Verificación Modular de Atomicidad en Bytecode Java Multi-Thread *

Francisco Bavera
Departamento de Computación,
Universidad Nacional de Río Cuarto
Río Cuarto, Argentina
pancho@dc.exa.unrc.edu.ar

Resumen

En este trabajo se presenta una técnica para verificar modularmente atomicidad de programas bytecode Java multi-thread. Los programas deben contar con una especificación referente a los bloqueos y al acceso a los recursos compartidos para realizar la verificación modular. Se presenta la compilación propuesta de programas fuente Java con la especificacion de atomicidad a bytecode Java, con dichas especificaciones incluídas el código compilado. Garantizar atomicidad en programas mult-thread permite, posteriormente, verificar propiedades funcionales más complejas utilizando técnicas para programas secuenciales.

Palabras Clave: Seguridad, Código Móvil Seguro, Concurrencia, Teoría de Lipton

*Este trabajo ha sido realizado en el marco de proyectos subsidiados por la SECyT de la UNRC y por la Agencia Córdoba Ciencia.

1 Introducción

El avance tecnológico ha integrado mecanismos computacionales que comprenden hardware y software a los más diversos artefactos. Cada vez más se busca que estos mecanismos hagan uso de la facilidad de adaptación que significa la posibilidad de variar el software que los integra para dotarlos de nuevas funcionalidades o perfeccionar las existentes. Este fenómeno ha hecho que el problema de poder disponer de Código Móvil Seguro sea cada vez de mayor interés científico e industrial. A su vez el extendido uso de código *multi-thread* en aplicaciones embebidas y en aplicaciones críticas hace necesario contar con un sistema de certificación y verificación que garantice seguridad de dicho código.

El análisis de programas *multi-thread* es difícil y costoso. Esto se debe: (1) al uso, por parte de los *threads*, de objetos compartidos (uso global y concurrente), (2) a la necesidad de utilizar sentencias de sincronización para el uso de objetos compartidos entre *threads*, y (3) al gran incremento del número de posibles trazas de ejecución (por *interleaving* de *threads*). Estas características pueden provocar errores en los programas producidos por no considerar posibles *interleavings* entre *threads* o por usar de forma inadecuada la sincronización.

Los programas *bytecode* Java *multi-thread* son usados por una gran variedad de aplicaciones y tecnologías tal como teléfonos móviles, tarjetas inteligentes y navegadores de internet. Por lo tanto, es necesario y de gran importancia contar con herramientas de verificación para estos programas. Al realizar la verificación sobre *bytecode* no es necesario contar con el código fuente. Además, si el compilador tiene algún bug que introduce algún posible error en el código, dicho error será detectado en la etapa de verificación. Notar que no ocurre lo mismo en el caso de realizar la verificación sobre el código fuente.

Cuando se analizan programas concurrentes es muy importante el modelo de memoria que se utiliza. El Modelo de Memoria de Java (JMM) es una abstracción que describe un conjunto de optimizaciones permitidas. Este modelo abstracto es suficiente para describir optimizaciones de compiladores tales como el reordenamiento de instrucciones. En general, los compiladores (procesadores y caches) se toman ciertas libertades en cuanto al tiempo y orden en la realización de operaciones de memoria. Las técnicas de sincronización son usadas para inducir un cierto orden en la realización de las operaciones de memoria, pero, sin una buena sincronización pueden suceder cosas sorprendentes. Muchos desarrolladores asumen que la sincronización es simplemente una forma de definir *secciones críticas*. Mientras que la exclusión mutua es un elemento de la semántica de sincronización. Existen dos elementos más: la *visibilidad* y el *orden* [4].

La especificación original del JMM, en el capítulo 17 del Java Virtual Machine Specification (JVMS) [8]), es ambigua y tiene serias inconsistencias en su semántica. Esto causa efectos no deseados y/o situaciones no intuitivas para los programadores.

El nuevo JMM, definido por W. Pugh et. al [12], formaliza el modelo de memoria de Java. De esta manera si los programadores conocen (y entienden) el modelo, sus aplicaciones *multi-thread* se comportarán de la manera esperada. Además, según W. Pugh [13], el modelo anterior prohibe la aplicación de muchas optimizaciones de código, mientras que el acceso no sincronizado a objetos inmutables (`final`) puede no ser seguros. El nuevo JMM es parte de Java 5.0 y soluciona las limitaciones que tenía el modelo anterior.

En este trabajo se presenta una técnica para verificar atomicidad de programas *bytecode* Java multi-thread. Los programas a verificar deben contar con una especificación del comportamiento de los bloqueos. Se propone un proceso de compilación de las especificaciones incluídas en el código fuente Java a código *bytecode*. El proceso de compilación genera un archivo `.class` que incluye la especificación. Se podría utilizar un compilador estándar Java (que no realice ninguna optimización) para generar el `.class`. Luego se inserta la especificación de modo tal que el *bytecode* generado pueda ser ejecutado en cualquier maquina virtual Java estándar. Por último, se analiza la relación entre las especificaciones y el JMM. . Este análisis es necesario dado que es muy importante la optimización del código generado. Al optimizar se deben tener en cuenta las restricciones impuestas y analizar las acciones que debe realizar el compilador para no violar ni la especificación ni el modelo de memoria al realizar optimizaciones.

