedad 4, en este caso hubo que instrumentar el código para agregar algunos métodos que detecten la ocurrencia de la solicitud y atención al piso 4 (*solicitudAscensorPiso4* y *atenderSolicitudPiso4*).

Para representar la propiedad fue necesario generar un modelo en que la persona realice primero una solicitud al piso 4, luego una al 2 y por último al 1, para simular la condición en que un ascensor esté viajando en una dirección (hacia arriba), se detenga para atender una solicitud en el medio y, luego, haya que decidir si continuar para arriba o para abajo. De esta forma, al explorar la rama en que el ascensor está subiendo, llegando al piso 2, se atenderá dicha solicitud y debería continuar el viaje al piso 4 en vez de bajar hacia el piso 1.

La propiedad detectó un error en la lógica de priorización de la dirección del ascensor, en el código del controlador:

```
@Override
public void estoyEn(Ascensor a, int piso) {
	if (haySolicitudEn(a, piso)) {
		atenderSolicitudPiso(a, piso);

		/**
		 * BUG
		 * si esta bajando prioriza el atender solicitudes hacia
arriba y viceversa
		 */
		if (a.estaBajando()) {
			if (haySolicitudArriba(a, piso)) {
				a.continuarSubir();
			} else if (haySolicitudAbajo(a, piso)) {
				a.continuarBajar();
			}
		} else {
			if (haySolicitudAbajo(a, piso)) {
				a.continuarBajar();
			} else if (haySolicitudArriba(a, piso)) {
				a.continuarSubir();
			}
		}
	}

	if (piso==0) {
		if (haySolicitudArriba(a, piso))
			a.continuarSubir();
		else if (!a.estaParado())
			a.detenerse();
	}
	else if (piso==ALTURA) {
		if (haySolicitudAbajo(a, piso))
			a.continuarBajar();
		else if (!a.estaParado())
			a.detenerse();
	}
}
```

La antipropiedad para verificar esto está dada por el siguiente AFD:

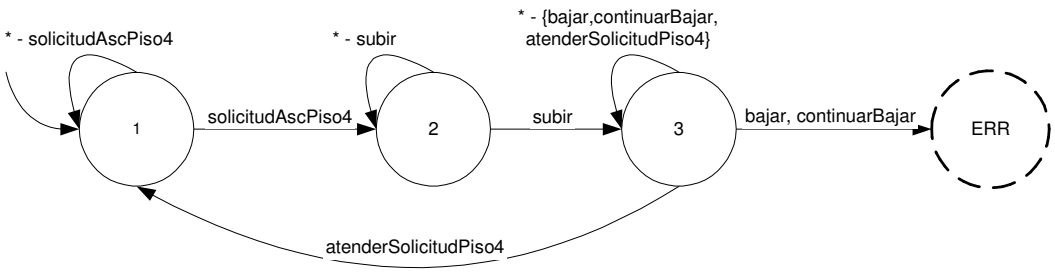

Figura 7.11: Representación gráfica de la quinta propiedad del Controlador

Para la verificación de esta propiedad se utilizó el siguiente escenario de control:

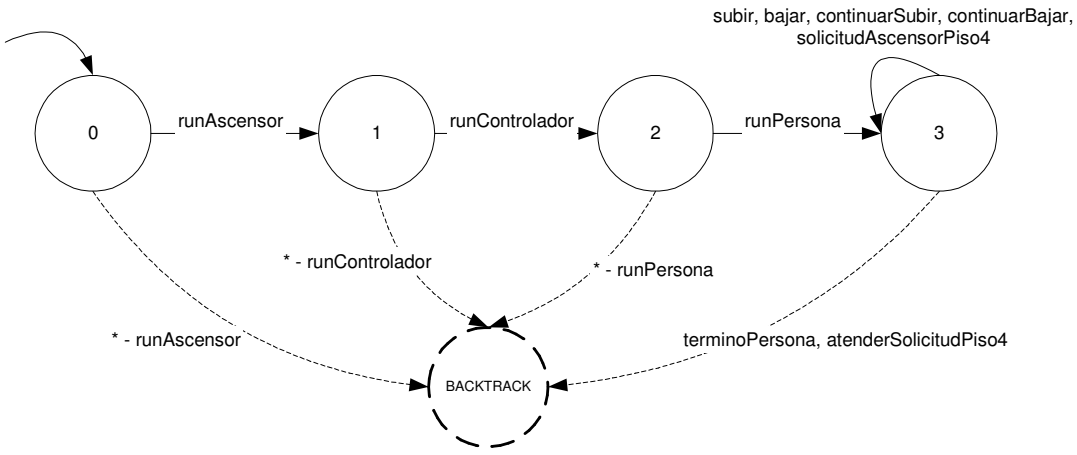

Figura 7.12: Representación gráfica del escenario de control utilizado para la quinta propiedad del Controlador

Este escenario de control es muy similar a los utilizados en las cuatro propiedades anteriores, sin embargo, en el estado 3 además poda la búsqueda en caso de que se produzca el evento *atenderSolicitudPiso4*. Esto se debe a que en el caso en que se atiende la solicitud del piso 4, no es relevante continuar la exploración de esa rama ya que se está buscando una violación de la propiedad antes de que se atienda dicho piso.

7.4 Resultados

Sobre el modelo y propiedades descritos anteriormente se realizaron verificaciones sobre 4 versiones del modelo de ascensores de distintas complejidad. La idea principal era evaluar el framework desarrollado con cierta diversidad de tamaño de problema.

- **Escenario 1 (1P1A)**: En este escenario se utiliza un ascensor y una persona que realiza 3 solicitudes.
- **Escenario 2 (2P1A)**: En este escenario se utiliza un ascensor y dos personas que realizan 3 solicitudes cada una.
- **Escenario 3 (2P2A)**: En este escenario se utilizan dos ascensores y dos personas (cada una operando con un ascensor) que realiza 3 solicitudes cada una.
- **Escenario 4 (3P2A)**: En este escenario se utilizan tres ascensores y tres personas (2 operando con un ascensor distinto y 1 realizando solicitudes para subir y bajar en diferentes pisos) que realizan 3 solicitudes cada una.

Las verificaciones de las distintas propiedades se ejecutaron sobre un equipo Pentium 4 con HT de 2.8Ghz, 512MB de memoria RAM asignada a la máquina virtual Java (versión 1.5.0_13), sobre Windows XP.

En la Tabla 7-1 se muestra, para cada uno de los escenarios planteados anteriormente, la duración de la verificación (utilizando JPF únicamente, sin el agregado de nuestro framework) medida en Horas, minutos y segundos y la cantidad de estados explorados. Estos números sirven de referencia para medir la performance y utilidad del framework desarrollado.

En la siguiente tabla no se especifican las propiedades ya que corresponde a la verificación estándar de JPF (que chequea sólo las propiedades default de la herramienta).

