Especificación formal y verificación de invariantes para un protocolo de coherencia del caché

Sergio Ramírez Rico

Director
Dr. Camilo Rocha

Programa de Matemáticas
Escuela Colombiana de Ingeniería Julio Garavito
Bogotá, Colombia
Mayo, 2015

Resumen

Este documento presenta un caso de estudio en la especificación y verificación de invariantes de un protocolo de coherencia del caché. Este protocolo está fundamentado en la técnica ESI de coordinación basada en acceso exclusivo o compartido a un recurso. La verificación de los invariantes utiliza análisis algorítmico y deductivo, y se emplea Maude como lenguaje de especificación y sistema de verificación.

Agradecimientos

Quiero agradecer al Profesor Camilo Rocha por ofrecerme trabajar con él y mostrarme el mundo de la investigación. Le agradezco por sus consejos, enseñanzas, tiempo y apoyo. Su experiencia, conocimiento y dedicación son fuente de inspiración para mejorar y seguir adelante.

A mis padres Gloria y Jorge, por su apoyo incondicional y consejos, y a Laura por su acompañamiento y apoyo a lo largo de mi carrera.

Índice general

Capítulo 1

Introducción

Esta tesis se desarrolla a partir del caso de estudio del protocolo ESI para la coherencia del caché [**10**]. En este protocolo el sistema tiene un controlador, el cual coordina el acceso de procesos a un recurso compartido, manteniendo información sobre los procesos que hacen parte del sistema.

En el tipo de protocolo estudiado en esta tesis es importante la verificación de *invariantes* que garanticen su correcto funcionamiento. El protocolo ESI es utilizado en el manejo coherente de bloques de memoria y por lo tanto es primordial asegurar que el sistema no va a hacer uso incorrecto de la memoria como recurso compartido. La principal propiedad del sistema estudiado en esta tesis es un invariante de exclusión mutua entre procesos para el acceso al recurso compartido.

Los invariantes son fórmulas temporales utilizadas en la especificación y verificación de sistemas de software. Estas propiedades son de gran importancia ya que si son ciertas garantizan que nada malo va a ocurrir en el sistema. En resumen, un invariante indica que no es posible alcanzar un estado erroneo o inseguro.

La especificación formal del sistema se hace en el marco de la lógica de reescritura. Con esta estructura los sistemas se pueden axiomatizar por medio de ecuaciones y reglas de reescritura. La manera como trabaja el sistema es haciendo transiciones entre estados utilizando las reglas de reescritura correspondientes. Como lenguaje de especificación y sistema de verificación se utiliza Maude [**3**], y en particular se utilizan sus comandos `search`, `reduce`, `modelCheck` y la herramienta InvA [**11**], para análisis deductivo.

Específicamente este documento presenta el desarrollo de:

- Modelamiento formal de estados y transiciones del sistema.

- Verificación algorítmica y deductiva de invariantes de ESI.
- Uso de Maude como sistema de verificación.

Las verificaciones algorítmicas y deductivas incluidas en este documento se basan en:

- La exploración de grafos de estados alcanzables desde un conjunto finito de estados iniciales y el uso de *LTL-model checking*.
- Reducción de demostraciones de invariantes sobre grafos de transiciones potencialmente infinitos a verificaciones de obligaciones netamente ecuacionales con el uso de técnicas de análisis simbólico y deductivo.

La diferencía más relevante entre los dos tipos de análisis incluidos en este trabajo es que la verificación algorítmica se puede usar en sistemas que tienen una cantidad finita de estados iniciales y alcanzables desde cada estado inicial, mientras la verificación deductiva puede utilizarse en sistemas con una cantidad infinita de estados. Algunas de estas limitaciones se exploran en esta tesis.

Las verificaciones incluidas aquí permiten hacer explícitas la complejidad del protocolo y las consecuencias de su alta concurrencia mediante el cálculo de estados alcanzables, transiciones entre ellos y tiempo empleado en las exploraciones.

El documento está organizado de la siguiente manera:

1. El Capítulo 2 incluye algunos conceptos requeridos para la lectura del documento.
2. El Capítulo 3 presenta la descripción del protocolo ESI, y el modelamiento de sus estados y transiciones.
3. Los capítulos 4 y 5 presentan, respectivamente, los análisis algorítmico y deductivo de ESI.
4. El Anexo A incluye los módulos de sistema y funcionales que se utilizan en el caso de estudio.

Capítulo 2

Preliminares

2.1. Lógica de reescritura

Lógica de reescritura [**7**] es una lógica de cambios concurrentes. Las reglas de la lógica de reescritura son patrones generales para acciones básicas que pueden ocurrir concurrentemente con otras acciones en un sistema concurrente. Por lo tanto, la lógica de reescritura permite razonar sobre cambios complejos en un sistema, teniendo en cuenta que los cambios corresponden a las acciones básicas axiomatizadas por las reglas de reescritura.

2.1.1. Teoría ecuacional. Una *signatura* ordanada por tipos Σ es una tupla $\Sigma = (S, \leq, F)$ con un conjunto parcialmente ordenado de tipos $(S, \leq)$ y un conjunto de símbolos de función F. La relación binaria $\equiv_{\leq}$ denota la relación de equivalencia generada por $\leq$ sobre S y su extensión a cadenas en S^*.

La colección de variables X es una familia S-indexada $X = \{X_s\}_{s \in S}$ de conjuntos de variables disyuntos con cada X_s infinito contable. $T_\Sigma(X)_s$ es el *conjunto de términos de tipo s* y el *conjunto de términos simples de tipo s* se denota por $T_{\Sigma,s}$. Las expresiones $\mathcal{T}_\Sigma(X)$ y $\mathcal{T}_\Sigma$ denotan las correspondientes algebras de Σ-términos ordenadas por tipos.

Una Σ-*ecuación* es una pareja $t = u$ con $t \in T_\Sigma(X)_{s_t}$, $u \in T_\Sigma(X)_{s_u}$ y $s_t \equiv_{\leq} s_u$. Una Σ-*ecuación condicional* es una ecuación condicional $t = u$ **if** γ con $t = u$ una Σ-ecuación y γ una conjunción finita de Σ-ecuaciones. Un *tipo al tope* en Σ es un tipo $s \in S$ tal que si $s' \in S$ y $s \equiv_{\leq} s'$, entonces $s' \leq s$.

Una *teoría ecuacional* es un par (Σ, E), donde Σ es una signatura y E es una colección finita de ecuaciones, posiblemente condicionales.

Una teoría ecuacional $\mathcal{E} = (\Sigma, E)$ induce la relación de congruencia $=_{\mathcal{E}}$ sobre $T_\Sigma(X)$ definida por $t =_{\mathcal{E}} u$, con $t, u \in T_\Sigma(X)$, sí y sólo sí $\mathcal{E} \vdash t = u$ por las reglas de deducción para lógica ecuacional ordenada por tipos en [**8**], sí y sólo sí, en [**8**] $t = u$ es válido en todos los modelos de $\mathcal{E}$.

Las expresiones $\mathcal{T}_{\Sigma/E}(X)$ y $\mathcal{T}_{\Sigma/E}$ corresponden a las algebras de cocientes inducidas por $=_{\mathcal{E}}$ sobre las algebras de términos $\mathcal{T}_\Sigma(X)$ y $\mathcal{T}_\Sigma$. El algebra $\mathcal{T}_{\Sigma/E}$ es llamado el *algebra inicial* de (Σ, E).

Las Σ-ecuaciones están divididas en un conjunto A de axiomas estructurales (tales como asociatividad, conmutatividad y/o identidad) y el conjunto E de ecuaciones.

2.1.2. Teoría de reescritura. Una *teoría de reescritura* es una tupla $\mathcal{R} = (\Sigma, E, R)$ con una teoría ecuacional $\mathcal{E}_\mathcal{R} = (\Sigma, E)$ y un conjunto finito de Σ-reglas R. Una *teoría de reescritura al tope* es una teoría de reescritura $\mathcal{R} = (\Sigma, E, R)$, tal que cada regla $t \to u$ **if** $\gamma \in R$ es tal que $l, r \in T_\Sigma(X)_s$ para algún $s = [s]$ en Σ, $l \notin X$, y no hay operadores en Σ que tengan al tipo s como argumento.

Una teoría de reescritura $\mathcal{R} = (\Sigma, E, R)$ induce la relación de reescritura $\to_\mathcal{R}$ sobre $T_\Sigma(X)$ definida por $t \to_\mathcal{R} u$, con $t, u \in T_\Sigma(X)$, sí y sólo sí una demostración de reescritura de un paso de $\mathcal{R} \vdash t \to u$ puede ser obtenida por las reglas de deducción para teorías de reescritura ordenadas por tipos en [**1**], sí sólo sí, en [**1**] $t \to u$ es válido en todos los modelos de $\mathcal{R}$. La expresión $\mathcal{T}_\mathcal{R} = (\mathcal{T}_{\Sigma/E}, \to^*_\mathcal{R})$ denota el modelo de alcanzabilidad inicial de $\mathcal{R} = (\Sigma, E, R)$ [**1**], donde $\to^*_\mathcal{R}$ representa la clausura transitiva-reflexiva de $\to_\mathcal{R}$.

2.2. Maude

Maude [**3**] es un lenguaje declarativo. Un programa en Maude es una teoría lógica y un cálculo en Maude es una deducción lógica que utiliza los axiomas especificados en la teoría o el programa, según corresponde, bajo el sistema deductivo de la lógica de reescritura [**7**].

Maude cuenta con dos tipos de módulos: funcional y de sistema.

2.2.1. Módulos funcionales. Los módulos funcionales corresponden a teorías ecuacionales y definen tipos de datos y operaciones sobre estos tipos de datos.

Los módulos funcionales de Maude suponen la propiedad de que la ecuaciones son consideradas reglas de simplificación que se usan únicamanete de izquierda a derecha y su repetida aplicación reduce un término a su forma canónica, la cual no depende del orden en el que se usen las ecuaciones. Este tipo de simplificación canónica es posible si la teoría ecuacional asociada a un módulo funcional es Church-Rosser [**4**] y terminante [**6**].

Los módulos funcionales pueden contener: ecuaciones con o sin atributos, pertenencias y operadores con sus respectivos atributos. Las ecuaciones y pertenencias pueden ser condicionales o incondicionales. Un módulo funcional se define con la palabra clave `fmod`. Para más información sobre conceptos básicos y sintaxis de Maude para módulos funcionales se refiere al lector a [**3**].