Este trabajo esta estructurado de la siguiente manera, primero se presenta el lenguaje de es-

pecificación usado, sección 2. Dicho lenguaje se basa en una extensión del *Java Modeling Language* (JML) [7] , introducida por Rodrígez et. al [14], para especificar atomicidad. Se presentan, en la subsección 2.2 ,la propuesta del proceso de compilación de las especificaciones JML en el *bytecode* (inserción en el `.class`) y las estructuras definidas para realizar este proceso. En la sección 3, se propone informalmente un método para verificar las especificaciones y realizar la verificación de atomicidad. Luego, en la sección 4, se describe el nuevo modelo de memoria de Java definido por W. Pugh et. al [12]. En la sección 5 se analizan el efecto de las especificaciones sobre las optimizaciones de *bytecode* permitidas por el el nuevo JMM. Por último se presentan las conclusiones y trabajos futuros en la sección 6.

2 El Lenguaje de Especificación

Java Modeling Language (JML) [7] es un lenguaje de especificación formal que permite especificar contratos en programas Java por medio de invariantes de clases, pre y post-condiciones de métodos. Rodrígez et. al [14] presentan una extensión de JML en la cual los constructores provistos permiten especificar atomicidad de métodos y el comportamiento de bloqueos en programas Java *multi-thread*. El concepto de atomicidad es utilizado para evitar los problemas de interferencia interna y externa lo que permite realizar verificaciones de forma modular.

Las especificaciones JML en el código permiten obtener un sistema de certificación/verificación para programas *multi-thread* más efectivo, potente y eficiente. Además, contar con la posibilidad de realizar verificaciones modulares se podría obtener un sistema de certificación/verificación para los contratos de la especificación JML estandar. Para esto se deben compilar las especificaciones JML en el *bytecode* y extender las técnicas de certificación/verificación de contratos a bytecode para obtener un ambiente PCC sobre este *bytecode* anotado. Un componente muy importante de esta línea son los métodos para poder razonar usando las especificaciones JML sobre *bytecode*. En este trabajo solo focalizamos en la verificación de atomicidad.

Las extensiones del JML realizadas por Rodrígez et. al [14] soportan el concepto de atomicidad y permiten la especificación de comportamiento de bloqueos. El concepto de atomicidad (que proviene de la teoría de reducción de Lipton para programas paralelos [9]) es usado para evitar los problemas de interferencia. Este enfoque que introduce concurrencia en JML separa la especificación de los métodos en dos partes: (1) especificación de atomicidad e independencia y (2) especificación de comportamiento secuencial (o funcional).

Interferencia causa problemas que afectan el razonamiento modular del estado de los programas *multi-thread*. Podemos clasificar interferencia en: Interna y Externa. **Interferencia Interna** ocurre cuando un *thread* afecta la ejecución del método corriente de otro *thread* (porque cambió datos que el método puede observar). **Interferencia Externa** ocurre cuando algún otro *thread* realiza un cambio en el estado observable entre la invocación a un método y su ejecución (o entre la salida de un método y el retorno al método que realizó la llamada).

Los programadores pueden usar una gran variedad de mecanismos para asegurar *thread safety* pero la noción central es: no interferencia. Esta es la observación fundamental en la que se basa el trabajo de Rodrígez et al. [14]. Interferencia puede ser evitada usando sincronización. La sincronización evita *interleavings* no deseados y controla el acceso a los datos. Cuando se menciona *thread safety* se esta refiriendo a sincronización, a control de acceso a los datos o una combinación de ambos.

Evitar los problemas de interferencia permite razonar modularmente. El enfoque de Rodrígez et. al [14] se basa en el concepto de **atomicidad** e **independencia**. Una región de código (por ejemplo, el cuerpo de un método) es atómica si las sentencias de esta región son *serializables*. Es decir, para cualquier traza de ejecución de las sentencias de la región (posiblemente intercaladas con sentencias de otros *threads*) hay una traza de ejecución equivalente donde las sentencias de la región son ejecutadas secuencialmente (sin *interleavings* con otros *threads*).

Si una región de código es atómica, entonces es correcto razonar acerca de sus acciones como si estas ocurrieran en un único paso atómico. En otras palabras, se pueden usar técnicas tradicionales de razonamiento secuencial en la región de código atómica. Desde otro punto de vista, para una región de código atómica es correcto considerar solo dos estados: el estado previo (precondición) y

el estado posterior (postcondición). Es decir, cualquier interferencia (*interleavings*) de otros *threads* no produce cambios en su estado, pero este paso atómico puede interferir en la ejecución de otros *threads*.

Una región de código es independiente si todas las sentencias que contiene son independientes. Una sentencia es independiente si puede moverse (reordenarse a derecha e izquierda con respecto a *interleavings* con otros *threads*) sin que se modifique el efecto que produce en el estado del programa cuando se ejecuta. En la teoría de Reducción de Lipton estas sentencias se denominan *both mover* [9].