Escenario	Cantidad de Estados	Duración de la Prueba (H:M:S)	Resultado
1P1A	117.625	00:01:04	Terminó correctamente
2P1A	18.604.105	03:06:25	Terminó correctamente
2P2A	22.727.202	04:21:59	Reportó un error de OutOfMemory
3P2A	19.063.769	03:44:02	Reportó un error de Uncaught Exception

Tabla 7-1: Cantidad de estados y duración de la verificación utilizando sólo JPF

En la Tabla 7-2 se muestra, para la verificación de las propiedades 1 a 3 sobre cada uno de los escenarios planteados anteriormente, la duración medida en Horas, minutos y segundos para las siguientes variantes:

- Utilizando únicamente JPF
- Utilizando el framework desarrollado sin aplicar contexto (sólo para el escenario 1P1A).
- Utilizando el framework desarrollado y aplicando el contexto mencionado en la sección anterior

Esc	Prop	Duración			% overhead		Resultado
		JPF s/FWK	FWK Sin contexto	FWK con contexto	FWK sin contexto	FWK con contexto	
1P1A	1	00:01:04	00:02:22	00:01:38	222%	153%	Finalizado correctamente
	2	00:01:04	00:01:55	00:01:31	180%	142%	Finalizado correctamente
	3	00:01:04	00:02:01	00:01:37	189%	152%	Finalizado correctamente
2P1A	1	03:06:25	No Ejecutado	06:21:06	-	204%	Finalizado correctamente
	2	03:06:25	No Ejecutado	06:39:06	-	214%	Finalizado correctamente
	3	03:06:25	No Ejecutado	08:25:17	-	271%	Error: OutOfMemory
2P2A	1	04:21:59	No Ejecutado	00:39:21	-	15%	Error: OutOfMemory
	2	04:21:59	No Ejecutado	03:52:26	-	89%	Finalizado correctamente
	3	04:21:59	No Ejecutado	00:41:05	-	16%	Error: OutOfMemory
3P2A	1	03:44:02	No Ejecutado	05:32:11	-	148%	Finalizado correctamente
	2	03:44:02	No Ejecutado	02:15:35	-	61%	Error: OutOfMemory
	3	03:44:02	No Ejecutado	05:10:38	-	139%	Finalizado correctamente

Tabla 7-2: Duración de las pruebas (propiedades 1, 2 y 3)

En la Tabla 7-3 se muestra, para la verificación de las propiedades 1 a 3 sobre cada uno de los escenarios planteados anteriormente, la cantidad de estados explorados para las siguientes variantes:

- Utilizando únicamente JPF

- Utilizando el framework desarrollado sin aplicar contexto (sólo para el escenario 1P1A).
- Utilizando el framework desarrollado y aplicando el contexto mencionado en la sección anterior

Esc	Prop	# Estados			% overhead		Resultado
		JPF s/FWK	FWK Sin contexto	FWK con contexto	FWK sin contexto	FWK con contexto	
1P1A	1	117.625	360.083	266.002	306%	226%	Finalizado correctamente
	2	117.625	360.083	266.002	306%	226%	Finalizado correctamente
	3	117.625	360.083	266.002	306%	226%	Finalizado correctamente
2P1A	1	18.604.105	No Ejecutado	11.298.202	-	61%	Finalizado correctamente
	2	18.604.105	No Ejecutado	11.739.153	-	63%	Finalizado correctamente
	3	18.604.105	No Ejecutado	11.580.835	-	62%	Error: OutOfMemory
2P2A	1	22.727.202	No Ejecutado	9.675.630	-	43%	Error: OutOfMemory
	2	22.727.202	No Ejecutado	11.357.126	-	50%	Finalizado correctamente
	3	22.727.202	No Ejecutado	9.675.630	-	43%	Error: OutOfMemory
3P2A	1	19.063.769	No Ejecutado	12.306.228	-	65%	Finalizado correctamente
	2	19.063.769	No Ejecutado	12.835.358	-	67%	Error: OutOfMemory
	3	19.063.769	No Ejecutado	12.493.843	-	66%	Finalizado correctamente

Tabla 7-3: Cantidad de estados de las pruebas (propiedades 1, 2 y 3)

Los resultados obtenidos durante la verificación de las propiedades 4 y 5 no pueden compararse con JPF, ya que producen errores (se encuentra violación a las dos propiedades) y esto hace que se interrumpa la verificación y no se explore todo el espacio de estados. Por otro lado, sólo tienen sentido cuando se trabaja en un único ascensor.

En la Tabla 7-4 se muestra, para la verificación de las propiedades 4 y 5 sobre cada uno de los escenarios que utilizan un único ascensor, la duración medida en Horas, minutos y segundos para las siguientes variantes, hasta que se encuentra la violación de cada propiedad:

- Utilizando el framework desarrollado sin aplicar contexto
- Utilizando el framework desarrollado y aplicando el contexto mencionado en la sección anterior

Esc	Prop	Duración		Dif % (CC / SC)[4]
		sin contexto	con contexto	
1P1A	4	00:00:54	00:00:37	69%
2P1A	4	06:40:38	00:48:53	12%
1P1A	5	00:00:26	00:00:04	15%
2P1A	5	00:00:18	00:00:03	17%

Tabla 7-4: Duración de las pruebas (propiedades 4 y 5)

En la Tabla 7-5 se muestra, para la verificación de las propiedades 4 y 5 sobre cada uno de los escenarios que utilizan un único ascensor, la cantidad de estados explorados para las siguientes variantes:

- Utilizando el framework desarrollado sin aplicar contexto
- Utilizando el framework desarrollado y aplicando el contexto mencionado en la sección anterior.

Esc	Prop	# Estados		Dif % (CC / SC)[1]
		sin contexto	con contexto	
1P1A	4	103.628	66.499	64%
2P1A	4	14.654.429	6.947.795	47%
1P1A	5	50.852	3.493	7%
2P1A	5	50.852	3.493	7%

Tabla 7-5: Cantidad de estados de las pruebas (propiedades 4 y 5)

A partir de los resultados obtenidos, es posible realizar algunas observaciones:

- Las verificaciones de los escenarios de más de una persona y/o ascensor no se realizaron sin contexto debido a que la capacidad de procesamiento necesaria supera los recursos disponibles. Por ejemplo para el escenario 2P1A, se intentó realizar una verificación con el FWK que, luego de 18 horas continuó ejecutando sin producirse resultados. Por este motivo, se optó por no avanzar con estas pruebas.

[4] SC = Sin Contexto
CC = Con Contexto

- Las ejecuciones de los escenarios 2P2A y 3P3A mostradas en la Tabla 7-1 no pueden utilizarse para hacer comparaciones de performance con nuestro framework, ya que produjeron errores debidos a la falta de capacidad de la PC en la que se realizó la verificación.
- En todos los casos, la cantidad de estados explorados utilizando nuestro framework sin el contexto es superior a la utilizada por JPF. Es importante destacar que los estados internos generados por JPF son los mismos; la diferencia radica en que el framework fuerza a JPF a procesar estados ya visitados (teniendo en cuenta el estado de la propiedad).
- Analizando las tablas anteriores, puede verse que no existe una relación marcada entre los porcentajes de diferencia de duración y de cantidad de estados. Si bien esto último no es lo esperable, entendemos que se necesitarían mayor cantidad y diversidad de experimentos para establecer dicha relación. La causa de esto podría ser el hecho de que trabajar con los estados compuestos utilizados por nuestro framework sea mas dificultoso para el procesador, ya que se comparan estructuras de datos complejas y se hacen búsquedas sobre colecciones que no se encuentran optimizadas; mientras que en el caso de JPF, se utilizan estructuras optimizadas (por ejemplo, para comparar estados simplemente se comparan números).
- Con la excepción del escenario 1P1A, el contexto siempre permite disminuir la cantidad de estados explorados (aunque no necesariamente la duración de las pruebas). Esta anomalía puede deberse a que, dado que el modelo es el de menor complejidad (en términos de cantidad de threads), la cantidad de estados del escenario es demasiado pequeña y el contexto seleccionado para este caso no es lo suficientemente restrictivo.
- El uso de determinados contextos permitió disminuir significativamente el espacio de estados explorados en algunas de las pruebas. A modo de ejemplo, se pueden observar las diferencias al verificar la propiedad 5 con y sin contexto. Esto se debe a que el contexto guía la búsqueda de estados casi hasta el punto en el que se encuentra el problema.