2.2.2. Módulos de sistema. Un módulo de sistema especifica una teoría de reescritura. Una teoría de reescritura contiene tipos de datos, clases de equivalencia de tipos de datos, operadores, y ecuaciones, pertenencias y reglas de reescritura, las cuales pueden ser condicionales. De lo anterior se puede afirmar que toda teoría de reescritura tiene una teoría ecuacional subyacente. Un módulo de sistema se declara con la palabra clave `mod`. Para más información sobre conceptos básicos y sintaxis de Maude para módulos de sistema se refiere al lector a [**3**].

El conjunto de Σ-reglas R es coherente con respecto a las ecuaciones E módulo A, si Maude puede ejecutar un módulo de sistema admisible [**3**] usando la estrategia de primero simplificar un término t a su forma E/A-canónica y luego usar una regla con R módulo A para lograr el efecto de reescribir con R módulo $E \cup A$.

2.3. *LTL model-checking*

Con *LTL model-checking* es posible demostrar propiedades de lógica lineal temporal para módulos de sistema que tienen una cantidad finita de estados alcanzables desde un estado unicial dado.

Una estructura de Kripke **[2]** es un sistema (total) de transiciones sin etiquetar al cual se le agrega una colección de predicados de estados unarios en su conjunto de estados. Suponiendo que dado un módulo de sistema `M`, el cual especifica una teoría de reescritura $\mathcal{R} = (\Sigma, E, R)$, se tiene:

- elegido un tipo $k = [s]$ en `M` para los estados.
- definido algún conjunto de predicados de estado Π y su semántica en el conjunto de ecuaciones D.

Se define la estructura de Kripke $\mathcal{K}(\mathcal{R}, k)_\Pi$ sobre el conjunto de predicados atómicos AP_Π. Dado un estado inicial $[t] \in T_{\Sigma/E,k}$ y una formula LTL $\varphi \in LTL(AP)_\Pi$ se quiere obtener un procedimiento para decidir la relación

$\mathcal{K}(\mathcal{R}, k)_\Pi, [t] \models \varphi$,

que significa que desde el estado inicial $[t]$ se satisface φ en la estructura de Kripke $\mathcal{K}(\mathcal{R}, k)_\Pi$. En general, esta relación es indecidible pero puede convertirse en decidible si se cumplen las siguientes condiciones:

1. el conjunto de estados en $[t] \in T_{\Sigma/E,k}$ que son alcanzables desde $[t]$ es finito, y
2. la teoría de reescritura $\mathcal{R} = (\Sigma, E, R)$ especificada por el módulo `M` y las ecuaciones D, satisfacen que:
   - E y $E \cup D$ son (simples) Church-Rosser **[4]** y terminante **[6]**, módulo algunos axiomas de A, con $(\Sigma, E) \subseteq (\Sigma \cup \Pi, E \cup D)$ una extensión protegida, y
   - R es (simple) coherente con relación a E, de nuevo, módulo algunos axiomas de A.

Bajo estas suposiciones, los predicados de estados Π y la relación de transición $\rightarrow^1_{\mathcal{R}}$, son calculables y, dada la suposición de alcanzabilidad, se puede resolver el problema de satisfacción usando un procedimiento de *model-checking* **[2]**.

2.4. InvA: El analizador de invariantes de Maude.

Dada una propiedad de estabilidad o invariancia simple φ, InvA genera obligaciones de demostraciones ecuacionales tales que, si estas se satisfacen, entonces $\mathcal{T}_{\mathcal{R}} \models \varphi$ [**11**].

2.4.1. Comandos de InvA. Los comandos de InvA [**11**] disponibles para el usuario son los siguientes:

`(help .):` muestra la lista de comandos disponibles en la herramienta.

`(analyze-stable <pred>in <module><module>.):` genera las obligaciones de pruebas necesarias para demostrar las premisas de inferencia ST con inferencia NR1, para los predicados y módulos dados. Este comando intenta resolver todas las obligaciones de demostraciones; las obligaciones que no se pueden resolver se muestran al usuario.

`(analyze-stable <pred>in <module><module>assuming <pred>.):` genera las obligaciones de pruebas para demostrar la tercera premisa de inferencia STR2 con inferencia NR2, para el predicado y módulos dados. El primer módulo especifica ecuacionalmente los predicados de estados y el segundo la teoría de reescriturura al tope. Este comando intenta resolver todas las obligaciones de demostraciones; las obligaciones que no se pueden resolver se muestran al usuario.

`(analyze <pred>implies <pred>in <module>.):` genera las obligaciones de pruebas para demostrar la implicación en el módulo dado, de acuerdo a la inferencia C⇒. Este comando intenta resolver todas las obligaciones de demostraciones; las obligaciones que no se pueden resolver se muestran al usuario.

`(show pos .):` muestra únicamente la obligaciones de demostraciones calculadas en el último comando `analyze` que no fue posible realizar.

`(show-all pos .):` muestra únicamente la obligaciones de demostraciones calculadas en el último comando `analyze`.

2.4.2. Técnica de InvA. Para una teoria de reescritura $\mathcal{R}$ al tope y el conjunto de predicados Π, la herramienta InvA mecaniza las reglas de inferencia ST, INV, STR1, STR2, NR1 y NR2 [**11**]. Aplicando estas reglas según las especificaciones del usuario, InvA utiliza procedimientos de razonamiento y reducción basados en reescritura, y procedimientos de decisión SMT (i.e., Satisfiability Modulo Theories) para descargar automáticamente tantas obligaciones de demostraciones ecuacionales como sea posible.

Para una especificación ecuacional $\mathcal{E} = (\Sigma, E \cup A)$ y una obligación de prueba condicional φ de la forma

$$t = u \textbf{ if } \gamma,$$

InvA aplica una estrategia de demostración de búsqueda tal que si la ecuación condicional de arriba se satisface, entonces la estructura de Kripke [**2**] asociada al modelo de alcanzabilidad inicial satisface φ. De otra manera, si la demostración de búsqueda falla, se retorna al usuario la obligación de prueba φ o una variante lógicamente equivalente a esta.

En la demostración de búsqueda, InvA primero intenta reducir expresiones Booleanas en φ. El objetivo de la transformación Booleana es obtener inductivamente, si es posible, una obligación de prueba más fuerte φ', para la cual las técnicas deductivas automáticas tienen mayor posibilidad de éxito.

Para más información se refiere al lector a [**11**] y [**12**].

Capítulo 3

El protocolo ESI para la coherencia del caché

El *ESI Cache Coherence Protocol* [**10**] es un protocolo que usa un *controlador* para coordinar *procesos* que requieren acceso a bloques de memoria o líneas de caché. La sigla ESI proviene del ingles "*exclusive*, *share*, *invalid*" e identifica una técnica de coordinación basada en acceso exclusivo o compartido a un recurso, y la posibilidad de remover o invalidar el acceso de un proceso a dicho recurso.

Este capítulo presenta una especificación formal de ESI en la sintaxis de Maude [**3**]. La especificación formal consta de un módulo funcional que define la sintaxis y los tipos requeridos para especificar los estados del protocolo. Este módulo incluye, entre otros, la definición de procesos, identificadores, valores de memoria, etc. Las transiciones de ESI estan formalizadas a partir de reglas de reescritura que son parte de un módulo de sistema.

Adicionalmente, este capítulo presenta ejemplos de algunas simulaciones del protocolo que usan la especificación formal y el comando `search` de Maude.

La Sección 3.1 presenta la descripción del protocolo. Las secciones 3.2 y 3.3 presentan los módulos funcional `ESI-STATE` y de sistema `ESI` que especifican, respectivamente, la sintaxis y las transiciones del protocolo. Finalmente, la Sección 3.4 presenta algunos ejemplos de simulación del protocolo usando el comando `search` de Maude.

3.1. Descripción del protocolo

El sistema *ESI Cache Coherence Protocol* está fundado en la técnica ESI de coordinación basada en acceso exclusivo o compartido a un recurso, y la posibilidad de remover o invalidar el acceso de un proceso a dicho recurso. El protocolo ESI permite a procesos acceder a bloques

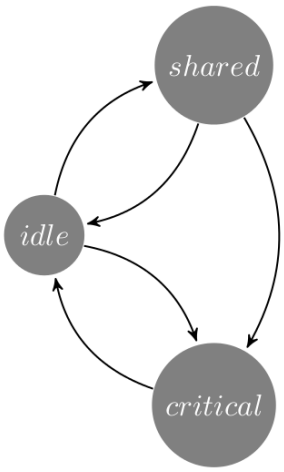

FIGURA 1. Autómata de estados de un proceso.

de memoria o líneas de caché de manera coordinada. Los datos de la memoria se pueden leer si se tiene acceso a ella; el acceso puede ser de tipo *compartido* o *exclusivo*, y lo asigna un controlador. Con acceso compartido únicamente es posible cargar datos desde la memoria y con acceso exclusivo, además, los datos pueden ser almacenados en la memoria. Un proceso también cuenta con una copia local de la memoria; el controlador puede invalidar el acceso de un proceso, lo cual resulta en que los cambios hechos en la copia local del proceso se almacenan en la memoria sí el proceso tiene acceso exclusivo. Cada proceso cuenta con un identificador y una etiqueta para indicar el estado en el que se encuentra. Un proceso puede estar en uno de tres estados: *idle*, *share* o *critical*. Si el estado de un proceso es *idle* indica que el proceso está en espera de recibir un acceso a la memoria, si el estado es *share* indica que el proceso puede leer datos de la memoria (Figura 2) y si el estado es *critical* indica que el proceso puede guardar datos en la memoria (Figura 3).

La Figura 1 muestra los cambios de estado que puede tener un proceso.

La dinámica del protocolo está definida por cinco transiciones que se encargan de entregar, retirar y renovar accesos, y leer y escribir

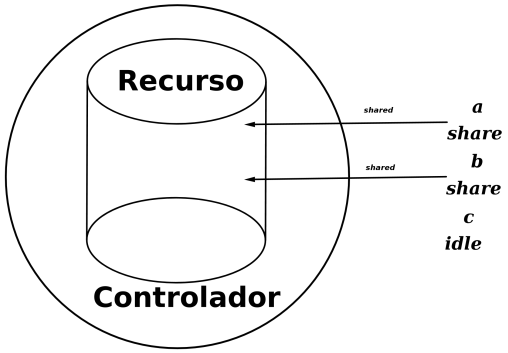

FIGURA 2. Procesos a y b tienen acceso compartido al recurso.