2.1 Los Constructores de JML para especificar atomicidad

A continuación se presentan los nuevos constructores de JML introducidos por Rodrígez et. al [14] relativos a atomicidad, independencia, bloqueos y otras propiedades específicas de programas *multi-thread*.

2.1.1 Notaciones para Bloqueos

monitors_for <identificador>←<lista de referencias> Esta cláusula permite especificar los bloqueos que protegen el acceso al atributo dado por el identificador. Es decir, significa que todos los bloqueos nombrados en la <lista de referencias> deben ser obtenidos en orden por el *thread* para acceder al atributo <identificador>. Esta <lista de referencias> es una lista de identificadores y atributos separados por comas.

\lockset() Esta expresión retorna el conjunto de objetos que el *thread* corriente mantiene bloqueados en el estado corriente. Por ejemplo, para especificar que el *thread* corriente mantiene bloqueado a o_1 se utiliza la expresión `\lockset().has(`o_1`)`.

locks <lista de referencias> Esta cláusula puede aparecer en la especificación de una precondición y tiene dos propósitos: (1) es una sentencia explícita de los bloqueos que adquiere y libera el método corriente durante su ejecución; (2) establece implícitamente una condición de independencia. Si el método bloquea los objetos especificados entonces la llamada debe garantizar ser independiente. Esto ayuda a realizar verificación modular de atomicidad.

\lock_protected (referencia) Este predicado establece que el objeto referenciado tiene el acceso protegido por algún conjunto de bloqueos (no vacío) y todos estos bloqueos son mantenidos por el *thread* corriente. Hay que notar que es una propiedad muy fuerte ya que que el acceso restringido es sobre el objeto referenciado (no sobre la referencia) y que otros *threads* pueden tener *alias* al objeto pero su acceso está restringido.

2.1.2 Notaciones de Restricción de Heap

rep Este modificador de tipo puede ser usado en la declaración de atributos. Establece que el objeto referenciado por el atributo especificado es parte de la representación de la clase. Es decir, no puede haber ninguna referencia del objeto referenciado por el atributo fuera del objeto de la clase que lo contiene. Esta posibilidad de especificar ausencia de *alias* externos facilita la verificación modular. Este modificador de tipos proviene del Sistema de Tipos *Universe* [11].

readonly Este modificador de tipo puede ser usado en la declaración de atributos. Este modificador establece que el atributo contiene una referencia de solo de lectura (el objeto referenciado no se puede modificar con esa referencia). Las referencias *read-only* no son necesariamente apropiadas por el objeto que contiene al atributo *read-only* (como en el caso de existir *aliasing*). La idea es que solamente la identidad de un objeto *read-only* importa al estado abstracto del objeto. Este modificador de tipos también proviene del Sistema de Tipos *Universe* [11].

\thread_local (referencia) Cláusula que especifica que el objeto referenciado es adueñado por el *thread*. Un objeto `o` es adueñado por un *thread* `t` si `t` es el único que puede acceder a `o`

por alguna cadena de referencias. Esta cláusula es muy usada para verificación modular de atomicidad porque accesos a objetos locales de un *thread* son independientes (no interferentes).

2.1.3 Modificadores de Atomicidad

atomic Este modificador de métodos especifica que cuando un método es invocado en un estado que cumple con su precondición su implementación debe asegurar que la ejecución resultante es serializable. Este modificador resuelve el problema de la interferencia interna ya que estable que todo *interleaving* de otros *threads* durante la ejecución del método no afectan su estado.

2.1.4 Interferencia Externa

\independent Este predicado solo puede ser usado en postcondiciones y especifica que el método cumple la propiedad de ser independiente (todas sus sentencias son independientes). Por ejemplo, accesos a objetos locales al *thread* y acceso a objetos que son protegidos por bloqueos son sentencias independientes, dado que otros *threads* no pueden observar dichos accesos. Es decir, cuando se cumple la precondición la ejecución del método es independiente.

\thread_safe(SR) esta cláusula especifica en la pre o en la postcondición que el objeto `SR` es `\thread_local(SR)` o que es `\locK_protected(SR)`. Esta cláusula se utiliza para evitar interferencia externa deshabilitando el acceso a aquellos objetos que se encuentran en la pre y en la postcondición.

2.2 Compilación de las Especificaciones

En esta sección se describe el formato que se propone para introducir las anotaciones en el archivo `.class`. El formato sigue los lineamientos del formato estándar definido por la JVMS. El formato del archivo `.class` se presenta usando pseudo-estructuras en notación C. Para evitar confusiones (por ejemplo, entre atributos de clases e instancias de clases, etc.) los contenidos de las estructuras que describen el formato del archivo `.class` se denominan *ítems*.

2.3 Información Necesaria para Compilar las Especificaciones

Un archivo `.class` consiste de una secuencia de bytes. 16-bit, 32-bit y 64-bit son construídos por leer dos, cuatro u ocho bytes consecutivos, respectivamente. Los tipos `u1`, `u2` y `u4` representan respectivamente uno, dos o cuatro bytes sin signo.