7.5 Conclusiones

La idea de incluir un caso de estudio en el presente trabajo surge a partir de la necesidad de validar el framework desarrollado sobre un modelo de mayor complejidad a los utilizados durante el resto del trabajo.

Si bien los ejemplos utilizados durante el desarrollo del framework ayudaron en esta tarea, es importante mostrar una experiencia más cercana a la realidad.

En esta sección presentaremos algunas conclusiones importantes surgidas a partir del desarrollo del caso de estudio:

- En primera instancia es importante destacar que el framework funciona favorablemente. Permite definir propiedades y escenarios de control utilizando el lenguaje seleccionado en la sección 3.
- El poder expresivo del lenguaje de definición de propiedades y escenarios de control fue adecuado para la generación y ejecución de las pruebas desarrolladas. Como se menciona en 8.4.2, es un punto de partida para futuras extensiones al lenguaje (en términos de definición de eventos y complejidad de las propiedades).
- Se encontraron algunas falencias de usabilidad que están relacionadas principalmente con la forma de definir los eventos, las propiedades y los escenarios de control. Para versiones futuras sería recomendable proveer una alguna interfaz de usuario gráfica en lugar de los XMLs implementados hasta el momento.
- Se detectó una posible mejora que aumentaría muchísimo el poder de expresividad de las propiedades. Esta es, utilizar los parámetros de las invocaciones a los métodos dentro de las propiedades. En nuestro ejemplo, esto permitiría por ejemplo discriminar los eventos *solicitudAscensor* a distintos pisos, lo cual hubiera evitado la necesidad de instrumentar el código para definir la propiedad 4.
- Se vio claramente que es factible guiar la búsqueda de manera tal de controlar la explosión de estados que se produce durante la verificación. Si bien la utilización del Framework implica mayor uso de recursos (procesador y memoria), permite la posibilidad de acotar, tanto como se quiera, el conjunto de estados explorado en función de los escenarios de control.

Capítulo 8

Conclusiones generales y trabajo futuro

8.1 Introducción

En este capítulo se presentarán, en primera instancia, los resultados obtenidos durante el trabajo; luego, las conclusiones más importantes; y, por último, se describirán brevemente algunas posibilidades de investigación futura.

8.2 Resultados obtenidos

A partir del presente trabajo se obtiene un framework que permite, entre otras cosas, definir propiedades basadas en patrones de eventos sobre Java Path Finder y acotar la explosión de estados que éste produce durante la verificación.

Principalmente, el framework resuelve al usuario dos aspectos importantes a la hora de definir un modelo (y sus respectivas propiedades) utilizando JPF:

- Le brinda la posibilidad de definir propiedades basadas en patrones de eventos, escenarios de control y propiedades typestate.
- Le brinda un protocolo XML que ahorra al usuario trabajo de programación de los autómatas correspondientes a las propiedades y a los escenarios de control.

8.3 Conclusiones

El presente trabajo es esencialmente práctico. Los principales objetivos fueron: explorar las posibilidades de controlar la explosión de estados con la que los model checkers se enfrentan y permitir la definición de propiedades basadas en patrones de eventos. Para esto seleccionamos dos herramientas, SPIN y JPF, que cuentan con casos de éxito (Laboratorios Bell en el caso de SPIN y la NASA en el caso de JPF) y son utilizadas por otros equipos de investigación.

SPIN, que es un model checker orientado a modelos (y no a aplicaciones reales), tiene algunas virtudes que destacan sobre otros: el modelo que define el usuario el mismo modelo que la herramienta verifica; es muy declarativo y presenta una sintaxis simple y compacta; y por último, explora absolutamente todo el espacio de estados. No obstante, la tarea de extenderlo para lograr un mecanismo relativamente simple de reconocer Eventos y Patrones de Eventos se hizo extremadamente compleja. Hasta tal punto que se decidió dejar la búsqueda por este camino.

En [HOLZ03] el autor destaca que SPIN es esencialmente un model checker de estados, y que para predicar sobre eventos es necesario encontrar la manera de representar los predicados sobre estados en términos de eventos. Por otro lado, es una herramienta cerrada,

en el sentido que no provee mecanismos de extensión. Estos dos factores sumados, hicieron que la única alternativa viable para lograr nuestros objetivos fuera instrumentar demasiado el código, e inclusive así, no pudimos garantizar el éxito de los experimentos, lo que en definitiva nos impulsó a tomar la decisión de finalizar la investigación sobre esta herramienta sin llegar a un resultado satisfactorio.

JPF por su parte, es un model checker desarrollado en Java y explícitamente orientado a verificar programas Java ([VKBPL02]). Para los efectos de este trabajo, esto presentó una ventaja fundamental, que es que nos dio la libertad y la posibilidad de alterar el código según nuestras necesidades. Esto requirió comprender gran parte del diseño de la herramienta antes de poder empezar a definir la forma en que trabajaríamos.

El resultado final de esta tesis fue un framework que complementa a JPF; por un lado, proveyéndole una forma sencilla e intuitiva de definir propiedades basadas en patrones de eventos utilizando un lenguaje simple orientado a usuarios desarrolladores; y por el otro, permitiendo acotar arbitrariamente la explosión del espacio de estados sobre el cual se verifica utilizando exactamente el mismo lenguaje. Todo esto, sin alterar en lo más mínimo el proceso de verificación de propiedades empleado por JPF.

Este framework ha sido validado mediante el uso de un caso de estudio. Los resultados obtenidos sobre los experimentos realizados son alentadores, pero aún hace falta mayor cantidad de pruebas sobre una mayor cantidad y diversidad de problemas. Es importante destacar que el framework no fue optimizado, lo cual explica algunos de los problemas de performance encontrados durante la ejecución de los experimentos. Por otro lado, si bien no fue necesario llevar a cabo modificaciones sobre el framework durante las pruebas, se encontraron diversas posibilidades de mejoras que podrán ser llevadas a cabo por otros equipos de investigación (entre ellas, las optimizaciones antes mencionadas).

Tal vez la primera y más importante de las conclusiones a las que se llegó durante la elaboración de esta tesis, es que controlar la explosión del espacio de estados es factible; y gracias a esto se pueden flexibilizar de alguna manera los parámetros de calidad sobre los que se trabaja, ya que podemos evitar la exploración exhaustiva de estados, acotando los tiempos de las pruebas, pero aún así garantizando un mayor cubrimiento a menor costo que las actividades tradicionales de testing.

Este control puede lograrse, principalmente porque el framework permite encontrar un punto intermedio entre el testing manual y la exploración de todo el universo de estados.

En este sentido, creemos que este trabajo constituye un puntapié inicial. Existen otras maneras de acotar el espacio de estados que no fueron exploradas aún y que permitirán enriquecer el trabajo aquí comenzado. Es de esperar que en el futuro, este framework evolucione de forma de cubrir las necesidades de los investigadores y equipos de desarrollo que lo utilicen.

8.4 Trabajo futuro

Como dijimos en la sección anterior este es un trabajo esencialmente práctico, pero de carácter inicial. Tener un framework constituye un avance importante, sin embargo, aún quedan muchos aspectos sobre los cuales profundizar. A continuación se presentan algunos puntos de interés para el trabajo futuro.

8.4.1 Sobre las Propiedades TypeState

En Java, las interfaces, representan protocolos de comunicación entre objetos y no comportamiento. Por lo tanto, pueden existir clases que implementan distintas interfaces sin heredar o extender comportamiento de otras clases.

Un agregado importante al Framework, sería permitir al usuario definir propiedades TypeState sobre interfaces. Al momento del desarrollo del Framework, JPF no implementaba el método de reflexión *getIntefaces* de la clase *Class*. Este método es imprescindible para lograr este objetivo.