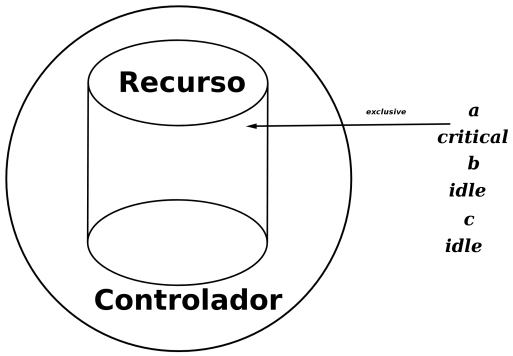

FIGURA 3. Proceso a tiene acceso exclusivo al recurso.

desde y en la memoria. El funcionamiento del protocolo se ilustra a continuación con las transiciones de un proceso:

- Inicialmente el proceso está en espera y se le asigna un acceso exclusivo o compartido.
- Una vez el proceso tiene acceso puede leer datos de la memoria y hacer una copia local.
- Si el proceso tiene acceso exclusivo puede guardar su copia local de datos en la memoria.
- Al proceso se le puede revocar el acceso y queda libre para pedir un nuevo acceso
- ...

Para hacer más explicitas las transiciones de los procesos considere el caso en el que hay tres procesos identificados *a*, *b* y *c*, todos con estado inicial *idle*. En el primer paso cada proceso tiene dos opciones, entrar en estado *share* o en estado *critical*. Suponga que *a* es el primer proceso en recibir un acceso y que su nuevo estado es *share*. Luego *a* lee la memoria, hace una copia local y posteriormente su estado es renovado y pasa a estado *idle*. Por otra parte, si *a* recibe acceso exclusivo, su nuevo estado es *critical* y puede escribir en la memoria o hacer una copia de ella. En cualquier momento, habiendo o no escrito en la memoria, el acceso de *a* puede también ser revocado.

3.2. Modelamiento formal del estado

La especificación formal de ESI presentada en esta sección usa la sintaxis de Maude. Los estados del sistema están representados por el sort `Sys` definido en el módulo funcional `ESI-STATE` de la siguiente manera:

```
sort Sys .
op _:_:_|_ : Nat NatBag NatBag ProcBag -> Sys .
```

Los argumentos de un estado son el valor que contiene la memoria, una bolsa o multiconjunto de identificadores de procesos con acceso compartido o exclusivo a la memoria, una bolsa de identificadores de procesos con acceso exclusivo a la memoria y una bolsa que contiene los procesos que hacen parte del sistema. Cada uno de estos elementos esta representado, respectivamente, por elementos de tipo `Nat`, `NatBag`, `NatBag` y `ProcBag`.

El sort `Nat` representa los números naturales que son usados para identificar procesos y también representar el contenido de la memoria. El sort `Nat` está dado en la notación de Peano de la siguiente manera:

```
op 0  : -> Nat .
op s_ : Nat -> NzNat .
```

El sort `NatBag` es una bolsa de números naturales. Los elementos de la bolsa son identidicadores de los procesos y se usa para identificar aquellos con acceso compartido o exclusivo a la memoria. La bolsa vacía

está representada por `mtpi` y la operación para unir bolsas se expresa por medio de yuxtaposición (i.e., sintaxis vacía):

```
op mtpi : -> NatBag .
op __ : NatBag NatBag -> NatBag [assoc comm id: mtpi] .
```

El sort `ProcBag` es una bolsa de procesos. Cuando la bolsa de procesos es vacía se representa con `mtpr`. Las bolsas no vacías se construyen por yuxtaposición de bolsas de la siguiente manera:

```
op mtpr : -> ProcBag .
op __ : ProcBag ProcBag -> ProcBag [assoc comm id: mtpr] .
```

La operación para construir bolsas no vacías es conmutativa, asociativa y tiene un elemento identidad. En la sintaxis de Maude, la manera de agregar estos atributos a cada símbolo es incluyendo las palabras clave `assoc`, `comm` y `id` en la declaración de las operaciones. Para la primera operación el elemento identidad es `mtpi` y para la segunda es `mtpr`.

Un proceso en el protocolo se representa con el sort `Proc`:

```
op <_,_,_> : Nat Mode Nat -> Proc .
```

Los argumentos de un proceso son el identificador de tipo `Nat`, el estado en el que se encuentra de tipo `Mode` y una copia local de la memoria de tipo `Nat`. El sort `Mode` representa el estado de un proceso, el cual puede tomar uno de los siguientes valores:

```
ops idle share crit : -> Mode .
```

donde `idle` representa que el proceso está en espera, `share` que el proceso tiene acceso compartido y `crit` que el proceso tiene acceso exclusivo.

3.3. Modelamiento formal de las transiciones

Las transiciones del protocolo están modeladas por cinco reglas de reescritura en la sintaxis de Maude. En las transiciones los procesos cambian su estado y se hace la lectura o escritura desde y en la memoria. Las reglas de reescritura se especifican a continuación:

```
rl [fill]:
   Mem:Nat : IDSV:NatBag : mtpi
                 | < ID:Nat, idle, Cac:Nat > PS:ProcBag
   => Mem:Nat : ID:Nat IDSV:NatBag : mtpi
                 | < ID:Nat, share, Cac:Nat > PS:ProcBag .

rl [unfill]:
   Mem:Nat : ID:Nat IDSV:NatBag : IDSE:NatBag
                 | < ID:Nat, share, Cac:Nat > PS:ProcBag
   => Mem:Nat : IDSV:NatBag : IDSE:NatBag
                 | < ID:Nat, idle, Cac:Nat > PS:ProcBag .

rl [fille]:
   Mem:Nat : mtpi : IDSE:Nat
                 | < ID:Nat, idle, Cac:Nat > PS:ProcBag
   => Mem:Nat : ID:Nat : ID:Nat IDSE:Nat
                 | < ID:Nat, crit, Cac:Nat > PS:ProcBag .

rl [flush] :
   Mem:Nat : ID:Nat IDSV:NatBag : ID:Nat IDSE:NatBag
                 | < ID:Nat, M:Mode, Cac:Nat > PS:ProcBag
   => Cac:Nat : IDSV:NatBag : IDSE:NatBag
                 | < ID:Nat, idle, Cac:Nat > PS:ProcBag .

rl [store]:
   Mem:Nat : ID:Nat IDSV:NatBag : IDSE:NatBag
                 | < ID:Nat, M:Mode, Cac:Nat > PS:ProcBag
   => Mem:Nat : ID:Nat IDSV:NatBag : IDSE:NatBag
                 | < ID:Nat, M:Mode, Mem:Nat > PS:ProcBag .
```

El efecto de aplicar cada una de estas reglas se explica a continuación:

`[fill]:` solicitar un acceso compartido a la memoria.

`[unfill]:` retirar el acceso compartido a la memoria.

`[fille]:` solicitar un acceso exclusivo a la memoria.

`[flush]:` revocar el acceso exclusivo y guardar la copia local en la memoria.

`[store]:` hacer una copia local de la memoria.

3.4. Ejemplos

Esta sección muestra cómo usando la especificación de los estados del protocolo en la Sección 3.2 y las reglas de reescritura de la Sección 3.3 se pueden explorar todos los estados alcanzables a partir de un estado inicial dado. Además, esta sección presenta algunas aclaraciones sobre el funcionamiento del protocolo y los posibles estados que se pueden alcanzar.

Considere un estado inicial donde el valor de la memoria principal es 0 y el controlador no ha otorgado acceso (de ningún tipo) al recurso compartido. El estado inicial de los procesos es inactivo.
El estado inicial descrito anteriormente puede ser escrito con la sintaxis dada en la Sección 3.2 de la siguiente manera:

```
0 : mtpi : mtpi | < 1, idle, 31 > < 2, idle, 25 >
                  < 3, idle, 44 >
```

Usando el comando de búsqueda `search` de Maude se puede determinar que hay 979 estados alcanzables a partir de este estado inicial. La exploración de estos estados se puede realizar automáticamente en Maude de la siguiente manera:

```
search in ESI :
       0 : mtpi : mtpi | < 1, idle, 31 > < 2, idle, 25 >
                         < 3, idle, 44 >
       =>* X:Sys .
```

Maude tiene tres tipos de búsqueda representados por `=>1` (i.e., exactamente un paso), `=>+` (i.e., al menos un paso), `=>!` (i.e., estados sin posible avance) y `=>*` (i.e., cualquier cantidad de pasos). En lo que resta de esta sección se usa el comando `=>*` para tratar de encontrar la totalidad de estados que satisfacen el patrón de búsqueda. Para más información se refiere al lector a [**3**]. En este tipo de búsquedas, la expresión `X:Sys` representa cualquier estado alcanzable desde el estado inicial.
Teniendo como estado inicial el estado descrito anteriormente pueden buscarse todos aquellos estados alcanzables en los cuales los procesos con identificadores 1, 2 y 3 tienen acceso compartido a la memoria:

```
search in ESI :
       0 : mtpi : mtpi | < 1, idle, 31 > < 2, idle, 25 >
                         < 3, idle, 44 >
       =>* Mem:Nat : NBV:NatBag : NBE:NatBag |
       < ID0:Nat, share, C0:Nat > < ID1:Nat, share, C1:Nat >
       < ID2:Nat, share, C2:Nat > .
```

Este comando de búsqueda indica que sin importar el valor de la memoria o de las copias locales de los procesos se buscará un estado en el cual `share` sea el estado de los tres procesos.
El resultado de la búsqueda es el siguiente:

```
Solution 1 (state 19)
states: 20  rewrites: 25 in 0ms cpu (0ms real)|
        (~ rewrites/second)
Mem:Nat --> 0
NBV:NatBag --> 1 2 3
NBE:NatBag --> mtpi
ID0:Nat --> 1
C0:Nat --> 31
ID1:Nat --> 2
C1:Nat --> 25
ID2:Nat --> 3
C2:Nat --> 44

...

Solution 534 (state 978)
states: 979  rewrites: 3973 in 72ms cpu (171ms real)
        (55180 rewrites/second)
Mem:Nat --> 0
NBV:NatBag --> 1 2 3
NBE:NatBag --> mtpi
ID0:Nat --> 3
C0:Nat --> 31
ID1:Nat --> 2
C1:Nat --> 31
ID2:Nat --> 1
C2:Nat --> 0

No more solutions.
```

```
states: 979  rewrites: 4005 in 72ms cpu (171ms real)
        (55625 rewrites/second)
```

En total se encontraron 534 estados que satisfacen la condición dada para los procesos. Los estados de la solución son 89; el orden en el que los procesos acceden a la memoria es importante y por esta razón, dada la alta concurrencia en el protocolo, cada uno de los 89 estados es alcanzable seis veces.