Las tablas consisten de cero o más *ítems* de tamaño variable; estas son usadas en varias estructuras del archivo `.class`. Si bien se utiliza la sintaxis de arreglos de C para referirse a tablas de *ítems* no es posible traducir un índice de una tabla directamente a un desplazamiento en bytes en la tabla. Esto es consecuencia de que las tablas son secuencias de bytes de tamaño variable Las referencias a estructuras de datos como arreglos pueden ser indexadas como un arreglo, ya que, consisten de cero o más *ítems* continuos de tamaño fijo [8].

Cuando se compila la especificación del método m el compilador usa información de las variables locales (`Local_Variable_table`) y de la tabla que relaciona el número de línea del código fuente con las instrucciones del *bytecode* (`Line_Number_table`). Estos son atributos opcionales de la tabla `attributes` de `Code_attributes` perteneciente a `Method_Info`. Este archivo `.class` puede ser generado por cualquier compilador estándar Java que genere estas dos tablas para todo método que se encuentre en el código. Se considera un compilador estándar a aquel que no realiza ninguna optimización especial en el código generado [1].

Los compiladores pueden definir y generar archivos `.class` con nuevos atributos en las tablas de atributos de estructuras del archivo `class`. Implementaciones dela maquina virtual de Java (JVM) pueden reconocer y utilizar nuevos atributos encontrados en las distintas tablas de atributos. Sin embargo, cualquier atributo no definido como parte de la JVMS no debería afectar la semántica de las clases o interfaces. Por ejemplo, se permite definir nuevos atributos para soportar algún

uso específico [8]. No hay que dejar de mencionar que las implementaciones de la JVM requieren ignorar aquellos atributos que no reconocen. Es decir, los archivos `.class` previstos para cada implementación particular de JVM podrán ser usados por otras implementaciones. Incluso si esas implementaciones no pueden utilizar la información que los archivos `.class` contienen [8].

2.4 Inclusión de las Anotaciones en el Archivo `.Class` Java

2.4.1 Anotaciones en Clases

Las siguientes anotaciones son incluídas como *ítems* en el arreglo de atributos (`attributes[]`) de la estructura `ClassFile` que contiene toda la información de una clase (el `bytecode de una clase`) [8, §4.1]. Cada valor de esta tabla de *ítems* debe ser del tipo `attribute_info` [8, §4.7]. Como ya se mencionó la implementación de una JVM debe ignorar todos los *ítems* en la tabla (de atributos) de la estructura del archivo `Class` que no reconoce. Aquellos *ítems* no definidos en la especificación no afectan a la semántica del archivo `Class` pero proveen información adicional [8, §4.7.1]. En este último caso se encuadran los nuevos *ítems* introducidos. Un tratamiento similar reciben los nuevos *ítems* para métodos, atributos y expresiones.

Ítem `locked_if <predicado>` Esta cláusula permite especificar cuando una instancia de un bloqueo (*lock type*) está en estado de bloqueo. En la figura 1 se puede apreciar la estructura que se define para este *ítem*. Donde,

attribute_name_index es un índice válido en la tabla `constant_pool`. El valor que debe contener la tabla en esa entrada debe ser una estructura `CONSTANT_Utf8_info` que represente el string "Lockedif".

attribute_length contiene la longitud del *ítem* en bytes menos 6 (bytes).

attribute_formula contiene el código de la fórmula que representa el `<predicado>` de la cláusula.

```
Locked_If_attribute {
u2 attribute_name_index;
         u4 attributte_length;
         formula attribute_formula;
}
```

Figura 1: Estructura para el *Ítem* `locked_if`.

2.4.2 Anotaciones en Métodos

Para la cláusula `\atomic` y la cláusula la `\locks` se definió un nuevo *ítem* en el JVM `bytecode` para cada una de las cláusulas. Estos se incluyen en la estructura `method_info` como elementos del arreglo `attributes`. También podrían ser incluídos como un nuevo *ítem* del `JMLMethod_attribute` definido por L. Burdy y M. Pavlova [1, §4.1]. Para ejemplificar se muestra la estructura definida para la cláusula `\locks`:

Ítem `\locks <store_ref_list>` Esta cláusula especifica los bloqueos y desbloqueos que realiza un método. En la figura 2 se puede apreciar la estructura que se define para este *ítem*. Donde,

attribute_name_index es un índice válido en la tabla `constant_pool`. El valor que debe contener la tabla en esa entrada debe ser una estructura `CONSTANT_Utf8_info` que represente el string "Locks".

attribute_length contiene la longitud del *ítem* en bytes menos 6 (bytes).

locks_count cantidad de bloqueos que se encuentran en `<store_ref_list>`.

locks cada entrada en este arreglo representa un bloqueo.

```
Locks_attribute {
u2 attribute_name_index;
u4 attributte_length;
u2 locks_count;
field_info locks[locks_count];
}
```

Figura 2: Estructura para el *Ítem* `\locks`.

2.4.3 Anotaciones en Fields (ítems de clases)

Para las cláusulas `\rep`, `\readonly` y `\monitors_for` se definieron nuevos *ítem*s en el JVM `bytecode` que se incluyen en la estructura `field_info` como elementos del arreglo `attributes`. Como ejemplo se presenta la información generada para la cláusula `\monitors_for`.