8.4.2 Sobre los tipos de eventos aceptados por el framework

En el presente trabajo, se logró con éxito implementar la verificación de patrones de eventos sobre un subconjunto predeterminado de instrucciones Java (bytecodes). Estos bytecodes corresponden al tipo de instrucciones de invocación de métodos (ej. INVOKEVIRTUAL).

Sin embargo, se podría extender el framework, concretamente las clases de especificación y reconocimiento de eventos, para que se puedan utilizar otro tipo de bytecodes y, por consiguiente, mayor variedad de tipos de eventos para la verificación (por ejemplo acceso a una variable dentro de un objeto, creación/destrucción de instancias, ejecución de una instrucción de decisión “If”, etc.). Con la adaptación para soportar más tipos de instrucciones, se podrían definir patrones de eventos de diversos tipos, lo que enriquecería la verificación.

Otra necesidad detectada durante la utilización del framework y relacionada con este aspecto es la de tener en cuenta los parámetros del método sobre el que se define un evento. Esto se ve claramente durante la elaboración del caso de estudio, donde para determinadas propiedades, fue necesario instrumentar el código agregando métodos que se invocan condicionalmente en función de los valores de los parámetros (por ejemplo el número de piso).

8.4.3 Sobre la exploración de estados

El tamaño del espacio de estados a recorrer durante la verificación es uno de los mayores problemas que tienen los model checkers. Durante el presente trabajo se implementaron dos mecanismos que permiten acotar el espacio de estados: Preámbulo y Contexto de Ejecución.

No obstante, existen otras alternativas que no fueron exploradas. En particular, se podrían filtrar los estados recorridos utilizando predicados sobre los objetos del modelo. De esta manera, se podría indicar a JPF que cuando se llega a un estado en el que algún objeto O cumple con un predicado P, se haga backtrack.

8.4.4 Sobre el espacio de estados

Como se ha mencionado anteriormente, JPF implementa algunas optimizaciones en la generación de estados, lo cual provoca que haya algunas opciones de interleaving entre threads que no se evalúen durante la verificación.

Ésta no es una traba para verificaciones sobre programas Java, pero sí lo sería si se quisieran ejecutar verificaciones sobre modelos (y no sobre aplicaciones reales escritas en código Java).

En algunos casos, esto puede ser una limitación. Una mejora posible sería modificar la lógica de generación de estados para que ante cada ejecución de cada bytecode, se genere un estado nuevo. Esto permitiría evaluar todas las opciones de interleaving. Es importante destacar que esto implicaría un costo mucho más elevado a la hora de ejecutar las verificaciones.

8.4.5 Sobre el retraso en la verificación de la propiedad

Debido al diseño actual de JPF y las adaptaciones incorporadas en el presente trabajo, la interacción entre la búsqueda y el *Listener* se realiza siempre luego de la ejecución completa de un estado de la VM (JPF). Este procesamiento del estado involucra la invocación de un conjunto de instrucciones (bytecodes) y, en consecuencia, eventos (observables o no).

Por lo tanto, durante la ejecución de un estado de la VM, podrán producirse muchas transiciones en 1 o más AFDs que estén verificando cada patrón de eventos. Debido a que no se alteró la lógica de la herramienta, la verificación de las propiedades se realizará únicamente al completarse la ejecución del estado actual. Esto produce un gap entre el momento en que se viola una propiedad (o preámbulo, o contexto) y el momento en que se interrumpe el camino actual de verificación.

La oportunidad de mejora en este caso, es rediseñar la interacción entre la búsqueda/exploración de estados de JPF y el framework de verificación de patrones de eventos (coordinador de los AFDs) para que se pueda consultar en forma online, es decir luego de la ejecución de cada bytecode, el estado de las propiedades. Esta modificación, si bien es factible, afecta en gran parte a las clases que realizan la ejecución de un lote de instrucción de JPF (bytecodes), por lo tanto, implica realizar un rediseño completo en la interacción de un conjunto significativo de clases.

8.4.6 Sobre la verificación de múltiples propiedades

Para poder ampliar el potencial de la verificación de múltiples propiedades en forma concurrente, se podrían plantear las siguientes mejoras al framework sobre JPF:

a. permitir que se pueda definir y verificar más de una Propiedad Global

b. permitir que, para una misma clase, se pueda definir más de una Propiedad *Typestate*

Para el primer punto no es necesario modificar la lógica de verificación, ya que alcanza con extender la funcionalidad del verificador para administrar varias propiedades globales.

Para el segundo punto, tampoco es necesario modificar la lógica de verificación, pero agrega la complejidad de definir la lógica para determinar el conjunto de propiedades pertinentes, en el momento de la creación de un objeto en particular, según la clase desde la cual se está definiendo la instancia. Debido a que las propiedades *Typestate* se pueden definir no sólo para clases concretas sino también para las abstractas y para distintos niveles de la jerarquía de clases, es necesario extender la funcionalidad de exploración completa de la rama a la que pertenece el tipo de la instancia u objeto creado. Esta extensión se podría implementar agregando dicha complejidad en las clases del framework.

El principal aporte de esta modificación está relacionado con la definición de propiedades y su complejidad. Por ejemplo, se podrían definir n propiedades Typestate para cada instancia que se genere de tipo Canal, permitiendo de esta forma dividir una única propiedad, que puede ser extensa y compleja (con muchos estados en el AFD), en distintas propiedades individuales.

Capítulo 9

Apéndices

9.1 Descripción técnica del framework desarrollado

9.1.1 Introducción

El presente apéndice tiene como objetivo detallar los aspectos de implementación más importantes del desarrollo del framework para verificación de patrones de eventos. Dichos aspectos están relacionados con el diseño técnico de las clases propias que componen el framework, las modificaciones en el código de JavaPathFinder, aspectos específicos de implementación y una guía técnica de requerimientos y de uso para la verificación de propiedades.

9.1.2 Diseño

La premisa principal sobre la cual se basó el diseño de nuestro framework fue minimizar la cantidad y complejidad de modificaciones realizadas sobre el código de JPF. Se construyó el modelo de clases siguiendo los lineamientos del patrón de diseño *Mediator* ([GHJV94]), que tiene como principal objetivo minimizar la cantidad de interfaces y comunicación entre las diferentes entidades del modelo.

En el caso de JPF, las entidades más relevantes para la verificación de patrones están relacionadas con los algoritmos de búsqueda, los *Listeners* y la información propia de la JVM para inspeccionar atributos de cada instancia (objeto) que participa en la verificación del modelo. Con este escenario, se definió una clase principal llamada *Coordinador* que es la encargada de manejar la interacción con las diferentes entidades de JPF y las entidades auxiliares del framework (*Eventos*, *AFDs*, *Builders*, etc.). Por otra parte, el *Coordinador* también es el que contiene la lógica de seguimiento y administración de las propiedades (AFDs) de la verificación.

9.1.2.1 Glosario de las entidades del framework

- *Coordinador*: es la clase principal. Contiene la lógica central del framework. Recibe todas las notificaciones por parte del Listener (inicio/fin de la verificación, ocurrencia de instrucciones, creación / destrucción de objetos, etc.) y se encarga de administrar toda la información relacionada con las propiedades activas, contéxto/preámbulo, caminos de estados de las mismas, realizar backtracking, etc.
- *AutomataVerificacion*: Se utiliza para la representación de las propiedades de patrones de eventos (ya sean Global o TypeState). Cada instancia se genera a partir de la definición de una propiedad, es decir, de un AFD. A partir de la

estructura del AFD permite, dado un evento, consumirlo o no, es decir cambiar el estado actual según las transiciones definidas para el evento ocurrido.