Otro cálculo que se puede hacer utilizando el comando `search` es ¿cuántas veces entra un proceso a estado crítico?. Con el siguiente comando se puede responder automáticamente esta pregunta para, por ejemplo, el proceso con identificador 1:

```
search in ESI :
       0 : mtpi : mtpi | < 1, idle, 31 > < 2, idle, 25 >
                         < 3, idle, 44 >
       =>* Mem:Nat : NBV:NatBag : NBE:NatBag |
       < 1, crit, C0:Nat > < 2, M1:Mode, C1:Nat >
       < 3, M2:Mode, C2:Nat > .
```

El resultado de la búsqueda es el siguiente:

```
Solution 1 (state 4)
states: 5  rewrites: 4 in 0ms cpu (0ms real)
        (~ rewrites/second)
Mem:Nat --> 0
NBV:NatBag --> 1
NBE:NatBag --> 1
ID1:Nat --> 2
M:Mode --> idle
C1:Nat --> 25
ID2:Nat --> 3
M1:Mode --> idle|
C2:Nat --> 44
C0:Nat --> 31

...

Solution 178 (state 964)
states: 965  rewrites: 3850 in 24ms cpu (30ms real)
        (160416 rewrites/second)
```

```
Mem:Nat --> 0
NBV:NatBag --> 1
NBE:NatBag --> 1
ID1:Nat --> 3
M:Mode --> idle
C1:Nat --> 31
ID2:Nat --> 2
M1:Mode --> idle
C2:Nat --> 31
C0:Nat --> 0

No more solutions.
states: 979  rewrites: 4005 in 28ms cpu (31ms real)
        (143035 rewrites/second)
```

Esta búsqueda indica que el proceso con identificador 1 tiene acceso exclusivo a la memoria de 178 manera distintas.

Si la misma búsqueda se hace con los procesos con identificadores 2 y 3, estos entrarian en estado crítico la misma cantidad de veces.

Como último ejemplo de la sección considere la búsqueda para calcular cuántos estados hay en los cuales al menos dos procesos tienen acceso exlusivo a la memoria:

```
search in ESI :
       0 : mtpi : mtpi | < 1, idle, 31 > < 2, idle, 25 >
                         < 3, idle, 44 >
       =>* Mem:Nat : NBV:NatBag : NBE:NatBag |
       < ID0:Nat, M0:Mode, C0:Nat > < ID1:Nat, crit, C1:Nat >
       < ID2:Nat, crit, C2:Nat > .
```

El resultado de la búsqueda es el siguiente:

```
No solution.
states: 979  rewrites: 4005 in 32ms cpu (29ms real)
(125156 rewrites/second)
```

Esto indica que desde el estado inicial desde donde parte la búsqueda, no hay estados en los que más de un proceso tenga acceso exclusivo a la memoria en el mismo estado. Es decir, para estos tres procesos y

el estado inicial dado inicialmente se consigue demostrar automáticamente que el protocolo ESI logra exclusión mutua entre procesos para la memoria compartida.

Capítulo 4

Análisis algorítmico de invariantes de ESI

El protocolo ESI para la coherencia del caché debe satisfacer algunas propiedades de *invariancia* (en inglés "*safety*"). Una propiedad de invariancia establece que el sistema nunca va a llegar a un estado erróneo o inseguro.

La principal propiedad de invariancia que debe satisfacer ESI está relacionada con el acceso exclusivo al recurso de memoria compartido por los procesos. En particular, el protocolo debe cumplir la siguiente propiedad de exclusión mutua: a partir de estados iniciales razonables es imposible que dos o más procesos tengan acceso simultáneo al recurso compartido. Adicionalmente, hay otros invariantes que el protocolo debe satisfacer como, por ejemplo, la consistencia de las bolsas de identificadores que mantiene el controlador para otorgar acceso al recurso compartido con los estados de los procesos en el sistema.

En este capítulo se usa Maude para verificar algorítmicamente algunos invariantes de ESI. Los experimentos se hacen predefiniendo una cantidad fija de procesos, y unas condiciones iniciales sobre el estado de la memoria y de los procesos. Posteriormente, se usan el comando `search` de búsqueda y el *LTL model-checker* de Maude [**3**] para analizar los invariantes en mención por medio de la exploración del grafo de estados alcanzables del protocolo. Para un estado inicial dado la cantidad de estados alcanzables es finita y, consecuentemente, los métodos de búsqueda presentados en este capítulo son procedimientos de decisión para verificar propiedades de invariancia de ESI.

Sin embargo, y en general, los métodos de búsqueda algorítmicos están limitados por la explosión combinatoria del conjunto de estados alcanzables en función de la cantidad de procesos iniciales. En este capítulo se incluye un resumen de este fenómeno para ESI que ilustra

cómo la alta concurrencia del protocolo induce un espacio de búsqueda muy grande.

La Sección 4.1 presenta algunas definiciones auxiliares que serán útiles en el anásis algorítmico de invariantes de ESI. Las secciones 4.2 y 4.3 presentan la verificación algorítmica de invariantes del protocolo usando el comando `search` y el *LTL model-checker* de Maude, respectivamente. La Sección 4.4 recopila algunas métricas que indican limitaciones que surgen al analizar algorítmicamente el protocolo ESI.

4.1. Definiciones auxiliares

A continuación se definen y explican cuatro operaciones auxiliares que se utilizarán en el análisis algorítmico del protocolo:

```
op card : NatBag -> Nat .
op in : Nat NatBag -> Bool .
op _===_ : Nat Nat -> Bool [ comm ] .
op subbag : NatBag NatBag -> Bool .
```

El objetivo de definir estas operaciones es:

- `card`: calcular la cantidad de elementos que tiene una bolsa de identificadores.
- `in`: decidir si un identificador pertenece a una bolsa de identificadores.
- `===`: determinar si dos números naturales son iguales.
- `subbag`: saber si una bolsa de identificadores está contenida en otra.

En la declaración de la operación `===` la palabra clave `comm` agrega a la operación la característica de ser conmutativa.

Estas operaciones se definen recurrentemente en Maude de la siguiente manera:

```
vars N N' : Nat .
vars NB NB' : NatBag .

eq card(mtpi) = 0 .
eq card(N NB) = s(card(NB)) .
```

```
eq 0 === 0 = true .
eq N === N = true .
eq s(N) === 0 = false .
eq s(N) === s(N') = N === N' .

eq in(N, mtpi) = false .
eq in(N, N' NB) = (N === N') or in(N,NB) .

eq subbag(mtpi, NB') = true .
eq subbag(N NB, mtpi) = false .
eq subbag(N NB, N NB') = subbag(NB,NB') .
ceq subbag(N NB, NB') = false if not(in(N,NB')) .
```

Las palabras clave `if`, `vars`, `eq` y `ceq` se usan para declarar, respectivamente, condiciones, variables, ecuaciones no condicionales y ecuaciones condicionales en Maude. A continuación se explican las definiciones ecuacionales de estas operaciones:

`card`: la bolsa vacía tiene cardinalidad 0 y en una bolsa no vacía cada uno de sus elementos acumula una unidad.

`_===_`: cualquier número natural `N` es igual a sí mismo, incluyendo el caso cuando los dos números son 0; 0 no es igual a un número distinto de 0; dos números distintos de 0 son iguales si al quitarles una unidad los números resultantes son iguales.

`in`: la bolsa vacía no contiene elemento alguno; un elemento `N` pertenece a una bolsa `NB`, sí `N` es igual a alguno de los elementos de la bolsa `NB`.

`subbag`: la bolsa vacía `mtpi` está contenida en cualquier bolsa; una bolsa no vacía no está contenida en la bolsa vacía; una bolsa no vacía está contenida en otra bolsa no vacía si ambas contienen un elemento en común y recurrentemente, sin ese elemento, las bolsas resultantes satisfacen la propiedad en cuestión.

La operación `_===_` se llama *enriquecimiento de igualdad* del inglés "*equality enrichment*" y sirve para definir inductiva y operacionalmente la igualdad para cualquier tipo abstracto de datos [**5**].

4.2. Verificación de invariantes con el comando `search`

Los invariantes son propiedades que se mantienen en todos los estados alcanzables en una simulación a partir de un conjunto de estados iniciales. En el protocolo ESI para la coherencia del caché se debe garantizar, entre otros, que en cualquier estado en el que se vaya a modificar la memoria sólo un proceso tenga acceso exclusivo a ella, que ningún proceso lea la memoria mientras se está modificando y que cada proceso con acceso exclusivo tenga acceso compartido.
Estos invariantes del protocolo se pueden formalizar de la siguiente manera:

1. $\mid exclusive \mid \leq 1$.
2. $exclusive \subseteq valid$.
3. Si $exclusive \neq$ `mtpi`, entonces $valid = exclusive$

4.2.1. $\mid$ ***exclusive*** $\mid \leq 1$**.** Sí el protocolo satisface esta propiedad se puede afirmar que en ningún estado alcanzable desde un estado inicial dado, más de un proceso intentará escribir en la memoria simultáneamente. Esto garantiza la coherencia del caché con respecto al valor de la memoria en todos los procesos.

Para comprobar que $\mid exclusive \mid \leq 1$ en todos los estados alcanzables, se busca falsificar el predicado tratando de encontrar un estado en el que `card`$(exclusive) > 1$. En Maude, usando el comando `search` se puede hacer la siguiente búsqueda:

```
search in ESI-PROP :
       0 : mtpi : mtpi | < 1, idle, 31 > < 2, idle, 25 >
                         < 3, idle, 44 >
       =>* Mem:Nat : NBV:NatBag : NBE:NatBag | PS:ProcBag
       such that
       card(NBE:NatBag) > 1 .
```

En este tipo de búsquedas se agregan una o más condiciones las cuales se deben cumplir en todos los estados alcanzables. Estas condiciones se especifican escribiendo `such that` al final del comando y enseguida la condición que se va a verificar. Para este caso, se quiere encontrar estados en los que `card(NBE:NatBag)` > 1.

En el comando anterior la bolsa *exclusive* esta representada por `NBE` de tipo `NatBag`. Al final de la exploración el resultado es el siguiente:

```
No solution.
states: 979  rewrites: 6230 in 16ms cpu (14ms real)
        (389375 rewrites/second)
```

Esto significa que desde el estado inicial dado no se encontró un estado en el que el invariante, `card(NBE:NatBag)` ≤ 1, no se mantenga.