Ítem `monitors_for <ident> <-- <store_ref_list>` Esta cláusula especifica que el thread debe mantener (en ese orden) los bloqueos especificados en `<store_ref_list>` para acceder al field `<ident>`. `<ident>` es el field donde se almacena el *ítem* y `<store_ref_list>` es una lista de identificadores, fields, accesos a arreglos, etc. En la figura 3 se puede apreciar la estructura que se define para este *ítem*. Donde,

attribute_name_index es un índice válido en la tabla `constant_pool`. El valor que debe contener la tabla en esa entrada debe ser una estructura `CONSTANT_Utf8_info` que represente el string "Monitorsfor".

attribute_length contiene la longitud del *ítem* en bytes menos 6 (bytes).

locks_count cantidad de bloqueos que se encuentran en `<store_ref_list>`.

locks cada entrada en este arreglo representa un bloqueo.

```
Monitors_For_attribute {
u2 attribute_name_index;
u4 attributte_length;
u2 locks_count;
field_info locks[locks_count];
}
```

Figura 3: Estructura para el *Ítem* `monitors_for`.

2.4.4 Fórmulas y Expresiones

Para compilar las expresiones se extendió la gramática de las expresiones JML presentada por L. Burdy y M. Pavlova [1, §7] para que contemple las cláusulas: `\independent`, `\lockset`, `\thread_local`, `\lock_protected` y `\thread_safe`. El proceso de compilación es similar al presentado por L. Burdy y M. Pavlova [1, §7], solo se debe extender la función de compilación para que contemple estos constructores.

```
JML_Expression ::=  \independent
| \lockset()
| \thread_local( Identifier )
| \lock_protected( Identifier )
| \thread_safe( Identifier )
...
```

Figura 4: Gramática de Expresiones.

3 Verificación Modular de las Especificaciones JML

Para verificar las especificaciones JML de atomicidad sobre *bytecode* se realiza una análisis de flujo de control que explora todos los posibles caminos de ejecución. Es decir, se realiza una ejecución abstracta considerando unicamente las instrucciones que afectan o son afectadas por las especificaciones. Este análisis explota la noción de *independencia*.

A continuación se analizan informalmente las reglas a aplicar para verificar modularmente si un método especificado como **atomic** realmente lo es. Verificar que un método es **atomic** usando la noción de independencia consiste en corroborar que cuando se accede a un objeto este es local al *thread* o que el acceso al objeto está protegido por un conjunto de bloqueos (uno o más bloqueos). En el primer caso, el objeto es local si esta especificado como **rep**, **read_only** o **thread_local**. En el segundo caso, el objeto no solo debe formar parte de una cláusula **monitors_for** y/o **locks** sino que también debe encontrarse (el acceso al objeto) dentro de la cadena de bloqueos establecida. También, el objeto, puede estar especificado como **lock_protected** en la precondición del método.

Para verificar que un atributo declarado **rep** realmente lo es se debe tener la certeza que es imposible que exista una referencia al atributo fuera del *thread* corriente. Esto se puede realizar con *escape analysis*. Verificar que un atributo u objeto es **read_only** o **thread_local** de manera modular (sin conocer el sistema final) es imposible ya que pueden existir *alias* del objeto en otros *threads*. Notar que la existencia de *aliasing* no invalida especificaciones **read_only** y **thread_local**. Con estas especificaciones se podría usar verificaciones dinámicas o verificarlas cuando se integra el *thread* al sistema final. Hay que notar que en algunos casos se puede verificar la clausula **thread_local**, por ejemplo, en aquellos casos donde el objeto especificado es **rep** o esta protegido por algún conjunto de bloqueos.

La especificación **locks** solo implica corroborar que durante la ejecución del método los bloqueos especificados son obtenidos y posteriormente liberados. Verificar la cláusula **monitors_for** no presenta mayores inconvenientes en el análisis del *thread* corriente. Pero, si otros *threads* acceden al objeto deberían hacerlo respetando la misma cadenas de bloqueos. Esto no se puede realizar modularmente. Con lo cual, se puede plantear hacer la verificación antes de la integración del *thread*. Un caso similar sucede con la cláusula **lock_protected** que no especifica cuales son los bloqueos involucrados (solo que existe un conjunto no vacío de bloqueos para un objeto determinado). El problema con esta especificación se encuentra en el punto de verificar cuales son los bloqueos especificados por **lock_protected**. Por ejemplo, para verificar `lock_protected(x.f)` se debería usar la cláusula `monitors_for` para `f` en la clase `x`. Pero con esto se pierde modularidad. Otra alternativa es considerar solo los bloqueos que adquiere el *thread* corriente hasta el punto donde se referencia el objeto especificado por `lock_protected`, los cuales deberían estar especificados en la precondición del método que se está analizando. Pero, esto no garantiza que los bloqueos sean los mismos que los especificados para `f` en la clase `x`. Lo que se verifica en este punto es que todos los métodos que accedan a $x.f$ adquieran siempre la misma cadena de bloqueos (en la misma secuencia).