- AutomataVerificacionVacio: es una especialización de AutomataVerificacion. Se utiliza para representar propiedades que no tienen definición en XML.
- ContextoBusqueda: es la clase análoga a AutomataVerificacion, con la diferencia que implementa la lógica particular de un Contexto, es decir, que si ocurre un Evento no esperado o válido para el contexto se toma como una violación al mismo.
- ContextoValidoBusqueda: especialización de ContextoBusqueda, se utiliza solamente para representar contextos vacíos, es decir, sin restricciones para la verificación.
- DFSearchTesis: esta clase, que hereda de gov.nasa.jpf.search.Search, se diseñó con el objetivo de adaptar la lógica de exploración de los estados del árbol de JPF sin alterar el código original.
- EventBuilder: esta clase da soporte a la construcción de eventos, a partir de la ejecución de una instrucción (definido en el XML, construye instancias a partir de instrucciones).
- Evento: es la clase que representa los diferentes eventos que ocurren en la verificación. No implementa comportamiento, sólo se utiliza como abstracción de otros tipos predefinidos (int, String, etc.).
- Listener: hereda de PropertyListenerAdapter e implementa la interfaz JPFListener. Es el nexo con el motor de JPF para poder determinar la ocurrencia de determinados sucesos (eventos, creación de objetos, etc.) y que permite los estímulos necesarios para modificar los estados de los AFDs que representan las propiedades.
- PropertyTemplate: representa la definición de una propiedad Global, es decir, es la base para la creación de un AFD que representará la instancia de la propiedad. Se genera a partir del XML correspondiente a la GlobalProperty.
- TypeStatePropertyTemplate: es una especialización de PropertyTemplate para contemplar el Type asociado a la propiedad. Se utiliza para la construcción de AFDs que representan TypeStateProperty. El Type es utilizado posteriormente en la generación de los resultados relacionados con la propiedad que ocasionó el fallo en la verificación.
- State: esta entidad representa los estados en un AFD. Se diseñó como abstracción de un tipo predefinido. Implementa los métodos para poder comparar los valores de diferentes instancias.

- Transición: cada instancia representa una transición en el AFD correspondiente a una propiedad
- XMLReader: implementa las operaciones básica de lectura de un XML, para obtener los valores de los atributo. Importa org.dom4j.Attribute
- XMLAFDReader: especialización de XMLReader. Contiene la lógica para leer e interpretar la estructura de los AFD (propiedades) dentro de un XML.
- XMLContextoBusquedaReader: ídem para Contexto / Preámbulo.
- XMLEventBuilderReader: ídem para EventBuilder.

9.1.2.2 Modelo de ejecución de la verificación

En esta sección se describe, mediante lenguaje UML, la interacción entre las clases principales del Framework y JPF durante la ejecución de una verificación. El objetivo es representar los diferentes actores y su interacción, desde que comienza la verificación hasta la detección de la violación de una propiedad.

El proceso completo correspondiente a la verificación está dividido en tres diagramas de secuencia, cada uno de los cuales representa un nivel de abstracción diferente.

La Figura 9.1 representa la interacción entre la clase *VerificationLauncher* y las principales entidades del framework para configurar los objetos que intervendrán en la verificación de propiedades (*coordinador*, *listener* y búsqueda). Finalmente, se invoca al método *run* de JPF para comenzar con el proceso propiamente dicho.

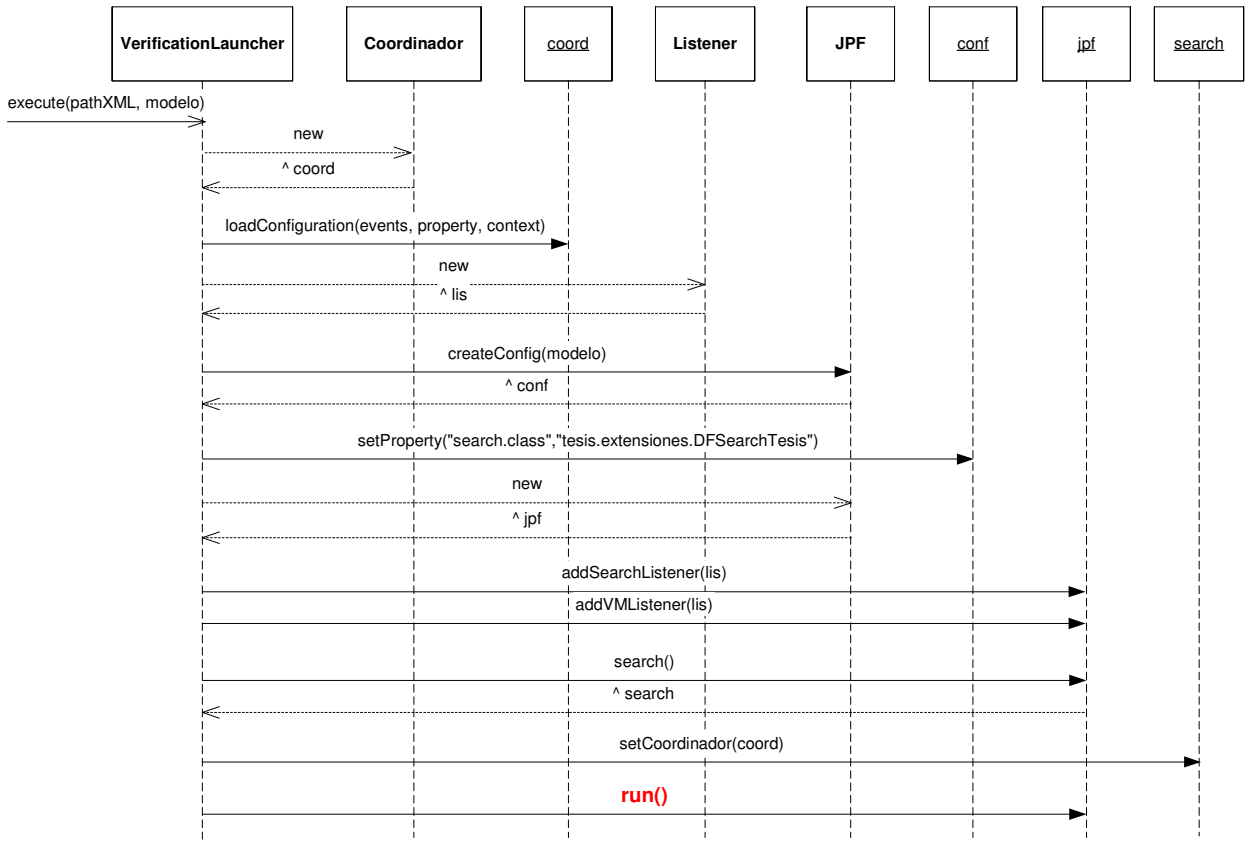

Figura 9.1: Lanzamiento de la verificación

En la Figura 9.2 se representa la ejecución del proceso de verificación a nivel general. Se puede notar que la lógica de exploración de estados está contenida en la clase *search* (*DFSearchTesis*) que es la que se encarga de iterar sobre las ramas del árbol y realizar el backtracking. La búsqueda interactúa directamente con el *coordinador* para decidir cuándo backtrackear (que analizará en función del estado compuesto de JVM y AFDs) y cuándo continuar con la ejecución de la rama.

De este diagrama se puede destacar que el chequeo del estado de la verificación (válida / propiedades violadas) se realiza exclusivamente luego de un forward (en el método *hasPropertyTermination*). Esto respeta el diseño original del model checker.