Es posible verificar que se cumple el invariante en un grafo que contenga una mayor cantidad de estados. Esto se puede lograr agregando un proceso al estado inicial dado anteriormente

```
0 : mtpi : mtpi | < 1, idle, 31 > < 2, idle, 25 >
                  < 3, idle, 44 > < 4, idle, 12 >
```

Para hacer la búsqueda con el nuevo estado inicial se usa el siguiente comando:

```
search in ESI-PROP :
       0 : mtpi : mtpi | < 1, idle, 31 > < 2, idle, 25 >
                         < 3, idle, 44 > < 4, idle, 12 >
       =>* Mem:Nat : NBV:NatBag : NBE:NatBag | PS:ProcBag
       such that
       card(NBE:NatBag) > 1 .
```

El resultado de la exploración es el siguiente:

```
No solution.
states: 27720  rewrites: 210672 in 548ms cpu
        (550ms real) (384437 rewrites/second)
```

En total hay 27720 estado alcanzables desde el último estado inicial dado que contiene cuatro procesos. En ninguno de estos estados se encuentra que la bolsa *exclusive* contenga más de un elemento.

También se puede hacer la verificación de esta propiedad partiendo de un estado inicial que tenga cinco procesos. El nuevo estado inicial es el siguiente:

```
0 : mtpi : mtpi | < 1, idle, 31 > < 2, idle, 25 >
                  < 3, idle, 44 > < 4, idle, 12 >
                  < 5, idle, 41 >
```

Nuevamente se hace la búsqueda para verificar que se satisface que `card(NBE:NatBag)` ≤ 1, con el siguiente comando:

```
search in ESI-PROP :
       0 : mtpi : mtpi | < 1, idle, 31 > < 2, idle, 25 >
                         < 3, idle, 44 > < 4, idle, 12 >
                         < 5, idle, 41 >
       =>* Mem:Nat : NBV:NatBag : NBE:NatBag | PS:ProcBag
       such that
       card(NBE:NatBag) > 1 .
```

el resultado de la exploración es el siguiente:

```
No solution.
states: 900469  rewrites: 8128558 in 25008ms cpu
        (25006ms real) (325038 rewrites/second)
```

Hay 900469 estados alcanzables desde el estado inicial que tiene cinco procesos, y no se encuentran estados que no cumplan la condición dada para *exclusive*.

4.2.2. ***exclusive*** $\subseteq$ ***valid.*** En esta sección se verifica que todo proceso con acceso exclusivo tenga acceso compartido. Esto se hace con el fin de garantizar que un proceso que tenga acceso exclusivo, previamente haya hecho una copia local. Si esta condición no se cumple el proceso no puede hacer ninguna transición después de recibir un acceso, bien sea, compartido o exclusivo.

Para verificar esta propiedad se hace la siguiente búsqueda, partiendo del estado inicial dado anteriormente:

```
search in ESI-PROP :
       0 : mtpi : mtpi | < 1, idle, 31 > < 2, idle, 25 >
                         < 3, idle, 44 >
       =>* Mem:Nat : NBV:NatBag : NBE:NatBag | PS:ProcBag
       such that
       subbag(NBE:NatBag, NBV:NatBag) == false .
```

La bolsa *valid* se representa con `NBV:NatBag` y la bolsa *exclusive* con `NBE:NatBag`.

En esta exploración se buscan estados en los que *exclusive* no este contenido en *valid*. Utilizando la operación `subbag` de la Sección 4.1 se puede especificar esta condición así:

```
subbag(NBE:NatBag, NBV:NatBag) == false.
```

El resultado de la búsqueda es:

```
No solution.
states: 979  rewrites: 6230 in 96ms cpu
        (95ms real) (64895 rewrites/second)
```

Esto significa que en la exploración no se encontró un estado en el que no se satisfaga la condición dada para *exclusive* y *valid*.

Al igual que en la sección anterior, esta búsqueda se puede hacer con estados inciales que tienen cuatro y cinco procesos. Los comandos para realizar estas exploraciones son los siguientes:

```
search in ESI-PROP :
       0 : mtpi : mtpi | < 1, idle, 31 > < 2, idle, 25 >
                         < 3, idle, 44 > < 4, idle, 12 >
       =>* Mem:Nat : NBV:NatBag : NBE:NatBag | PS:ProcBag
       such that
       subbag(NBE:NatBag, NBV:NatBag) == false .
```

y

```
search in ESI-PROP :
       0 : mtpi : mtpi | < 1, idle, 31 > < 2, idle, 25 >
                         < 3, idle, 44 > < 4, idle, 12 >
                         < 5, idle, 41 >
       =>* Mem:Nat : NBV:NatBag : NBE:NatBag | PS:ProcBag
       such that
       subbag(NBE:NatBag, NBV:NatBag) == false .
```

Los resultados de estas búsquedas son, respectivamente:

```
No solution.
states: 27720  rewrites: 210672 in 700ms cpu
        (701ms real) (300960 rewrites/second)
```

y

```
No solution.
states: 900469  rewrites: 8128558 in 22924ms cpu
        (22924ms real) (354587 rewrites/second)
```

En todos los estados alcanzables a partir de los estados iniciales dados en cada una de las búsquedas se satisface que *exclusive* $\subseteq$ *valid* para los estados iniciales dados.

4.2.3. ***exclusive*** $\neq$ `mtpi` $\wedge$ ***valid*** $=$ ***exclusive.*** Esta propiedad garantiza que si hay un proceso con acceso exclusivo a la memoria, ningún otro proceso tiene acceso compartido o exclusivo a la memoria. Si este invariante se cumple, se puede afirmar que después de que un proceso modifique la memoria, el valor de las nuevas copias locales que hagan los procesos seran iguales.

Para verificar que desde el estado inicial dado, todos los estados alcanzables satisfacen esta condición, se hace la siguiente búsqueda:

```
search in ESI-PROP :
       0 : mtpi : mtpi | < 1, idle, 31 > < 2, idle, 25 >
                         < 3, idle, 44 >
       =>* Mem:Nat : NBV:NatBag : NBE:NatBag | PS:ProcBag
       such that
       NBE:NatBag =/= mtpi and NBV:NatBag =/= NBE:NatBag .
```

En este comando de búsqueda `NBV:NatBag` y `NBE:NatBag` representan, respectivamente, las bolsas *valid* y *exclusive*. Además, la condición que se quiere verificar se especifica con `NBE:NatBag =/= mtpi and NBV:NatBag =/= NBE:NatBag`. El resultado de la exploración es el siguiente:

```
No solution.
states: 979  rewrites: 6942 in 28ms cpu
        (28ms real) (247928 rewrites/second)
```

Agregando los procesos `<4, idle, 12 >` y `<5, idle, 41 >` a los estados iniciales de las búsquedas de la misma manera que se hizo en las secciones anteriores, se obtienen los siguientes resultados:

```
No solution.
states: 27720  rewrites: 232848 in 620ms cpu
```

```
        (621ms real) (375561 rewrites/second)
```

y

```
No solution.
states: 900469  rewrites: 8907342 in 24656ms cpu
        (24655ms real) (361264 rewrites/second)
```

En los tres grafos de ejecución de las búsquedas que parten desde los estados iniciales dados anteriormente, todos los estados alcanzables satisfacen que *valid* = *exclusive* siempre que *exclusive* ≠ `mtpi`.

4.3. Verificación de invariantes con el *LTL model-checker*

En esta sección se utiliza *LTL Model Checking* [**3**] para verificar las propiedades del sistema; esto es posible, al igual que en la Sección 4.2, ya que el conjunto de estados alcanzables desde un estado inicial dado es finito.

Para hacer las verificaciones de las propiedades es necesario definir algunas funciones y ecuaciones, que son la base para utilizar los módulos de sistema y funcionales del archivo `model-checker.maude`. Este archivo incluye las especificaciones en Maude de las fórmulas de LTL y el procedimiento de decisión sobre la veracidad o falsedad para las propiedades temporales del sistema.

Se definen las funciones *idle*, *share* y *crit*, las cuales indican el estado en el que se encuentra un proceso, y las funciones *in-idle*, *in-share* y *in-crit*, que indican cuándo el controlador del sistema supone que un proceso esta en estado *idle*, *share* o *critical*.

La declaración de estas funciones se hace en Maude de la siguiente manera y resulta en que cada una de estas proposiciones hace parte del conjunto AP de predicados atómicos:

```
op idle : Nat -> Prop .
op share : Nat -> Prop .
op crit : Nat -> Prop .

op in-idle : Nat -> Prop .
op in-share : Nat -> Prop .
op in-crit : Nat -> Prop .
```

El argumento de las seis funciones es de tipo `Nat`, ya que se verificarán propiedades sobre identificadores de procesos, los cuales se representan con números naturales.

En estas verificaciones se utiliza la relación de satisfacción

```
op _|=_ : State Prop -> Bool .
```

que permite determinar si estados del sistema satisfacen una propiedad dada. El sort `State` es genérico y se puede manipular usando los *subtipos* de Maude. Con la declaración:

```
subsort Sys < State .
```

se denota que los elementos de tipo `Sys`, que representa los estados del sistema ESI, son además elementos de tipo `State`.
Por otra parte, el sort `Bool` representa los booleanos y el sort `Prop` representa las fórmulas de LTL(AP) [**3**], donde AP es un conjunto de proposiciones o predicados atómicos.

A continuación se presentan ecuaciones para determinar bajo qué condiciones se satisfacen los predicados `idle`, `share`, `crit`, `in-idle`, `in-share` e `in-crit`.

Para estas deficiones ecuacionales se utilizará la siguiente definición de variables

```
vars ID Mem Cac : Nat .
vars IDSV IDSE : NatBag .
var  PS : ProcBag .
```

las cuales representan, respectivamente, identificadores de procesos, valor de la memoria y copia local de la memoria; bolsas *valid* y *exclusive*, y bolsas de procesos.

La ecuación para determinar cuándo el predicado *idle* se satisface en un estado es la siguiente:

```
eq (Mem : IDSV : IDSE | < ID, idle, Cac > PS) |= idle(ID)
 = true .
```

Esta ecuación indica que el predicado *idle* es verdadero para el identificador ID cuando su respectivo proceso está en estado *idle*.

Análogamente se definen ecuaciones para decidir cuándo los predicados `share` y `crit` se satisfacen

```
eq (Mem : IDSV : IDSE | < ID, share, Cac > PS) |= share(ID)
 = true .
eq (Mem : IDSV : IDSE | < ID, crit, Cac > PS) |= crit(ID)
 = true .
```

Las ecuaciones para determinar cuándo el controlador del sistema supone en qué estado esta cada proceso, son:

```
eq (Mem : IDSV : IDSE | PS) |= in-idle(ID)
 = not (in(ID,IDSV) or in(ID,IDSE)) .
eq (Mem : IDSV : IDSE | PS) |= in-share(ID)
 = in(ID,IDSV) and not in(ID,IDSE) .
eq (Mem : IDSV : IDSE | PS) |= in-crit(ID)
 = in(ID,IDSV) and in(ID,IDSE) .
```

Con estas ecuaciones se define que los predicados son verdaderos cuando, respectivamente:

- el identificador de proceso no está en las bolsas de identificadores de procesos que mantiene el controlador
- el identificador está en la bolsa de identificadores de procesos con acceso compartido
- el identificador está en la dos bolsas de identificadores de procesos.