Para verificar que se cumple la cláusula **thread_safe(O)** hay que chequear que el objeto especificado, **O**, es local al *thread* corriente (**thread_local(O)**) o está protegido por algún conjunto de bloqueos (**lock_protected(O)**). Notar que para verificar la ausencia de interferencia externa también debe verificarse que los objetos que intervienen en la precondición o en la postcondición son **thread_safe**.

Para verificar que se cumple el predicado **independent** se debe chequear que todas las sen-

tencias del método corriente son independientes.

4 Breve Descripción del Nuevo JMM

Un modelo de memoria describe la relación entre las variables de un programa (instancias de atributos, atributos estáticos y elementos de un arreglo) y los detalles de bajo nivel para representar su *binding* con valores, el acceso a ellos y su tiempo de vida sobre la memoria en el sistema de computación real. El modelo de memoria describe como las acciones realizadas por un thread sobre memoria (lecturas y escrituras) afectan las acciones realizadas en memoria por otro thread. Si bien los objetos finalmente son almacenados en memoria, el compilador, el procesador o la cache pueden decidir en que momento mover el valores a la memoria principal.

El modelo de memoria de `Java` permite (al compilador y a la cache) cierta libertad en el orden en el cual se deben mover datos entre registros (o memoria cache) y memoria principal. Pero, el programador puede restringir estas libertades definiendo ciertas reglas de visibilidad (utilizando `sinchronized` o `volatile`). Es decir, en ausencia de sincronización las operaciones en memoria pueden suceder en diferentes ordenes (hasta en los más sorprendentes e inesperados).

El modelo de memoria de Java es una abstracción que describe un conjunto de optimizaciones permitidas. Este modelo abstracto es suficiente para describir optimizaciones de compiladores tal como, por ejemplo, reordenamiento de instrucciones. En general, los compiladores (y también procesadores y memorias caches) se toman ciertas libertades en cuanto al tiempo y orden en la realización de operaciones de memoria. La sincronización es usada para inducir un cierto orden en la realización de las operaciones de memoria. Pero, como es bien conocido en procesos concurrentes, sin una buena sincronización pueden suceder cosas sorprendentes. Muchos desarrolladores asumen que sincronizar es simplemente una forma de definir *secciones críticas*. Mientras que exclusión mutua es un elemento de la semántica de sincronización. Además, existen dos elementos más: la *visibilidad* y el *orden* [4].

El JMM (como estaba especificado originalmente en el capítulo 17 del JVMS [8]) tenía serias inconsistencias en su semántica que permitían ciertas situaciones no deseadas y que no eran intuitivas para los programadores. Además, el JMM no permitía la aplicación de muchas optimizaciones de código, como lo señala W. Pugh [13]. Afortunadamente, fue posible definir un nuevo modelo de memoria para Java. Dicho modelo es consistente con la intuición de los desarrolladores y permite ser utilizado con programas bien sincronizados bajo el modelo anterior [5]. W. Pugh et al. [12] definen y formalizan el modelo de memoria de Java. De esta manera si los programadores conocen (y entienden) el modelo, sus aplicaciones multi-thread se comportarán de la manera esperada.

J. Manson y W. Pugh demuestran que el nuevo JMM permite las siguientes optimizaciones [10]:

- Reordenamiento de Instrucciones. Si se considera un programa P y un programa P' que se obtiene de intercambiar dos sentencias adyacentes x e y de P. Las sentencias x e y pueden ser dos sentencias tal que:
  - reordenar x e y no elimina ninguna relación *happens-before* en ninguna ejecución válida.
  - x e y no son accesos críticos a la misma variable.
  - x e y no son ambos acciones de sincronización.
  - la semántica intra-thread de x e y permite reordenarlas (por ejemplo, x no almacena en un registro leído por y). Si se respeta la semántica intra-thread entonces son legales las optimizaciones de compilación comunes para aplicaciones single-thread.
- *Unrolling/Merging* de Sentencias de Control de Flujo. El compilador puede hacer transformaciones del código que dividan (o unan) el flujo del programa en ejecuciones que sean equivalentes a la ejecución original. Toda forma de *splitting* y *merging* debe preservar la semántica intra-thread.
- Lecturas Especulativas. Una lectura especulativa no puede ser realizada antes que:
  - la última acción de sincronización que obtiene un bloqueo (*acquire*).

Puede Reordenar	Segunda Operación		
Primer Operación	Normal Load Normal Store	Volatile Load MonitorEnter	Volatile Store MonitorExit
Normal Load Normal Store			No
Volatile Load MonitorEnter	No	No	No
Volatile Store MonitorExit		No	No

Donde:

- Normal Loads es: `getfield, getstatic, array load` de fields no volatile.
- Normal Store es: `putfield, putstatic, array store` de fields no volatile.
- Volatile Loads es: `getfield, getstatic` de fields volatile que son accedidos por múltiples *threads*.
- Volatile Stores es: `putfield, putstatic` de fields volatile que son accedidos por múltiples *threads*.
- MonitorEnters bloqueos de objetos accedidos por múltiples *threads* (incluye entradas a métodos sincronizados).
- MonitorExits desbloqueos de objetos accedidos por múltiples *threads* (incluye salidas de métodos sincronizados).

Figura 5: Reordenamientos que se permiten en el nuevo JMM [6].