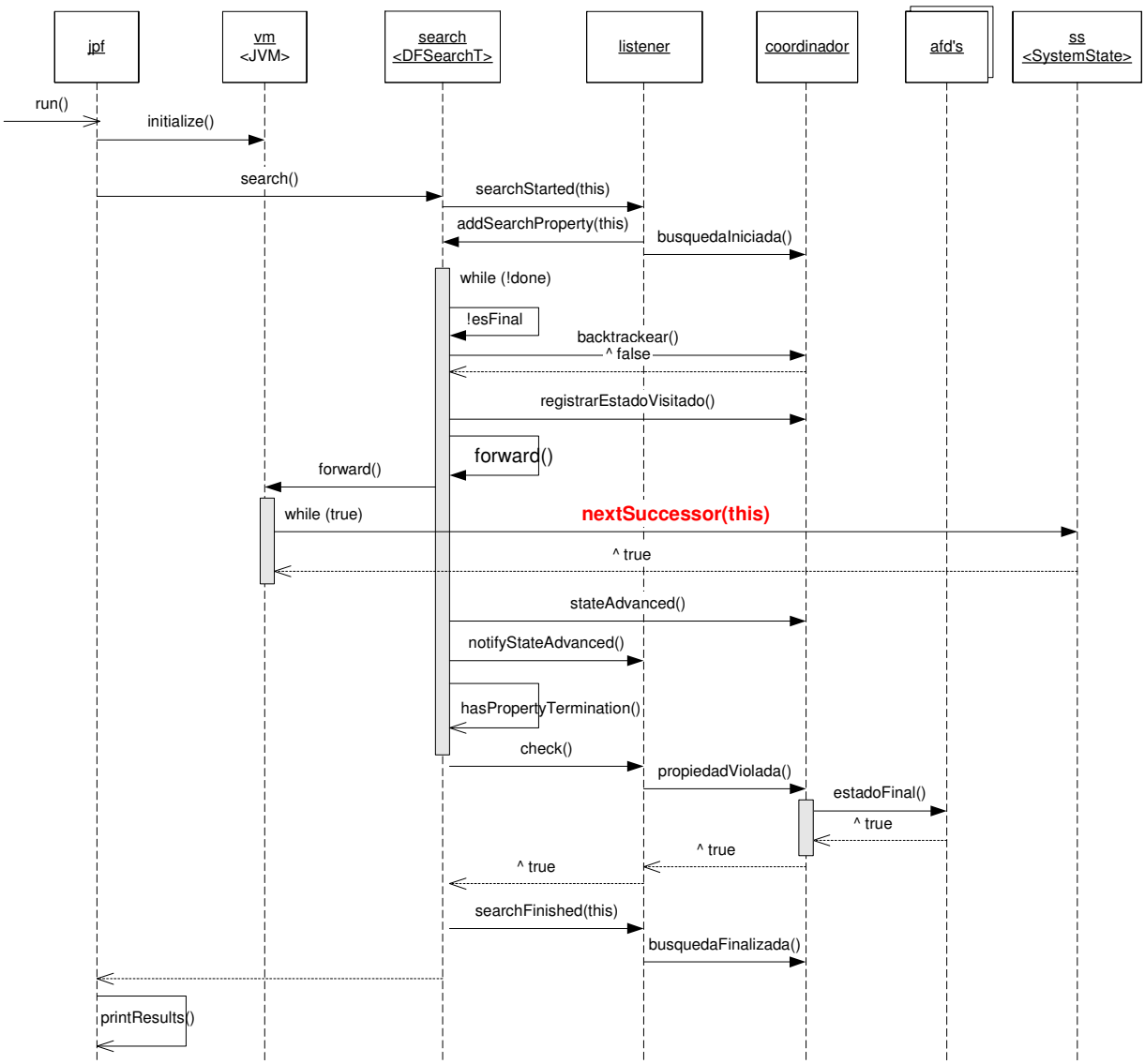

Figura 9.2: Ejecución (alto nivel) de la verificación

Por último, la Figura 9.3, muestra la interacción entre el núcleo de JPF, compuesto principalmente por la JVM, los *System* y *Kernel States* y *ThreadInfo*, y el *coordinador/listener* que son los encargados de verificar y alterar el estado de las propiedades (AFDs) luego de cada ejecución de un bytecode. Se puede apreciar en el diagrama, la interacción que se produce al realizar un ***forward*** sobre un estado del árbol y la ejecución del conjunto de instrucciones (bytecodes) definidos para dicho nodo.

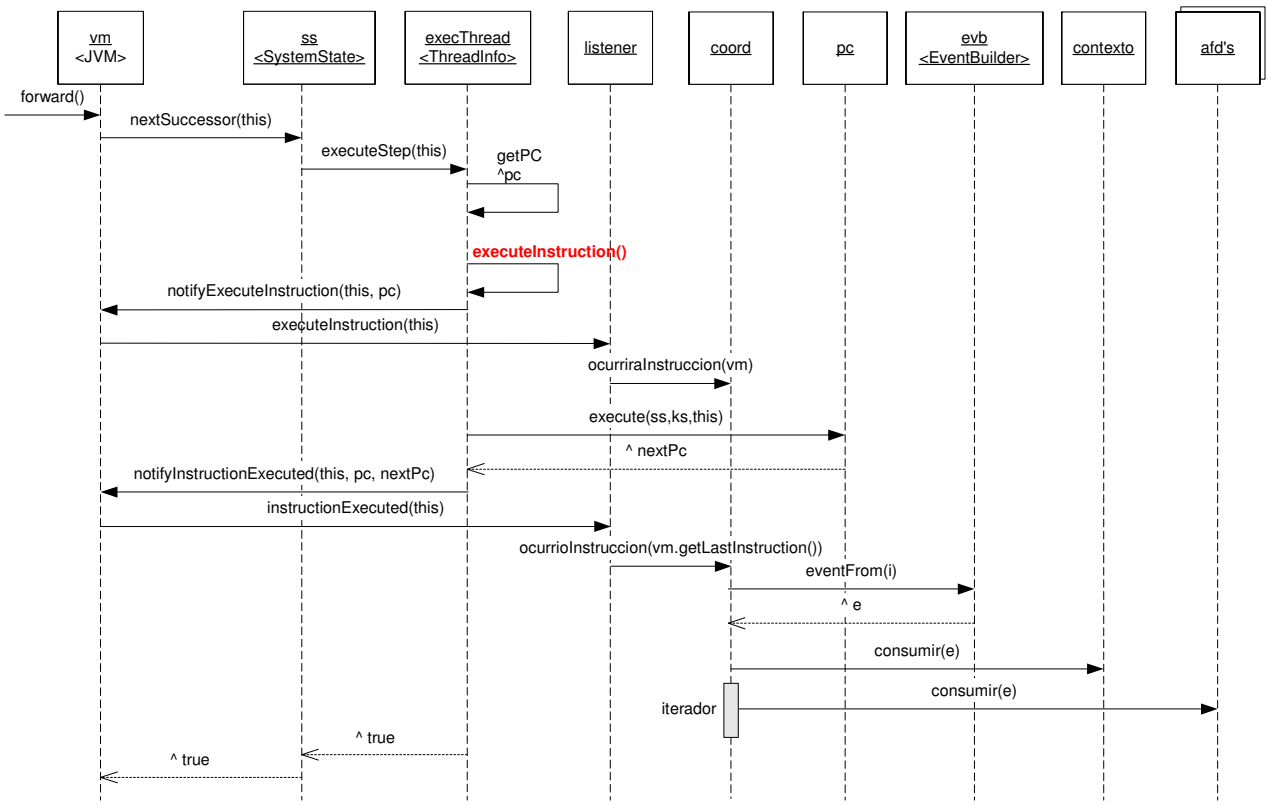

Figura 9.3: Ejecución del forward (en la JVM)

9.1.3 Modificaciones a JPF

Durante el desarrollo del framework fue necesario modificar un conjunto reducido de clases propias de JPF. En su mayoría, las modificaciones están relacionadas con la visibilidad entre las clases y la declaración de métodos.