En el módulo de funcional `MODEL-CHECKER` se define la función

```
op modelCheck : State Formula ~> ModelCheckResult .
```

que hace la verificación de una propiedad específica en un grafo que se contruye a partir de un estado inicial dado. Si se satisface la propiedad dada, el resultado de `modelCheck` es `true`, de lo contrario retorna un contraejemplo a la propiedad.

En la verificación de las propiedades se utiliza el comando de Maude `reduce` o `red`, cuya función es reducir el término especificado utilizando las ecuaciones del módulo de sistema `ESI-LTL-PREDS`.

Para verificar las propiedades de consistencia de las bolsas de identificadores que mantiene el controlador para otorgar acceso al recurso

compartido y los estados de los procesos en el sistema, se ejecutan los siguientes comandos en Maude, tomando el estado inicial dado en la Sección 3.4:

```
red modelCheck( 0 : mtpi : mtpi | < 1, idle, 31 >
                                  < 2, idle, 25 >
                                  < 3, idle, 44 >,
                [] (idle(3) ->  in-idle(3))) .
red modelCheck( 0 : mtpi : mtpi | < 1, idle, 31 >
                                  < 2, idle, 25 >
                                  < 3, idle, 44 >,
                [] (share(3) ->  in-share(3))) .
red modelCheck( 0 : mtpi : mtpi | < 1, idle, 31 >
                                  < 2, idle, 25 >
                                  < 3, idle, 44 >,
                [] (crit(3) ->  in-crit(3))) .
```

Los simbolos `[]` y `->` representan, respectivamente, el operador *siempre* ($\Box$) y la *implicación lógica* ($\rightarrow$).
Para $\phi \in$ LTL(AP), $\Box\phi \in$ LTL(AP) puede entenderse como: la fórmula ϕ siempre se satisface en los estados de un grafo de ejecución del protocolo, el cual depende del estado inicial que se fije [**2**].
Así, por ejemplo, para el primer comando de la lista anterior la fórmula significa: *siempre que el estado del proceso con identificador 3 sea idle, debe ocurrir que 3 no pertenece a alguna de las bolsas de identificadores que mantiene el controlador.*

Finalmente, el resultado de las reducciones es:

```
reduce in ESI-LTL-PREDS :
       modelCheck(0 : mtpi : mtpi | (< 2,idle,25 >
                      < 3,idle,44 >) < 1,idle,31 >,
                       [](idle(3) -> in-idle(3))) .
rewrites: 24041 in 32ms cpu (29ms real)
          (751281 rewrites/second)
result Bool: true

reduce in ESI-LTL-PREDS :
       modelCheck(0 : mtpi : mtpi | (< 2,idle,25 >
                      < 3,idle,44 >) < 1,idle,31 >,
                     [](share(3) -> in-share(3))) .
```

```
rewrites: 21104 in 36ms cpu (33ms real)
          (586222 rewrites/second)
result Bool: true

reduce in ESI-LTL-PREDS :
       modelCheck(0 : mtpi : mtpi | (< 2,idle,25 >
                      < 3,idle,44 >) < 1,idle,31 >,
                     [](share(3) -> in-share(3))) .
rewrites: 21104 in 36ms cpu (33ms real)
          (586222 rewrites/second)
result Bool: true
```

El resultado que retorna `modelCheck` en las tres verificaciones es `true`. Por consiguiente, en todos los estados alcanzables desde el estado inicial dado, se mantiene la coherencia entre las bolsas *valid* y *exclusive* mantenidas por el controlador y los estados de los procesos del sistema.

La verificación de estas propiedades se puede hacer en grafos que se construyen desde estados iniciales que tienen una mayor cantidad de procesos, por ejemplo, los estados iniciales

```
0 : mtpi : mtpi | < 1, idle, 31 > < 2, idle, 25 >
                  < 3, idle, 44 > < 4, idle, 12 >
```

y

```
0 : mtpi : mtpi | < 1, idle, 31 > < 2, idle, 25 >
                  < 3, idle, 44 > < 4, idle, 12 >
                  < 5, idle, 41 >
```

que se utilizaron en la Sección 4.2, generan grafos que en total tienen 27720 y 900469 estados alcanzables. La verificación de los invariantes en dichos grafos se hace cambiando los estados iniciales en el comando utilizado para el estado inicial con tres procesos y se obtienen los siguientes resultados:

```
reduce in ESI-LTL-PREDS :
       modelCheck(0 : mtpi : mtpi |
             ((< 3,idle,44 > < 4,idle,12 >) < 2,idle,25 >)
               < 1,idle,31 >,
             [](idle(3) -> in-idle(3))) .
rewrites: 814979 in 956ms cpu (952ms real)
```

```
          (852488 rewrites/second)
result Bool: true

reduce in ESI-LTL-PREDS :
       modelCheck(0 : mtpi : mtpi |
             ((< 3,idle,44 > < 4,idle,12 >) < 2,idle,25 >)
               < 1,idle,31 >,
             [](share(3) -> in-share(3))) .
rewrites: 731819 in 1100ms cpu (1104ms real)
          (665290 rewrites/second)
result Bool: true

reduce in ESI-LTL-PREDS :
       modelCheck(0 : mtpi : mtpi |
             ((< 3,idle,44 > < 4,idle,12 >) < 2,idle,25 >)
               < 1,idle,31 >,
             [](crit(3) -> in-crit(3))) .
rewrites: 676379 in 1368ms cpu (1366ms real)
          (494429 rewrites/second)
result Bool: true
```

y

```
reduce in ESI-LTL-PREDS :
       modelCheck(0 : mtpi : mtpi |
             (((< 4,idle,12 > < 5,idle,41 >) < 3,idle,44 >)
                < 2,idle,25 >) < 1,idle,31 >,
             [](idle(3) -> in-idle(3))) .
rewrites: 31394741 in 40760ms cpu (40761ms real)
          (770234 rewrites/second)
result Bool: true

reduce in ESI-LTL-PREDS :
       modelCheck(0 : mtpi : mtpi |
             (((< 4,idle,12 > < 5,idle,41 >) < 3,idle,44 >)
                < 2,idle,25 >) < 1,idle,31 >,
             [](share(3) -> in-share(3))) .
rewrites: 28693334 in 39548ms cpu (39548ms real)
          (725531 rewrites/second)
result Bool: true

reduce in ESI-LTL-PREDS :
```

```
          modelCheck(0 : mtpi : mtpi |
                (((< 4,idle,12 > < 5,idle,41 >) < 3,idle,44 >)
                   < 2,idle,25 >) < 1,idle,31 >,
                [](crit(3) -> in-crit(3))) .
rewrites: 26892396 in 38268ms cpu (38268ms real)
          (702738 rewrites/second)
result Bool: true
```

Las verificaciones de la sección se hicieron para el identificador 3, pero cabe aclarar que los invariantes también se satisfacen para los demás identificadores y la razón por la que no se incluyen en el documento es porque se pueden hacer de manera análoga.

4.4. Limitaciones de la verificación algorítmica

La verificación algorítmica de las propiedades que se hizo en las secciones 4.2 y 4.3, es problemática cuando la cantidad de procesos en los estados iniciales aumenta y por consiguiente, la cantidad de estados alcanzables es muy grande y no se puede garantizar, en un tiempo y con un uso de recursos razonables, que las propiedades verificadas en los grafos sean ciertas.

Recordando las soluciones obtenidas en los experimentos de la Sección 4.2, se pueden consolidar los tiempos que tardó el sistema en verificar algorítmicamente el invariante dado. Por ejemplo, para la propiedad de la Subsección 4.2.1, los tiempos empleados en las búsquedas con 3, 4 y 5 procesos en el estado inicial son, respectivamente, 0.016 s, 0.5 s y 25 s. En este experimento se observa cómo, aumentando sólo un proceso al estado inicial, los tiempos de exploración se incrementan considerablemente. En las propiedades de las subsecciones 4.2.2 y 4.2.3 los tiempos de búsqueda, aunque diferentes a los tiempos de la Subsección 4.2.1, aumentan en una proporción muy similar conforme aumenta la cantidad de procesos en el estado inicial.

La Tabla 1 contiene los datos de algunos experimentos en los que se utiliza el comando `search` de Maude. En estos experimentos se varía la cantidad de procesos en el estado inicial desde donde parte cada búsqueda. Los experimentos se hacen con comandos de la forma:

Cantidad de procesos	Estados alcanzables	Transiciones	Tiempo (ms)
1	9	42	0
2	60	360	4
3	979	7298	24
4	27720	255024	620
5	900469	10075518	28408
6	-	-	Excedido

FIGURA 1. Datos de búsquedas aumentando cantidad de procesos en el estado inicial

```
search in ESI-PROP : 0 : mtpi : mtpi | PSvar:ProcBag
       =>* Mem:Nat : NBV:NatBag : NBE:NatBag | PS:ProcBag
       such that card(NBV:NatBag) < 0 .
```

En el estado inicial aparece `PSvar:ProcBag` para indicar que la cantidad de procesos en los estados iniciales va a cambiar. En la condición del comando se pide verificar que `card(NBV:NatBag) <0`, lo cual nunca es posible; esto se hace para evitar que se muestren en pantalla todos los estados alcanzables desde el estado inicial dado, que en este caso es irrelevante.

El tiempo límite de ejecución que se fijó para los experimentos es de 12 minutos. Para el experimento con seis procesos en el estado inicial, terminado este tiempo no se obtiene una solución.

Aunque la Tabla 1 contiene datos de verificaciones que utilizan el comando `search`, para las verificaciones que usan el comando `modelCheck`, el grafo que se explora es el mismo, por lo tanto, la verificación de los invariantes de la Sección 4.3 emplearán tiempos proporcionales a los incluidos en la tabla.

Capítulo 5

Análisis deductivo de un invariente de ESI

El análisis algorítmico de invarientes está limitado por el tamaño del espacio de búsqueda y por la cantidad de estados iniciales.
Si alguno de estos conjuntos es muy grande o hasta infinito, entonces el análisis algorítmico es poco práctico o incluso imposible.