– la escritura de una variable de la cual se lee.

Las lecturas especulativas anteriores no excluyen a otras posibles optimizaciones que puedan ser permitidas por el modelo. Pero esta demostrado que estas no violan el JMM [10]. Dicha demostración no asevera nada de otras posibles lecturas especulativas.

Para un compilador el JMM consiste principalmente de reglas que deshabilitan el re-ordenamiento de ciertas instrucciones (bloques y accesos a objetos). En la figura 5 se muestran las instrucciones (asociadas a secuencias de *bytecode*) que no pueden ser reordenadas [6].

Las celdas en blanco de la tabla de la figura 5 significan que el reordenamiento es permitido si las instrucciones no son dependientes (según la semántica básica de Java). Por ejemplo, no se puede reordenar un `Normal Load` con un `Normal Store` de la misma posición de memoria. Pero, si se permite el reordenamiento de estas operaciones sobre posiciones distintas de memoria. Las celdas que contienen `No` significan que la primer instrucción no puede ser reordenada con ninguna ocurrencia posterior de la segunda operación.

El nuevo modelo de memoria también da una nueva semántica para `volatile` y `final`. La semántica original de `volatile` garantiza que las lecturas y escrituras de atributos `volatile` debe ser realizadas directamente de memoria principal y que se deben ejecutar en el orden en que se encuentran en el *thread*. Pero, en ella no se especificaba nada en cuanto a la relación entre atributos `volatile` y `no volatile`. Es decir, se permitía realizar cualquier reordenamiento que involucrara una operación sobre un objeto `volatile` y uno `no volatile`. Por lo cual, los atributos `volatile` no podían ser usados como *flags*. El nuevo JMM no permite cualquier reordenamiento entre lecturas y escrituras `volatile` y otras operaciones de memoria (como se puede apreciar en la figura 5).

El nuevo JMM garantiza que los atributos `final` sean visibles a otros *threads* luego de su inicialización. Para lograr esto, no permite que la referencia al objeto este disponible hasta que su constructor no se termine de ejecutar. El modelo de memoria anterior permitía (en ausencia de sincronización) la posibilidad que algún *threads* obtenga el valor por defecto de un atributo `final` y luego en otro instante de tiempo obtenga el valor correcto.

5 Las Especificaciones y el Nuevo JMM

En esta sección, se analiza el efecto de las nuevas cláusulas de JML en el modelo de memoria de Java (JMM). Es interesante analizar el impacto de las especificaciones en las optimizaciones permitidas

por el nuevo JMM.

Las especificaciones `monitor_for`, `lockset`, `lock_protected`, `locks`, `atomic` e `independent` no introducen ninguna información que deba ser tenida en cuenta para realizar optimizaciones permitidas por el JMM. Es decir, estas especificaciones no restringen las optimizaciones permitidas ni habilitan nuevas optimizaciones.

De manera similar ocurre con las cláusulas `rep`, `readonly` y `thread_local`. Es decir, permiten reordenar los objetos asociados a estas especificaciones. Pero, estas últimas cláusulas, pueden ser violadas por otros objetos que referencien a los objetos a que hacen alusión las cláusulas en cuestión. Esto no puede ser verificado modularmente. Entonces, que que sucede en aquellos casos en donde no se cumple la especificación. Por ejemplo, si un objeto especificado como `thread_local` no lo es realmente. Este tipo de casos solo se detectará cuando el *thread* sea integrado a un sistema (el sistema a final donde será ejecutado). Por lo cual se pueden plantear dos alternativas: (1) Realizar una verificación de la integración. Verificar que en el sistema multi-thread cumplen lo especificado, por ejemplo, analizando posibles condiciones de *aliasing* de determinados objetos. (2) Introducir verificaciones dinámicas que den una excepción si no se cumple lo especificado. Por ejemplo, introduciendo código bytecode que realice esta verificación. O modificar la JVM para que dispare una excepción cuando se viole una de las condiciones especificadas.

Como se puede apreciar utilizar estas especificaciones en programas *bytecode* no limitan las optimizaciones permitidas por el JMM. También, se debe notar que estas especificaciones no introducen ninguna inconsistencia con respecto a la especificación del JMM.

6 Conclusiones y Trabajos Futuros

Se presentó una técnica que permite verificar atomicidad de aplicaciones *bytecode* multi-thread. Verificar atomicidad permite posteriormente verificar propiedades funcionales más complejas como si fueran programas secuenciales. Se presentó el proceso de compilación de la especificación a Java *bytecode*. Se analizó la utilización del entorno con el nuevo JMM, concluyendo que la técnica presentada no introduce inconsistencias con respecto al modelo de memoria de Java.

El proceso de compilación presentado permite utilizar cualquier compilador Java (que no realice ninguna optimización), luego se pueden insertar la especificación JML y realizar las optimizaciones que se deseen (mientras respeten el JMM). Las especificaciones no condicionan las optimizaciones permitidas por el modelo de memoria ni las optimizaciones vuelven inconsistentes las especificaciones.