A continuación se detallan estos cambios:

- **Clase *gov.nasa.jpf.search.Search***
  - Se modificó la declaración del método *checkStateSpaceLimit* por *protected boolean checkStateSpaceLimit*, con el fin de poder especializar *Search* (*DFSearchTesis*) en forma externa al package JPF
- **Clase *gov.nasa.jpf.jvm.bytecode.VirtualInvocation***
  - Se modificó la declaración del método *getCalleeThis (ThreadInfo ti)* por *public int getCalleeThis (ThreadInfo ti)*, con el fin de poder obtener desde el framework la referencia a la instancia que es afectada por el evento ocurrido (objeto al cual se invocó el método).
  - Se modificó la declaración del método *getCalleeClassInfo (KernelState ks, int objRef)* por *public ClassInfo getCalleeClassInfo (KernelState ks, int objRef)*, con el fin de poder obtener la clase a la cual pertenece la instancia que es afectada por el evento ocurrido.

9.1.4 Requerimientos y utilización del framework

Para poder ejecutar verificaciones sobre el Framework integrado con JPF, es necesario proveerle al mismo la información necesaria con respecto a la aplicación Java del usuario y las propiedades basadas en eventos a verificar.

Para especificar las propiedades, se definió una interfaz a través de archivos XMLs que deben ser completados con la estructura correspondiente, según el propósito de cada uno. De esta forma, un lote de verificación de propiedades está comprendido por los siguientes archivos:

Events.xml: contiene la definición de los eventos, es decir de qué tipo son, el nombre por el cual se identifican y su asociación con una etiqueta.

La estructura del XML respeta la siguiente especificación:

```
<ProblemDefinition>
<events>
    <event type="invoke" name="<nombre método>" label="<label>"
/>
      ...
      ...
</events>
</ProblemDefinition>
```

ProblemContext.xml: contiene la especificación de la propiedad de contexto de la verificación. Como se explica en la sección 5, esta propiedad puede verificarse en dos diferentes formas: Contexto o Preámbulo. La propiedad se describe en términos de un AFD, es decir, detallando los estados y transiciones del mismo:

```
<ProblemDefinition>
<SearchContext mode="<contexto|preámbulo>">
    <states>
        <state label="<label>" start="<true|false>"
final="<true|false>"/>
          ...
          ...
    </states>
    <transitions>
        <transition from="<state>" to="<state>"
labelEvent="<evento>" />
          ...
          ...
    </transitions>
</SearchContext>
</ProblemDefinition>
```

ProblemProperty.xml: contiene la especificación de las propiedades que serán verificadas. En este archivo se pueden definir una propiedad global (*GlobalProperty*) y una o más propiedades *TypeState*. Se define de forma similar a la propiedad de contexto de la verificación conteniendo, lógicamente, un AFD por cada propiedad:

```
<ProblemDefinition>
<GlobalProperty>
    <states>
        <state label="<label>" start="<true|false>"
final="<true|false>"/>
            ...
            ...
    </states>
    <transitions>
        <transition from="<state>" to="<state>"
labelEvent="<evento>" />
            ...
            ...
    </transitions>
</GlobalProperty>

<TypeStateProperty class="<className>">
    <states>
        <state label="<label>" start="<true|false>"
final="<true|false>"/>
            ...
            ...
    </states>
    <transitions>
        <transition from="<state>" to="<state>"
labelEvent="<evento>" />
            ...
            ...
    </transitions>
</TypeStateProperty>
</ProblemDefinition>
```

La estructura de los XMLs es bastante simple y sólo contiene la información indispensable para poder configurar los eventos (identificadores), propiedades (bajo estructura

de AFD, con estados y transiciones) y contexto para la ejecución de la verificación sobre una aplicación.

Para ejecutar la verificación, es necesario incluir, preferentemente en una clase estática, el package `tesis.extensiones.*`. La verificación es configurada y ejecutada a través del método:

```
VerificationLauncher.execute(pathXML, nombreClaseMain);
```

Donde pathXML es la ruta donde se encuentran los 3 archivos XMLs (ej: "`./src/tesis/Ejemplo /`") y nombreClaseMain es la ruta completa a la clase principal de la aplicación a ser verificada (ej: "`tesis.Ejemplo.Modelo`").

9.2 Ejemplo completo de selección de líder

A continuación se muestra el ejemplo completo del algoritmo de selección de líder presentado en la sección 6.3, implementado en SPIN con la instrumentación de código.

```
#define N   5
#define I   3
#define L   10

#define ie InicioEjecucion==1
#define nw MensajeNW==1
#define w MensajeW==1

#define states(newIE,newNW,newW) atomic{ InicioEjecucion = newIE; MensajeNW = newNW;
MensajeW = newW;}
#define clearstates() states(0,0,0)

bit InicioEjecucion;
bit MensajeNW;
bit MensajeW;


mtype = { one, two, winner };
chan q[N] = [L] of { mtype, byte};

byte gWinner;

proctype node (chan in, out; byte mynumber)
{  bit Active = 1, know_winner = 0;
   byte nr, maximum = mynumber, neighbourR;

   xr in;
   xs out;

   printf("MSC: %d\n", mynumber);
   out!one(mynumber);
skip;
end:  do
   :: in?winner, nr ->
      states(0,0,1);
      out!winner, nr;
      break;
   :: in?one(nr) ->
      if
      :: Active ->
         if
         :: nr > mynumber ->
```

```
            printf("MSC: LOST\n");
            Active = 0;
            states(0,1,0);
            out!one, nr;
         :: nr == mynumber ->
            assert(nr == N);
            states(0,0,1);
            out!winner, mynumber;
            printf("MSC: LEADER\n");
            break;
         :: nr < mynumber ->
            states(0,1,0);
            out!one, mynumber;
         fi
      :: else ->
         states(0,1,0);
         out!one, nr;
      fi
   od
}

active proctype monitor (){
T0_init:
    if
    :: ie -> goto L_NW_INICIADO
    :: goto T0_init
    fi;
L_NW_INICIADO:
    if
    :: (w) -> goto trampa
    :: (nw) -> goto L_NW1
    :: goto L_NW_INICIADO
    fi;
L_NW1:
    if
    :: (w) -> goto accept_all
    :: goto L_NW1
    fi;
trampa:
    do
    :: true -> skip
    od;
L_WIN:
accept_all:
    skip;
}

init {
   byte proc;
   states(1,0,0);
   atomic {
      proc = 1;
      do
      :: proc <= N ->
         run node (q[proc-1], q[proc%N], (N+I-proc)%N+1);
         proc++
      :: proc > N ->
         break
      od
   }
}
```