El análisis decuctivo, en general, no depende del tamaño de los conjuntos mencionados anteriormente. Utilizando estas técnicas se pueden analizar invariantes en sistemas cuyos conjuntos de estados son infinitos.

En esta sección se utiliza la técnica de verificación deductiva de invariantes propuesta por C. Rocha y J. Meseguer para lógica de reescritura **[12, 13]**. Esta técnica consiste en reducir el razonamiento formal temporal sobre transiciones concurrentes a razonamiento ecuacional inductivo y no está limitado por la cantidad de estados iniciales o alcanzables.

En este capítulo se demuestra mecánicamente un invariante de ESI como ejemplo de la potencia de la técnica de análisis simbólico de invariantes y las herramientas que los apoyan.

Este capítulo está dividido en cuatro secciones: la Sección 5.1 presenta la especificación formal del invariante, la Sección 5.2 presenta la verficación mecánica del invariante y, por último, la Sección 5.3 contiene un ejemplo de un invariante no inductivo (i.e., un invariante que no se puede demostrar automáticamente sin ayuda de invariantes auxiliares).

5.1. Especificación formal del invariante

Para mostrar el funcionamiento de la herramienta InvA, se verifica que en cualquier estado alcanzable desde un estado inicial arbitrario, los

identificadores de procesos no se repiten. Este invariante se representa con e siguiente predicado:

```
ops good-ids init : Sys -> [Bool] .
```

El objetivo de verificar esta propiedad es garantizar que no se generan confusiones en la asignación de accesos al recurso compartido.

Las funciones auxiliares definidas a continuación hacen parte del módulo de sistema `ESI-PROP`

```
op set : NatBag -> Bool
op exids : ProcBag -> Bool
```

Con estas operaciones, respectivamente:

- se decide si una bolsa de números naturales es un conjunto, y
- se extraen los identificadores de los procesos que hacen parte del sistema

Estas operaciones se definen recurrentemente en Maude de la siguiente manera:

```
eq set(mtpi) = true .
eq set(N NB) = not(in(N,NB)) and set(NB) .

eq exids(mtpr) = mtpi .
eq exids(< ID, M, Cac > PS) = ID exids(PS) .
```

La anterior definición ecuacional se puede entender así:

set: la bolsa vacía es un conjunto y en un conjunto no se repiten elementos.

exids: la bolsa de procesos vacía genera la bolsa vacía de identificadores y una bolsa de procesos no vacía genera una bolsa de identificadores de procesos extrayendo un proceso y uniendo su identificador a la bolsa de identificadores y recurrentemente, sin ese proceso, se hace el mismo procedimiento para las bolsas resultantes.

Finalmente, las ecuaciones con las que el controlador del sistema supone que no se repiten identificadores, son las siguientes:

```
eq init(Mem : mtpi : mtpi | PS)
```

```
  = set(exids(PS))

eq good-ids(Mem : IDSV : IDSE | PS)
 = set(exids(PS))
```

5.2. Verificación mecánica del invariante

Esta sección presenta la verificación mecánica del invariante formalizado en la Sección 5.1, utilizando la herramienta InvA de la Sección 2.4.

La propiedad de invariancia $\mathcal{T}_{\mathcal{R}} \models$ `init` $\Rightarrow \Box$`good-ids`, significa que siempre que en $\mathcal{T}_{\mathcal{R}}$ se satisface `init` para algún estado alcanzable $[t]_E \in T_{\Sigma/E,\mathfrak{s}}$ del sistema, entonces `good-ids` se satisface en cualquier estado alcanzable $[t']_E \in T_{\Sigma/E,\mathfrak{s}}$, tal que $[t]_E \rightarrow_{\mathcal{R}} [t']_E$ **[11]**.
Como el conjunto de estados iniciales está definido en $\mathcal{E}_{AP}$ por el predicado de estados `init` $\in$ AP, la definición ecuacional de `init` involucra conjuntos de estados infinitos (con 0 procesos, 1 proceso, 2 procesos, etc.).

Para verificar que desde cualquier estado inicial se satisface el invariante `good-ids`, se utiliza el comando `analize` de InvA, Subsección 2.4.1, de la siguiente manera:

```
(analyze init(S:Sys) implies good-ids(S:Sys) in ESI-PREDS .)
```

con InvA se genera 1 obligación y es descargada automáticamente, como se ve a continuación:

```
rewrites: 6132 in 20ms cpu (20ms real)
          (306600 rewrites/second)
Checking
   ESI-PREDS ||- init(S:Sys) => good-ids(S:Sys) ...
Proof obligations generated:  1
Proof obligations discharged: 1
Success!
```

Ahora, para verificar que en cualquier estado alcanzable se mantiene el invariante `good-ids`, se ejecuta el comando

```
(analyze-stable good-ids(S:Sys) in ESI-PREDS ESI .)
```

y se obtiene el siguiente resultado:

```
rewrites: 281039 in 172ms cpu (170ms real)
          (1633947 rewrites/second)
Checking
   ESI-PREDS ||- good-ids(S:Sys) => O good-ids(S:Sys) ...
Proof obligations generated:  20
Proof obligations discharged: 12
The following proof obligations need to be discharged
```

La herramienta InvA genera 20 obligaciones de las cuales 12 se descargan automáticamente. Las 8 obligaciones restantes son las siguientes:

```
5. from store : pending
 good-ids(#5:Nat : #9:Nat mtpi : #6:NatBag | #10:ProcBag
          < #9:Nat,#7:Mode,#5:Nat >) = true
 if set(exids(#10:ProcBag < #9:Nat,#7:Mode,#8:Nat >)) = true .

7. from store : pending
 good-ids(#5:Nat : #9:Nat #10:NatBag : #6:NatBag |#11:ProcBag
          < #9:Nat,#7:Mode,#5:Nat >) = true
 if set(exids(#11:ProcBag < #9:Nat,#7:Mode,#8:Nat >)) = true .

9. from unfill : pending
 good-ids(#5:Nat : mtpi : #6:NatBag | #9:ProcBag
          < #8:Nat,idle,#7:Nat >) = true
 if set(exids(#9:ProcBag < #8:Nat,share,#7:Nat >)) = true .

11. from unfill : pending
 good-ids(#5:Nat : #9:NatBag : #6:NatBag | #10:ProcBag
          < #8:Nat,idle,#7:Nat >) = true
 if set(exids(#10:ProcBag < #8:Nat,share,#7:Nat >)) = true .

13. from flush : pending
 good-ids(#7:Nat : mtpi : mtpi | #9:ProcBag
          < #8:Nat,idle,#7:Nat >) = true
 if set(exids(#9:ProcBag < #8:Nat,#6:Mode,#7:Nat >)) = true .

15. from flush : pending
 good-ids(#7:Nat : mtpi : #9:NatBag | #10:ProcBag
          < #8:Nat,idle,#7:Nat >) = true
 if set(exids(#10:ProcBag < #8:Nat,#6:Mode,#7:Nat >)) = true .
```

```
17. from flush : pending
 good-ids(#7:Nat : #10:NatBag : #9:NatBag | #11:ProcBag
          < #8:Nat,idle,#7:Nat >) = true
 if set(exids(#11:ProcBag < #8:Nat,#6:Mode,#7:Nat >)) = true .
19. from flush : pending
 good-ids(#7:Nat : #9:NatBag : mtpi | #10:ProcBag
          < #8:Nat,idle,#7:Nat >) = true
 if set(exids(#10:ProcBag < #8:Nat,#6:Mode,#7:Nat >)) = true .
```

Para demostrar que las obligaciones anteriores se satisfacen, se verifica que, por ejemplo, para la obligación 19 el siguiente comando evalua `true`

```
red set(exids(#10:ProcBag < #8:Nat,#6:Mode,#7:Nat >))
    implies
    good-ids(#7:Nat : #9:NatBag : mtpi | #10:ProcBag
             < #8:Nat,idle,#7:Nat >) .
```

Lo que se verifica con este comando es:
Si

```
set(exids(#10:ProcBag < #8:Nat,#6:Mode,#7:Nat >)) = true
```

entonces

```
good-ids(#7:Nat : #9:NatBag : mtpi | #10:ProcBag
         < #8:Nat,idle,#7:Nat >) = true
```

El resultado de la ejecución es el siguiente:

```
reduce in ESI-PREDS :
       set(exids(#10:ProcBag < #9:Nat,#7:Mode,#8:Nat >))
       implies
       good-ids(#5:Nat : #9:Nat : #6:NatBag | #10:ProcBag
                < #9:Nat,#7:Mode,#5:Nat >) .
rewrites: 16 in 0ms cpu (0ms real)
          (~ rewrites/second)
result Bool: true
```

De manera análoga se puede verificar que las otras obligaciones se pueden descargar, y finalmente se asegura que `good-ids` se satisface en todos los estados alcanzables del sistema.

5.3. Ejemplo de un invariante no inductivo

En este ejemplo el objetivo es verificar que no hay estados en los que dos o más procesos tengan acceso simultáneo al recurso compartido. Como se hizo en la Sección 5.1, ahora se define el predicado `good-crit`, de la siguiente manera:

```
op good-crit : Sys -> [Bool] .

eq good-crit(Mem : IDSV : mtpi | PS)
 = true .
eq good-crit(Mem : IDSV : ID | PS)
 = in(ID, IDSV) .
eq good-crit(Mem : IDSV : ID ID' IDSE | PS)
 = false .
```

La anterior definición ecuacional indica cuándo el controlador supone que el invariante se satisface, y pueden ser entendidas, respectivamente, así:

- Si no hay procesos con acceso exclusivo, el invariante se satisface.
- Si sólo hay un proceso con acceso exclusivo, el invariante se satisface.
- Si hay al menos dos procesos con acceso exclusivo, el invariante no se satisface (y hay un fallo en el sistema).

El comando para verificar que `good-crit` se satisface es el siguiente:

```
(analyze init(S:Sys)implies good-crit(S:Sys) in ESI-PREDS .)

(analyze-stable good-crit(S:Sys) in ESI-PREDS ESI .)
```

cuya respuesta es:

```
rewrites: 5873 in 28ms cpu (27ms real)
          (209750 rewrites/second)
Checking
  ESI-PREDS ||- init(S:Sys) => good-crit(S:Sys) ...
Proof obligations generated:  1
Proof obligations discharged: 1
```

```
Success!


rewrites: 73157 in 88ms cpu (86ms real)
          (831329 rewrites/second)
Checking
  ESI-PREDS ||- good-crit(S:Sys) => O good-crit(S:Sys) ...
Proof obligations generated:  88
Proof obligations discharged: 84
The following proof obligations need to be discharged:

6. from unfill : pending
   false = true
   if #6:Nat === #8:Nat = true .
7. from unfill : pending
   false = true
   if #6:Nat === #8:Nat = true .
8. from unfill : pending
   in(#6:Nat,#9:NatBag) = true
   if in(#6:Nat,#8:Nat #9:NatBag) = true .
9. from unfill : pending
   in(#6:Nat,#9:NatBag) = true
   if in(#6:Nat,#8:Nat #9:NatBag) = true .
```

De las 88 obligaciones generadas por InvA, 84 se descargan automáticamente. Las 4 obligaciones restantes indican que `good-crit` no es un invariante inductivo, ya que estas obligaciones no se pueden descargar automáticamente. Por ejemplo, en la obligación número 9 no se puede garantizar que `in(#6:Nat,#9:NatBag)` sea verdadera, ya que no se sabe si `#6:Nat === #8:Nat` se satisface.