Cualquier JVM (que cumpla con la JVMS) puede utilizar el *bytecode* con la especificación JML. Esto se debe a que si la JVM no reconoce la especificación la descarta. Además las especificaciones no afectan a la semántica del *bytecode*. Si se desea verificar el *bytecode* se puede insertar un módulo (el verificador) antes de la JVM.

El JML extendido utilizado en este trabajo no permite verificar ausencia de *deadlock* y *starvation*. Tampoco permite verificar las consecuencias de integrar un *thread* a un sistema. Los consumidores de código deberían publicar los bloqueos (y el orden en que se deben realizar) que se deben mantener para acceder a cada objeto. También deberían publicar que objetos deben ser `thread_local`. Estos son aspectos que se desean profundizar e integrar al entorno presentado.

También es importante y de gran relevancia realizar casos de estudio complejos.

Se desea estudiar la posibilidad de generar automáticamente las especificaciones con este JML extendido. Pero generar modularmente la información de las cláusulas `thread_local` y `readonly` no es trivial.

También se desea generar un método que permita realizar la verificación y que genere el esquema de prueba para que pueda ser utilizada en un ambiente de código móvil siguiendo las ideas de de la técnica *Proof-Carrying Code* (PCC). Esta técnica exige a un productor de software que entregue su programa ejecutable conjuntamente con una prueba formal de que dicho programa respeta la política de seguridad del receptor. Dicha política de seguridad se formaliza mediante un sistema de axiomas y reglas de inferencia, sobre el cual debe basarse la demostración construida por el productor. El consumidor, por su parte, verifica que la prueba sea válida y que el código recibido

corresponda a la demostración, y sólo ejecuta el código en caso de que ambas respuestas sean positivas.

6.1 Trabajos Relacionados

Verificar, basándose en la Teoría de Reducción de Lipton [9], que un método declarado atómico lo es realmente puede realizarse mediante una gran variedad de caminos. Flanagan y Qadeer [2] utilizan reducción para verificar atomicidad en un sistema de tipos para `Java`. Otro técnica usada para detectar violaciones de atomicidad (usando reducción) consiste en determinar dinámicamente el conjunto de bloqueos que protegen el acceso a los datos compartidos y los objetos locales de los *threads*, técnica utilizada por la herramienta `Atomizer` [3]. Hatcliff et al [14] utilizan *Model Checking*, los resultados experimentales que obtuvieron permiten pensar que es factible realizar modularmente la verificación de atomicidad. También, se puede utilizar analisis de flujo de control y verificar independencia como se desarrolló en este trabajo.

Referencias

[1] Lilian Burdy, Mariela Pavlova. "Specification of java Modeling Language coding into the Java Bytecode" (Draft Version). Reporte Técnico del proyecto Everest, INRIA, Sophia-Antipolis. Febrero, 2005.

[2] C. Flanagan, S. Qadeer. ´´´A Type and Effect System for Atomicity". Proceedings of the ACM Conference Programming Language Design and Implementation. pp. 338-349. 2003.

[3] C. Flanagan, S. Freund. ´´Atomizer: A Dinamic Atomicity Checker for Multithreaded Programs". Proceedings of the ACM Symposium on the Principles of Programming Languages. 2004.

[4] Brian Goetz. "JSR 133 in Public Review". java.net articles. Abril de 2004. http://today.java.net/pub/a/today/2004/04/13/JSR133.html

[5] Brian Goetz. "Fixing the Java Memory Model, Part 1 y Part 2". Java Theory and Practice, IBM developerWorks on Java programming concepts, techniques, and best practices (columna mensual). Febrero-Marzo de 2004. http://www-106.ibm.com/developerworks/java/library/j-jtp03304/

[6] Doug Lea. "The JSR-133 Cookbook for Compiler Writers". http://gee.cs.oswego.edu/dl/jmm/cookbook.html

[7] G. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, J. Kiniry. "JML: Reference Manual". Departament of Computer Science, Iowa State University. Disponible: www.jmlspecs.org. 2004.

[8] Tim Lindholm, Frank Yellim. "Java Virtual Machine Specification". Segunda Edición, Java Software, Sun Microsystems, Inc., 2004.

[9] R. Lipton. "Reduction: a method of proving propierties of parallel programs". Communications of the ACM 18 717-721. 1975.

[10] Jeremy Manson, William Pugh. "Proof sketch the Manson/Pugh JMM allows reordering, Unrolling/Merging and Speculative Reads". JMM web page (http://www.cs.umd.edu/users/pugh/java/memoryModel). Febrero de 2004.

[11] Peter Müller, A. Poetzsch-Heffter. "A Type System for Alias and Dependency Control". Reporte Técnico 279, Universidd de Hagen. 2001.

[12] Jeremy Manson, William Pugh, Sarita V. Adve. "The Java memory model". POPL 2005. pp 378-391.

[13] William Pugh. "The Java memory model is fatally flawed". Concurrency - Practice and Experience 12(6). pp 445-455. 2000.

[14] Edwin Rodríguez, Matthew B. Dwyer, Cormac Flanagan, John Hatcliff, Gary T. Leavens, Robby. "Extending Sequential Specification Techniques for Modular Specification and Verification of Multi-Threaded Programs". ECOOP 2005.