Bibliografía

HOLZ03	Gerard J. Holzmann. Spin Model Checker, The: Primer and Reference Manual, Addison-Wesley, 2003.
HOLZ97	Gerard J. Holzmann. The Model Checker SPIN, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 23, NO. 5, 1997
SHE01	Margaret H. Smith, Gerard J. Holzmann, Kousha Etessam. Events and Constraints: A Graphical Editor for Capturing Logic Requirements of Programs, Proceedings of the 5th IEEE International Symposium on Requirements Engineering, p.14-22, August 27-31, 2001
BKO05	Victor Braberman, Nicolas Kicillof, and Alfredo Olivero. Scenario-Matching Approach to the Description and Model Checking of Real-Time Properties, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 31, NO. 12, DECEMBER 2005
HOLZ04	Gerard J. Holzmann. An Analysis of Bitstate Hashing, Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.
GPVW95	R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper, “Simple On-The-Fly Automatic Verification of Linear Temporal Logic,” Proc. IFIP/WG6.1 Symp. Protocol Specification, Testing, and Verification (PSTV95), pp. 3-18, Warsaw, Poland, Chapman & Hall, June 1995.
PNU77	A. Pnueli, “The Temporal Logic of Programs,” Proc. 18th IEEE Symp. Foundations of Computer Science, Providence, R.I., pp. 46-57, 1977
GH93	P. Godefroid, and G.J. Holzmann, “On the Verification of Temporal Properties,” Proc. IFIP/WG6.1 Symp. Protocol Specification, Testing, and Verification (PSTV93), pp. 109-124, Liege, Belgium, North-Holland, June 1993
VKBPL02	W. Visser, K. Havelund, G. Brat, S. Park and F. Lerda, “Model Checking Programs”, April 19, 2002
BCHJM04	D. Beyer, A.J. Chlipala, T. A. Henzinger, R. Jhala, R. Majumdar, “The BLAST Query Language for Software Verification”, 2004
GHNS93	J. Grabowski, D. Hogrefe, R. Nahm, A. Spichiger, “Relating Test Purposes to Formal Specifications: Towards a Theoretical Foundation of Practical Testing”, 1993
JPF01	http://javapathfinder.sourceforge.net/
GHJV94	E. Gamma, R. Helm, R. Johnson, J.M. Vlissides., “Design Patterns: Elements of Reusable Object-Oriented Software”, Oct 31, 1994
EH00	Etessami, K. and G.J. Holzmann. [2000]. "Optimizing Büchi automata." Proc. Proceedings of 11th Int. Conf. on Concurrency Theory. Aug. 2000. Springer Verlag, LNCS 1877, pp. 153–167.
EWS01	Etessami, K., T. Wilke, and R. Schuller. [2001]. "Fair simulation relations, parity games, and state space reduction for Büchi automata." Proc. 28th Int. Col. on Automata, Languages, and Programming. Springer Verlag, LNCS 2076, pp. 694–707.
ABK04	Alfonso A., Braberman V., Kicillof N.: “Visual Timed Event Scenarios”

FAG76	Fagan, M.E.,[1976] “Design and Code inspections to reduce errors in program development”, 1976, IBM Systems Journal, Vol. 15, No 3
USD02	U.S. Department of Commerce, [2002] “The Economic Impacts of Inadequate Infraestructure for Software Testing”
MYE79	Myers G. J. [1979]. “The art of Software Testing”
MCC04	McConnell S. C, [2004], “Code Complete 2nd Edition”, Microsoft Press

Referencias

11.1 Capítulos

Capítulo 1 5
Capítulo 2 10
Capítulo 3 15
Capítulo 4 21
Capítulo 5 31
Capítulo 6 46
Capítulo 7 61
Capítulo 8 77
Capítulo 9 82

11.2 Figuras

Figura 2.1: Modelo de Ejecución de una herramienta de *Model checking* 10
Figura 2.2: Posibles ejecuciones de dos procesos con dos instrucciones c/u 12
Figura 2.3: Posibles estados de la ejecución de dos procesos con dos instrucciones 13
Figura 3.1: Algunos posibles patrones de eventos del modelo del Ejemplo 1 16
Figura 3.2: Representación gráfica de la Propiedad del Ejemplo 1 18
Figura 3.3: Representación gráfica del preámbulo del Ejemplo 1 18
Figura 3.4: Representación gráfica del contexto del Ejemplo 1 19
Figura 4.1: Diagrama de secuencias de la arquitectura de JPF 23
Figura 4.2: Pseudo código del algoritmo DFS de búsqueda 24
Figura 4.3: Esquema de instrucciones generadoras de estados en JPF 26
Figura 4.4: Interfaz Property 27
Figura 4.5: Interfaz VMListener 28
Figura 4.6: Interfaz SearchListener 28
Figura 4.7: Ejemplo de PropertyListenerAdapter 28
Figura 5.1: Modelo de Ejemplo de un Canal compartido por múltiples Threads 34
Figura 5.2: AFD para la verificación de 3 WRITEs 34
Figura 5.3: Árbol de estados, sin contemplar estado de la property (AFD) 35
Figura 5.4: Árbol de estados, contemplando estado de la property (AFD) 36
Figura 5.5: AFD para la verificación de 3 WRITEs sin CLOSEs intermedios 38
Figura 5.6: Contexto para restringir CLOSE 38
Figura 5.7: AFD para la verificación del correcto uso de un Canal 39
Figura 5.8: Modelo de Ejemplo para comparar GlobalProperty vs. TypestateProperty 40
Figura 5.9: Global Property vs. Typestate Property 40
Figura 5.10: Modelo de Ejemplo para comparar GlobalProperty vs. TypeStateProperty 41
Figura 5.11: Global Property vs. Typestate Property (2) 41
Figura 5.12: Jerarquía de clases de Canal 43
Figura 6.1: Ejemplo de uso de atomic 47
Figura 6.2: Ejemplo de uso de if 48
Figura 6.3: Ejemplo de uso de do 48

Figura 6.4: Ejemplo de uso de end-state 49
Figura 6.5: Ejemplo de uso de progress-state 49
Figura 6.6: Ejemplo de uso de never claim 50
Figura 6.7: Modelo de selección de líder 52
Figura 6.8: Variables definidas para la observación de eventos 53
Figura 6.9: Instrucciones definidas para la instrumentación de código 53
Figura 6.10: Instrumentación realizada usando instrucciones auxiliares 53
Figura 6.11: Diagrama de la propiedad del Ejemplo 6 55
Figura 6.12: Proceso monitor que representa la propiedad 56
Figura 6.13: Diagrama de la propiedad "no se eligen dos winners" 57
Figura 6.14: Never Claim que representa la propiedad "No se eligen 2 Winners" 57
Figura 6.15: Código de la Figura 6.10 agregando la limpieza de los flags 58
Figura 6.16: Código de la Figura 6.14 agregando la limpieza de los flags 59
Figura 6.17: Test And Set de una expresión 59
Figura 7.1: Protocolo de comunicación del Controlador 62
Figura 7.2: Diagrama de secuencias del controlador 63
Figura 7.3: Representación gráfica de la primera propiedad del Controlador 64
Figura 7.4: Representación gráfica del escenario de control utilizado para la primera propiedad del Controlador 65
Figura 7.5: Representación gráfica de la segunda propiedad del Controlador 65
Figura 7.6: Representación gráfica del escenario de control utilizado para la segunda propiedad del Controlador 66
Figura 7.7: Representación gráfica de la tercera propiedad del Controlador 66
Figura 7.8: Representación gráfica de la cuarta propiedad del Controlador 67
Figura 7.9: Representación gráfica del escenario de control utilizado para la cuarta propiedad del Controlador 68
Figura 7.10: Representación gráfica del escenario de control mejorado utilizado para la cuarta propiedad del Controlador 68
Figura 7.11: Representación gráfica de la quinta propiedad del Controlador 70
Figura 7.12: Representación gráfica del escenario de control utilizado para la quinta propiedad del Controlador 71
Figura 9.1: Lanzamiento de la verificación 85
Figura 9.2: Ejecución (alto nivel) de la verificación 86
Figura 9.3: Ejecución del forward (en la JVM) 87

11.3 Ejemplos

Ejemplo 1: Un modelo abstracto 16
Ejemplo 2: Adaptación del algoritmo de Búsqueda de estados 33
Ejemplo 3: Restricción de las ramas, mediante el uso de Contexto de Verificación 38
Ejemplo 4: Jerarquía de clases de Canal 42
Ejemplo 5: Elección de líder en un anillo unidireccional 51
Ejemplo 6: Elección de líder en un anillo unidireccional (continuación) 55
Ejemplo 7: Elección de líder en un anillo unidireccional (usando never claims) 57