Este invariante puede demostrarse automáticamente por medio de un fortalecimiento de invariantes, pero esto sale del alcance de la tesis. Para más información se refiere al lector a [**11, 12**].

Capítulo 6

Conclusión

La lógica de reescritura es una lógica para especificar sistemas concurrentes la cual, al ser combinada con Maude y sus herramientas, facilita la verificación mecánica de las propiedades que debe satisfacer el sistema. El análisis algorítmico de invariantes, aunque limitado por la cantidad de estados iniciales y alcanzables desde cada estado inicial, es un método automático para verificar propiedades de invariancia.

Se verificaron invariantes de exclusión mutua entre procesos para el protocolo ESI utilizando *model checking* y el comando `search` de Maude. También, se demostraron invariantes inductivos por medio de métodos deductivos. La verificación deductiva fue utilizada para demostrar mecánicamente propiedades de sistemas con una cantidad infinita de estados lo cual es imposible haciendo uso de métodos algorítmicos como los mencionados anteriormente.

En esta tesis se presenta una especificación formal del protocolo ESI para la coherencia de caché en el lenguaje de la lógica de reescritura. Dicha especificación es ejecutable con el sistema Maude para lógica de reescritura.

Como trabajo futuro se sugiere utilizar técnicas de abstracción [**3, 9**] para verificación algorítmica de invariantes y así garantizar de manera más general la satisfabilidad de los invariantes del Capítulo 4. Además, la definición de nuevos invariantes para ser demostrados por medio de técnicas deductivas y el uso de fortalecimiento de invariantes para aquellos que resulten no inductivos.

Apéndice A

Especificación formal en Maude

Este anexo incluye el módulo funcional `ESI-STATE` y los módulos de sistema: `ESI`, `ESI-PROP`, `ESI-LTL-PREDS` y `ESI-PREDS`.

A.1. ESI-STATE

```
fmod ESI-STATE is
  pr NAT .

  sorts NatBag Mode Proc ProcBag Sys .

  subsort Nat < NatBag .
  subsort Proc < ProcBag .

  ops idle share crit : -> Mode .
  op <_,_,_> : Nat Mode Nat -> Proc .
  op mtpr : -> ProcBag .
  op __ : ProcBag ProcBag -> ProcBag [assoc comm id: mtpr] .
  op mtpi : -> NatBag .
  op __ : NatBag NatBag -> NatBag [assoc comm id: mtpi] .
  op _:_:_|_ : Nat NatBag NatBag ProcBag -> Sys .
endfm
```

A.2. ESI

```
mod ESI is
  pr ESI-STATE .

  var  M          : Mode .
  vars Cac ID Mem : Nat .
  vars IDSE IDSV  : NatBag .
  var  PS         : ProcBag .
```

```
  rl [fill]:
     Mem : IDSV : mtpi | < ID, idle, Cac > PS
     =>
     Mem : ID IDSV : mtpi | < ID, share, Cac > PS .

  rl [unfill]:
     Mem : ID IDSV : IDSE | < ID, share, Cac > PS
     =>
     Mem : IDSV : IDSE | < ID, idle, Cac > PS .

  rl [fille]:
     Mem : mtpi : IDSE | < ID, idle, Cac > PS
     =>
     Mem : ID : ID IDSE | < ID, crit, Cac > PS .

  rl [flush] :
     Mem : ID IDSV : ID IDSE | < ID, M, Cac > PS
     =>
     Cac : IDSV : IDSE | < ID, idle, Cac > PS .

  rl [store]:
     Mem : ID IDSV : IDSE | < ID, M, Cac > PS
     =>
     Mem : ID IDSV : IDSE | < ID, M, Mem > PS .
endm
```

A.3. ESI-PROP

```
mod ESI-PROP is
  pr ESI .

  op card : NatBag -> Nat .
  op in : Nat NatBag -> Bool .
  op _===_ : Nat Nat -> Bool [ comm ] .
  op subbag : NatBag NatBag -> Bool .
  op set : NatBag -> Bool .
  op exids : ProcBag -> NatBag .

  var  N N'       : Nat .
  var  NB NB'     : NatBag .
  var  M          : Mode .
  vars ID Mem Cac : Nat .
```

```
  vars IDSV IDSE  : NatBag .
  var  PS         : ProcBag .

  --- Cardinality
  eq card(mtpi) = 0 .
  eq card(N NB) = s(card(NB)) .

  --- Two equal natural numbers
  eq    N === N = true .
  eq    0 === 0 = true .
  eq s(N) === 0 = false .
  eq s(N) === s(N') = N === N' .

  --- Bag contains element
  eq in(N, mtpi)  = false .
  eq in(N, N' NB) = (N === N') or in(N,NB) .

  --- Bag contained in another bag
  eq subbag(mtpi, NB')   = true .
  eq subbag(N NB, mtpi)  = false .
  eq subbag(N NB, N NB') = subbag(NB,NB') .
 ceq subbag(N NB, NB')   = false if not(in(N,NB')) .

  --- Set
  eq set(mtpi) = true .
  eq set(N NB) = not(in(N,NB)) and set(NB) .

  --- Removing identifiers
  eq exids(mtpr) = mtpi .
  eq exids(< ID, M, Cac > PS) = ID exids(PS) .
endm
```

A.4. ESI-LTL-PREDS

```
mod ESI-LTL-PREDS is
  pr ESI-PROP .
  inc SATISFACTION .
  inc MODEL-CHECKER .
  inc LTL-SIMPLIFIER .

  subsort Sys < State .
```

```
  op idle  : Nat -> Prop .
  op share : Nat -> Prop .
  op crit  : Nat -> Prop .

  op in-idle  : Nat -> Prop .
  op in-share : Nat -> Prop .
  op in-crit  : Nat -> Prop .

  vars Cac ID Mem : Nat .
  vars IDSE IDSV  : NatBag .
  var  PS         : ProcBag .
  var  M          : Mode .

  eq (Mem : IDSV : IDSE | PS) |= in-idle(ID)
   = not (in(ID,IDSV) or in(ID,IDSE)) .
  eq (Mem : IDSV : IDSE | PS) |= in-share(ID)
   = in(ID,IDSV) and not in(ID,IDSE) .
  eq (Mem : IDSV : IDSE | PS) |= in-crit(ID)
   = in(ID,IDSV) and in(ID,IDSE) .

  eq (Mem : IDSV : IDSE | < ID, idle, Cac > PS) |= idle(ID)
   = true .
  eq (Mem : IDSV : IDSE | < ID, share, Cac > PS) |= share(ID)
   = true .
  eq (Mem : IDSV : IDSE | < ID, crit, Cac > PS) |= crit(ID)
   = true .
endm
```

A.5. ESI-PREDS

```
mod ESI-PREDS is

  pr ESI-PROP .

  ops init good-ids good-crit : Sys -> [Bool] .

  var  M              : Mode .
  vars ID Mem Cac ID' : Nat .
  vars IDSV IDSE NB   : NatBag .
  var  PS             : ProcBag .
```

```
  --- Initial states
  eq init(Mem : mtpi : mtpi | PS)
   = set(exids(PS)) .

  --- IDs are not repeated
  eq good-ids(Mem : IDSV : IDSE | PS)
   = set(exids(PS)) .

  --- Acceso exclusivo
  eq good-crit(Mem : IDSV : mtpi | PS)
   = true .
  eq good-crit(Mem : IDSV : ID | PS)
   = in(ID, IDSV) .
  eq good-crit(Mem : IDSV : ID ID' IDSE | PS)
   = false .
endm
```

Bibliografía

[1] R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *Theoretical Computer Science*, 360(1-3):386–414, 2006.

[2] E. M. Clarke, O. Grumberg, and D. Peled, editors. *Model Checking*. MIT Press, 1999.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[4] F. Durán and J. Meseguer. On the church-rosser and coherence properties of conditional order-sorted rewrite theories. *The Journal of Logic and Algebraic Programming*, 81(7-8):816–850, 2012. Rewriting Logic and its Applications.

[5] R. Gutiérrez, J. Meseguer, and C. Rocha. Order-sorted equality enrichments modulo axioms. In F. Durán, editor, *Rewriting Logic and Its Applications*, volume 7571 of *Lecture Notes in Computer Science*, pages 162–181. Springer Berlin Heidelberg, 2012.

[6] S. Lucas and J. Meseguer. Operational termination of membership equational programs: the order-sorted way. *Electronic Notes in Theoretical Computer Science*, 238(3):207 – 225, 2009. Proceedings of the Seventh International Workshop on Rewriting Logic and its Applications (WRLA 2008).

[7] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73 – 155, 1992. Selected Papers of the 2nd Workshop on Concurrency and Compositionality.

[8] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Presicce, editor, *Recent Trends in Algebraic Development Techniques*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer Berlin Heidelberg, 1998.

[9] J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. *Theoretical Computer Science*, 403(2–3):239 – 264, 2008.

[10] S. Ray. *Scalable Techniques for Formal Verification*. Springer, 2010.

[11] C. Rocha. *Symbolic Reachability Analysis for Rewrite Theories*. PhD thesis, University of Illinois at Urbana-Champaign, 2012.

[12] C. Rocha and J. Meseguer. Proving safety properties of rewrite theories. In A. Corradini, B. Klin, and C. Cîrstea, editors, *Algebra and Coalgebra in Computer Science*, volume 6859 of *Lecture Notes in Computer Science*, pages 314–328. Springer Berlin / Heidelberg, 2011.

[13] C. Rocha and J. Meseguer. Proving safety properties of rewrite theories. In A. Corradini, B. Klin, and C. Cîrstea, editors, *Algebra and Coalgebra in Computer Science*, volume 6859 of *Lecture Notes in Computer Science*, pages 314–328. Springer Berlin Heidelberg, 2011.